A Mobile Multi-Agent Autonomic Architecture for an Electronic Marketplace Application

by

Jing Zhou
B. Eng., Beijing University of Aeronautics and Astronautics, 1996

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

SUPERVISORY COMMITTEE

A Mobile Multi-Agent Autonomic Architecture for an Electronic Marketplace Application

By

Jing Zhou
B. Eng., Beijing University of Aeronautics and Astronautics, 1996

Supervisory Committee

Dr. Hausi A. Müller, Department of Computer Science

Supervisor

Dr. Bruce Kapron, Department of Computer Science

Departmental Member

Dr. Frank Roberts, Department of Computer Science

Departmental Member

Dr. Kin Fun Li, Department of Electrical & Computer Engineering

External Examiner

Supervisory Committee

Dr. Hausi A. Müller, Department of Computer Science
Supervisor

Dr. Bruce Kapron, Department of Computer Science
Departmental Member

Dr. Frank Roberts, Department of Computer Science
Departmental Member

Dr. Kin Fun Li, Department of Electrical & Computer Engineering
External Examiner

Abstract

Computing systems' complexity appears to be approaching the limits of human capabilities. To cope with the rapidly growing complexity of operating, managing and integrating computing systems, one of the most promising options is Autonomic Systems, which are computing systems that can manage themselves given high-level objectives from administrators. Self-management means that the system self-configures at run time to increase responsiveness and agility, self-heals to improve business resiliency, self-optimizes to improve operational efficiency and self-protects itself from malicious attacks. While traditional approaches to computer systems are often centralized and hierarchical, Autonomic Systems are highly distributed with complex connectivity and interactions, rendering centralized management schemes infeasible. In this thesis, we describe the design and implementation of a decentralized architecture based on mobile multi-agent systems—an approach to building Autonomic Systems. Based on the proposed architecture, we developed a prototype application—an electronic marketplace, which achieves a set of desired features of Autonomic Systems such as autonomy, hiding complexity and self-healing. We put forth Autonomic Systems, a self-managing distributed computing system, as a new and promising application domain for multi-agent system ideas and argue that an Agent-based approach is well suited to construct Autonomic Systems.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank everyone who supported me and offered me guidance throughout this research.

In particular, I am grateful to my supervisor, Dr. Hausi Müller, for his guidance, support, and encouragement throughout my graduate studies. He has been a great source of inspiration and has provided me with an admirable role model. He has also created an excellent research environment without which I would not have been able to complete this work.

I would like to thank my co-workers in the Rigi group who helped tremendously: Scott Brousseau, Sangeeta Neti, Priyanka Agrawal, Sweta Goyal, Qin Zhu, Feng Zou and Toni Lin.

Finally, I would like to thank my family and friends for all their encouragement and support.

Dedication

To my parents

Chapter 1 Introduction

1.1 Motivation

Today, even small IT environments have many interrelated components. Complexity arises from the interdependencies among these components, which increase exponentially as you add hardware and software to your IT architectures (servers, networks, storage systems, and management systems), upgrade facilities, and expand the scope of processes and organizational structures.

Up until now, we have relied mainly on human intervention and administration to manage this complexity. With millions of interdependencies, many are difficult to identify, some are difficult to understand, and others are outside our domain of control. Recent trends have made it clear that software complexity will continue to increase dramatically in the coming decades. The complexity of large-scale information systems has outstripped our ability to develop and manage them [Jones2002]. Thus, Horn's

statement that "the largest obstacle to further progress in the IT industry is a looming software complexity crisis [Horn2001]" rings true. Today's systems must evolve to become much more self-managing to reduce the dependence on human intervention in order to deal with "the single most important challenge facing the I/T industry: complexity [Horn2001]."

Wladawsky-Berger proposed the solution at the Kennedy Consulting Summit in November 2001: "There is only one answer: the technology needs to manage itself. Now, I do not mean any far out AI project; what I mean is that we need to develop the right software, the right architecture, the right mechanisms … So that instead of the technology behaving in its usual pedantic way and requiring a human being to do everything for it, it starts behaving more like the "intelligent" computer we all expect it to be, and starts taking care of its own needs. If it does not feel well, it does something. If someone is attacking it, the system recognizes it and deals with the attack. If it needs more computing power, it just goes and gets it, and it does not keep looking for human beings to step in [Wla2001]."

The dynamic and distributed nature of both data and applications require that software not only respond to requests for information but intelligently anticipate, adapt, and actively seek ways to support users as well. Autonomic Computing is just the next logical evolution of these past trends to address the increasingly complex and distributed computing environments of today [GC2003].

Autonomic Computing is used to design, develop, deploy and manage systems by using strategies inspired by the autonomic nervous systems of the human body to cope with complexity, heterogeneity, and uncertainty [GC2003].

An Autonomic System is an autonomous computing environment that hides its complexity. Complexity hiding from users/services means that Autonomic Computing will provide users with a computing environment that allows them to concentrate on what they want to do without worrying about how it has to be done. In other words, tasks are realized without conscious recognition or significant effort on the side of the users. Autonomic Systems need to handle and manage an ever increasingly complex environment consisting of software, hardware and communication infrastructure.

An Autonomic System has four fundamental capabilities [GC2003] [KC2003] [Horn2001]:

- Self-Configuration: An Autonomic Systems needs to be able to configure and reconfigure itself under varying and unpredictable conditions.
- Self-Optimizing: The system will continually monitor and tune its resource and operations. More generally, the system will continually seek to optimize its operation with respect to a set of prioritized non-functional requirements to meet the ever-changing needs of the enterprise.
- Self-Healing: The system will be able to recover—without loss of data or noticeable delays in processing—from routine and extraordinary events that might cause some of its parts to malfunction.
- Self-Protecting: The system will be capable of protecting itself by detecting and counteracting threats through the use of pattern recognition and other techniques such as firewall and intrusion-detection tools.

1.2 Objective and Approach

An Autonomic System is a self-managing system which means that the system can self-configure at run-time to meet changing operating environments, self-optimize to optimize its performance, self-heal when it encounters unexpected obstacles during its operation, and self-protect itself from malicious attacks. The focus of this research is to build such software systems so that they can meet given requirements for particular classes of users to make the process of building Autonomic Systems more systematic and less error-prone.

While traditional approaches to computer systems management are often centralized and hierarchical, Autonomic Systems are highly distributed with complex connectivity and interactions [KC2003], rendering centralized management schemes infeasible. In this thesis, we investigate a Multi-Agent Systems (MAS) approach and apply the technology to build Autonomic Systems. Multi-Agent Systems are well suited for Autonomic Systems because MAS offers rich sources of techniques for a wide variety of interaction mechanisms that can be beneficially incorporated into the design of Autonomic Systems.

Jennings advocates an agent-based approach to software engineering based on decomposing problems in terms of decentralized, autonomous agents that can engage in flexible, high-level interactions [Jennings2000]. This approach is particularly well-suited for Autonomic Systems.

In addition, Kephart and Chess state that "autonomy, proactivity, and goal-directed interactivity with their environment are distinguishing characteristics of software agents. Viewing autonomic elements as agents and autonomic systems as multi-agent systems make it clear that agent-oriented architectural concepts will be critically important [KC2003]." Moreover, "many ideas developed in the MAS community, such as those pertaining to automatic group formation, emergent behavior, multiagent adaptation, and agent coordination, among others, could likely be fruitfully adapted for Autonomic Computing [TCWD2004]."

In the light of the above quotation, an investigation of how suitable Multi-Agent Systems are for building Autonomic Systems becomes very interesting. The major objective of this research is to investigate the existing concepts, theories and technologies related to Multi-Agent Systems for building Multi-Agent Autonomic Systems.

Firstly, we investigate the properties of individual agents; then we discuss their extensions to multiple agents, in particular the interaction and communication among self-interested agents that can possibly be adapted for Autonomic Systems.

Our goal in the design of Multi-Agent Autonomic Systems is to propose an architecture for building Autonomic Systems that supports this approach. We show how the architecture addresses a subset of the main characteristics of Autonomic Systems: autonomy, complexity-hiding and self-healing.

A prototype application was implemented to demonstrate the viability of our approach. By creating an implementation of the architecture, we show that Multi-Agent Autonomic Systems are realizable.

1.3 Outline of the Thesis

This chapter discussed the motivation for this research, described the problem being focused on, detailed our research objectives, and outlined our approach to solving this problem.

Chapter 2 provides background and an introduction to Agents, Mobile Agents, Mobile Agent Platforms, Multi-Agent Systems and the communication language used among agents to interact. These techniques are employed in building multi-agent based Autonomic Systems.

Chapter 3 reviews three of the most important works related to our research. These examples of systems provide a general yet powerful solution to build multi-agent based Autonomic Systems.

Chapter 4 describes how Mobile Multi-Agent Autonomic Architecture can be developed to support the building of Autonomic Systems. The requirements of such a system are given and then the details of a framework are outlined.

Chapter 5 details the implementation of a prototype application, a multi-agent based electronic marketplace which is constructed based on the Mobile Multi-Agent Autonomic Architecture.

Chapter 6 includes our evaluations of the prototype application and summarizes a set of our development experiences and lessons learned.

Finally, Chapter 7 presents a final overview of the research that has been undertaken, summarizes the contributions of this work, and proposes possible directions for future research.

Chapter 2 Background

This chapter provides some background as a baseline for our research and reviews some essential knowledge about mobile multi-agent systems. It is divided into five parts describing key issues in the design of a mobile multi-agent based Autonomic System. The first part introduces the basic properties of Agents; the second part talks particularly about Mobile Agents; the third part talks about the purpose of mobile agent platform and compares several mobile agent platforms. Part 4 focuses on Multi-Agent Systems, which consist of a number of agents that interact with each other to achieve desired goals, and the language—Agent Communication Language (ACL)—needed for this interaction; the last part centers on a particular ACL—Knowledge Query and Manipulation Language (KQML).

2.1 Agents

An agent is the basic component of Multi-Agent Systems. Agents are viewed as next-generation software components, which offer greater flexibility, adaptability and autonomy than traditional software components [GP2001]. At present, there is a great deal of ongoing debate about exactly what constitutes an agent, yet there is nothing approaching a universal consensus. According to the commonly used definition, an agent is "an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives [Wooldridge1997]." The amount of flexibility in the agent's autonomy is determined by three factors. The first factor is the agent's reactivity to its environment. This depends on the agent's ability to perceive and respond to its surroundings. The second factor is the level of pro-activeness or initiative that the agent takes in meeting its design goals. The final factor is the level of social interaction between agents.

Agents exhibit a combination of several of the following characteristics [GP2001]:

- Autonomous: The agent proactively initiates activities in accord with its goal, has its own thread of control, and can act on its user's behalf, largely independent of messages other agents send.
- Adaptable: Either its own learning, user customization, or downloading new capabilities can change an agent's behavior after deployment.

- Knowledgeable: The agent can reason about its goals, acquired information, and knowledge about other agents and users.
- Mobile: The agent can move from one executing context to another, either by moving its code and starting afresh or by serializing its code and state, continuing execution in a new context and retaining its state to continue its work.
- Collaborative: The agent can communicate and work cooperatively with other agents to form dynamic or static multi-agent societies, collaborating to perform a task.
- Persistent: The infrastructure enables agents to retain knowledge and state over extended periods of time, including robustness in the face of possible runtime failures.

An agent is a highly autonomous goal-directed entity capable of performing actions on behalf of users without external intervention. The introduction of agents represents a fundamental change in the way users will interact with information; the emphasis will be moved away from a user interactively working with a piece of software to achieve a task to a user delegating the completion of a task to an agent.

Furthermore, agent related methods and technologies provide a basis for engineering autonomic, self-managing, self-healing and self-tuning systems that are able to dynamically adjust and reconfigure themselves in response to changes in their environment [Pour2003]. This suggests that agents can have a useful and important role to play within Autonomic Systems.

2.2 Mobile Agents

Mobile agents are a category of agents whose predominant feature is the ability to move between nodes on a network or between nodes across networks. Mobile agents may be thought of as programming entities that can freely roam the network and act on behalf of their users [MLC1999]. A slightly more technical definition states that "mobile agents are programs, typically written in a script language, which may be dispatched from a client computer and transported to a remote server computer for execution" [HCK1996].

The mobile agent approach extends the network computing mechanism by transferring executable code to the server (including its state or data) and behavior (execution logic) where it is then executed. Each mobile agent is a computation along with its own data and execution state. After its submission, the mobile agent proceeds autonomously and independently of the sending client. When the agent reaches a server, it is delivered to an agent execution environment. Then, if the agent possesses necessary authentication credentials by using trust-management system [BFL1996], its executable parts are started. To accomplish its task, the mobile agent can transport itself to another server, spawn new agents, or interact with other agents. Upon completion, the mobile agent delivers the results to the sending client or to another server [SSPE2004].

Mobile agent is one such paradigm for building distributed systems which has drawn a lot of attention in both academia and industry [QS2004]. Mobile agent systems provide the benefits of agent migration, multi-agent communication and negotiation [SDPV2005]. Kotz and Rus describe that the identification of key features such as persistence, resource allocation, orphan detection, state capture, security, communication

and co-ordination is important to ensure the efficient use and successful deployment of mobile agents [KGR2002].

A mobile agent is an agent, who has the capability of mobility in computer networks. Besides the agent properties mentioned above, mobility is a desirable characteristic in agents for a number of reasons, which can realize advantages in Autonomic Systems [LO1999], [HCK1996]:

- Mobile agents reduce the network load and allow you to package a conversation and dispatch it to a destination host, where the interactions can take place locally (see Figure 2.1). Mobile agents are also useful when it comes to reducing the flow of raw data in the network. When very large volumes of data are stored at remote hosts, these data should be processed in the locality of the data rather transferred over the network.

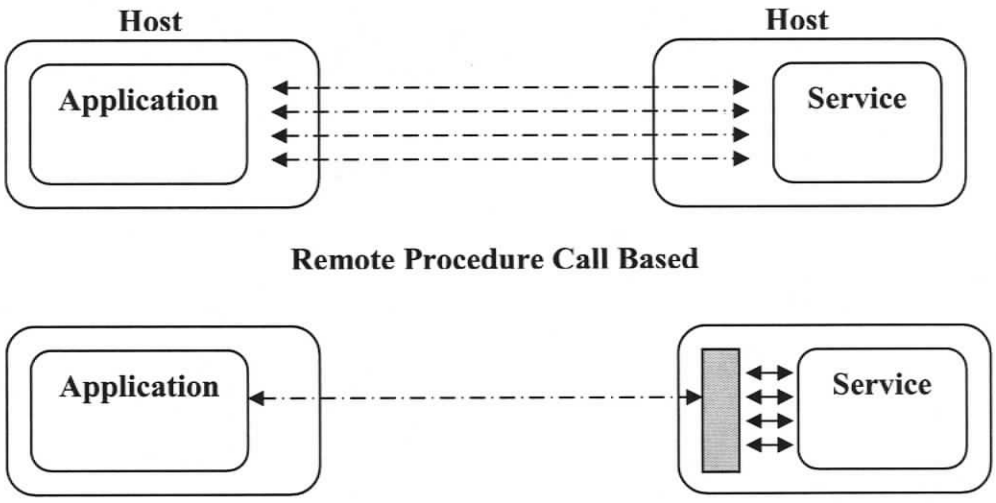


Figure 2.1 Mobile Agents and Network Load Reduction

- Mobile agents can overcome network latency. Critical real-time systems need to respond in real time to changes in their environments. Mobile agents offer a solution, because they can be dispatched from a central controller to act locally and directly execute the controller's directions.
- Mobile agents are naturally heterogeneous. Because mobile agents are generally computer and transport-layer-independent and are dependent only on their execution environment, they provide optimal conditions for seamless system integration.
- Mobile agents are robust and fault-tolerant. The nature of mobile agents makes it easier to build robust and fault-tolerant distributed systems. If a host is being shut down, all agents executing on that machine will be warned and given time to dispatch and continue their operation on another host in the network.

In addition to the distinguishing properties of agents, mobile agents can overcome challenges that are difficult for normal software to surmount, for example, movement across networks, platform heterogeneity, reduction of network traffic, support for disconnected operation. In light of these benefits associated with mobile agents, adopting mobile agents will confer numerous advantages in building Autonomic Systems.

2.3 Mobile Agent Platforms

Mobile agents are created by a distributed application at one computer site and launched to another site using an underlying mobile-agent platform. An instance of the platform running at the remote site can receive the mobile agent and dispatch it to the distributed application running at that site. Several mobile agent platforms have been proposed. They can be broadly categorized as Java and non-Java based ones. There is an increasing interest in those that are Java-based due to the inherent advantages of Java including platform independence, simple network communication, highly secure program execution, multithreaded programming, and object serialization. [HCMKK2000][GHMCKT2003]. These features of Java make the creation of a mobile agent a fairly simple task.

There are several Java-based mobile agent platforms available including IBM's Aglets Workbench [Aglets1996] ObjectSpaces's Voyager [Voyager1997], Mitsubishi's Concordia [WPWD1997], IKV++ GrassHopper [BBCM1998] and General Magic's Odyssey [White1996].

In this thesis we introduce four of the common mobile agent platforms: Aglets, Voyager, Odyssey and Concordia [HCMKK2000]:

- Aglet is nicely adapted to the Internet environment; it is robust and the most widely used system.
- Voyager provides the unique concept of serialization and mobility of objects and allows quick and easy creation of sophisticated network applications.

- Odyssey is more distributed systems oriented than other mobile agent systems but brings a new dimension to the programming.
- The Concordia system provides modularity, security and enables remote administration that makes it suitable for enterprise systems.

While all these systems provide the basic functionality expected from such mobile platforms, they differ significantly in their system architecture, the communication mechanism employed, the additional functionality they provide and their performance [SSPE2004].

2.4 Multi-Agent Systems (MAS)

The development of Multi-Agent Systems (MAS) greatly increases the scope of applications in which agents can be involved [Weiss1999]. The technology moved from working with a single concept or application using an individual agent to having numerous agents, each performing a specialized purpose. These related applications can be combined in a distributed system designed to meet much larger and more complex goals.

Stone and Veloso outlined the need for and usefulness of MAS [SV2000]:

- The most important reason to use MAS when designing a system is that some domains require it. In particular, if there are different people or organizations with different (possibly conflicting) goals and proprietary information, then MAS is needed to handle their interactions.

- Having multiple agents could speed up a system's operation by providing a method for parallel computation. For instance, a domain that is easily broken into components—several independent tasks that can be handled by separate agents—could benefit from MAS. Furthermore, the parallelism of MAS can help deal with limitations imposed by time-bounded reasoning requirements.
- Robustness is a benefit of Multi-agent Systems that have redundant agents. If control and responsibilities are sufficiently shared among different agents, the system can tolerate the failure from one or more agents.
- Another benefit of MAS is their scalability. Since they are inherently modular, it is easier to add new agents to Multi-agent Systems than it is to add new capabilities to a monolithic system. Systems whose capabilities and parameters are likely to need to change over time or across agents can also benefit from the modularity of MAS.

MAS can meet different people or organization's goals, speed up the system's operation, make the system more robust, and enhance the system's scalability. These features are also required by Autonomic Systems. Moreover, incorporating MAS will greatly improve the performance of Autonomic Systems. These properties enable Autonomic Systems to be one of the most significant and potentially valuable application domains for Mobile Multi-Agent Systems.

The critical difference between Multi-Agent Systems and an individual agent focuses on the patterns of communication. A Multi-Agent System communicates with the application and the user, as well as with the other agents in the system to achieve their objectives. The most important aspect of Multi-Agent Systems is the communication

between the agents. Many protocols have been developed that give the agents the ability to both receive and send information to each other. This type of information depends on the exact nature of the agent in question.

Although the agent may reside within the multi-agent environment, they do not necessarily have to work together. Certain agents will have self-serving agendas and will compete or collaborate with other agents in the system to meet their goals. Other agents will be working within a group construct and will collaborate or compete with other agents to meet the group goals.

The emphasis of MAS centers on incorporating a multi-agent system with inter-agent communication. In the next section, we introduce languages used for agent communication.

2.5 Agent Communication Language (ACL)

Agents in a Multi-Agent System must be able to "talk" to each other to decide what action to take and how this action can be coordinated with others' actions. The language used by the agents for this exchange is the Agent Communication Language (ACL). The main objective of an ACL is to model a suitable framework that allows heterogeneous agents to interact, to communicate with meaningful statements that convey information about their environment or knowledge [KSN2000].

KQML (Knowledge Query and Manipulation Language) [FLM1997] is one of the most widely used ACLs in Multi-Agent Systems [BNS2004]. The language has been very

successful in facilitating the communication and coordination of software agents in a variety of domains including organizational decision making, financial management, and even aircraft maintenance [BNS2004].

KQML is a high-level protocol and language for agent-to-service and agent-to-agent interaction and communication. It is based on a notion derived from Speech Act theory [Austin1975] [Searle1969] [Searle1979] in which inter-agent communication is thought of as similar to conversations between humans. KQML considers that each message not only contains the contents but also the "intention" the sender has for that contents.

KQML provides a formal specification for representing the intentions of messages through a set of predefined message types (called performatives) used in the messages. It includes many performatives of speech acts, all assertiveness (i.e., when it states a fact) or directives (i.e., when it reflects a command or request), which agents use to assert facts, request information or subscribe to services [CD2002].

A KQML message is divided into three layers: the content layer, the message layer, and the communication layer. The content layer bears the actual content of the message, in a language chosen by the sending agent. The communication layer encodes a set of features of the message that describe the lower-level communication parameters such as the identity of the sender and recipient, and a unique identifier associated with the communication. The message layer encodes the message, including its intention (by a chosen performative), the content language used and the ontology. The syntax of KQML

messages is based on a balanced parenthesis list. The first element of this list is the performative; the remaining elements are the arguments of the performative, as keyword/value pairs.

The following is an example of an actual KQML message sent by agent "joe" to agent "book-market," inquiring about the price of a particular book:

```
( request
  :sender       joe
  :receiver     book-market
  :language     XML
  :content      <book>
                <title>Computing Concepts with Java 2 Essentials</title>
                <isbn>0-471-34609-8</isbn>
                </book>
  :reply-with:  joe-1234 )
```

In the message, the performative request indicates that this is a query type of a message. XML is specified as the content language. The actual content is an XML fragment indicating the book title and ISBN number. The string *joe-1234* is a unique machine-generated reply ID. In essence, with this message agent *joe* asks for one answer from agent *book-market* for the price of this particular book.

Agent *book-market* might send *joe* the following KQML message if or when it has an answer to the query:

```
( tell
  :sender        book-market
  :receiver      joe
  :language      XML
  :in-reply-to   joe-1234
  :content       <book>
                 <title>Computing Concepts with Java 2 Essentials</title>
                 <isbn>0-471-34609-8</isbn>
                 <price>$80.95</price>
                 </book> )
```

Agent *book-market* indicates the price of the book is $80.95 in the reply message.

The advantages of using ACL and standard information representations are evident; if all agents within the system conform to an interchange standard, then the amount of data translation is reduced and the introduction of translation errors into the data is reduced.

2.6 Summary

This chapter introduced some key concepts and techniques regarding Agents, Mobile Agents, Mobile Agent Platforms, Multi-Agent Systems, and the Agent Communication Languages. To build Autonomous Systems using agent technology, these concepts must be well understood. With the aid of Mobile Multi-Agent Systems technology and related techniques, the effort of building Autonomic Systems can be greatly reduced. In the next chapter, a few principal systems related to our research are reviewed and discussed.

Chapter 3 Related Work

In this chapter we review and discuss three of the most important bodies of work related to our research: Unity [CSWW2004] [TCWD2004] [CKSW2004], the Federated Multi-Agent System [GHC2004], and a self-healing system based on multi-agent technologies [PYL2005]. These systems were chosen because they are excellent examples of successful applications in their particular domain. Our work is built upon the strengths and ideas espoused by these systems.

3.1 Unity

The Unity project was carried out at IBM Thomas J. Watson Research Center. It is a prototype Autonomic System demonstrating and validating a number of ideas about how self-management can be achieved in practice. The aspects of self-management in Unity

include self-configuration and self-optimization; the system initially configures parts of itself with a minimal amount of explicit human input, and during operation it reallocates and reconfigures part of its resources to optimize its behavior according to human-specified policies [CSWW2004] [TCWD2004] [CKSW2004].

Unity is a distributed computing system for autonomic computing which is structured as a set of individual elements. The application environment manager element is responsible for the management of the environment to obtain the resources that the environment needs and for communicating with other elements. The resource arbiter element is capable of deciding which resources from the finite pool should be assigned to which application environment. The registry element enables each element to locate other elements with which it needs to communicate. The policy repository element supports interfaces that allow the human administrators of the system to enter the high-level policies that guide the operation of the system. The sentinel element supports interfaces that allow one element to ask the sentinel to monitor the functioning of another. Finally, the solution manager element represents the "solution" as a whole to the outside world.

Every component is implemented as an autonomic element via interaction amongst a population of autonomous agent. It explores component behaviors, relationships and technologies that will support self-management of complex computing systems. Each autonomic element is a component that manages its own behavior in accordance with policies and interacts with other autonomic elements to provide or consume

computational services. It is responsible for its own internal autonomic behavior, including managing the resources that it controls, and managing its own internal operations, including self-configuration, self-optimization, self-protection, and self-healing. Each element is also responsible for forming and managing the relationships that it enters into with other autonomic elements whose services it uses or who use its service in order to accomplish its goals.

Unity realizes a number of desired Autonomic Systems behaviors: the ability for the system to self-configure at runtime initialization using goal-driven self-assembly; to develop software design patterns that enable self-healing within the system; and to self-optimize in real time the use of distributed computational resources in accordance with the system's overall business objectives.

3.2 The Federated Multi-Agent System

The federated multi-agent system is constructed for the autonomic organization and control of web services by taking advantage of agent federation and nested agent federation [GHC2004]. The concept of federation, by analogy with federal countries, combines central decision making (federal rules) with local autonomy (agents' decision-making capabilities). Each cooperation and coordination agent group is called a federation. The degree of autonomy varies depending on the specificities of the participating components. To meet different requirements, various levels of agent federations can dynamically cooperate to fulfill specific tasks or goals.

All agents in this multi-agent system can be divided into two categories based on their functions: task agents and management agents. According to the goals and rules of the specific agent federation, several task agents are recruited into an agent federation. In each agent federation, there is a management layer that consists of several management agents, which take care of various management issues such as registration, cooperation and transaction management.

The layers of a federated multi-agent system are depicted in Figure 3.1.

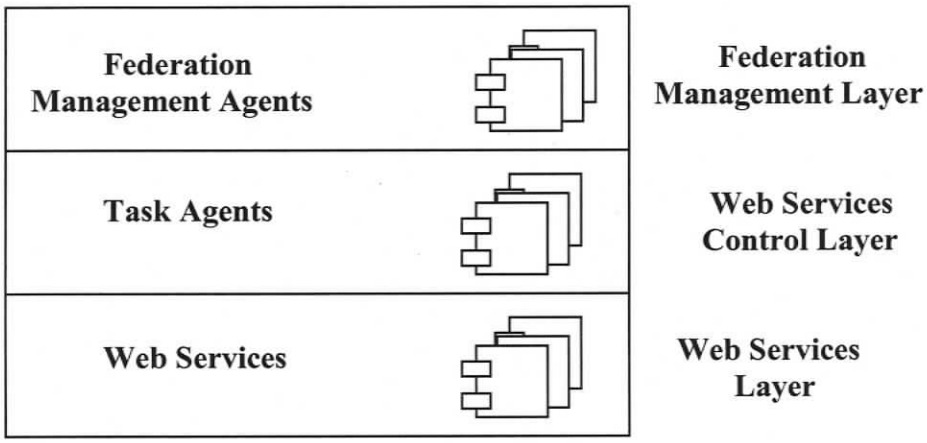


Figure 3.1 Layered Perspective of a Federated Multi-Agent System

The web services control layer consists of various task agents, which take care of the fine-grained control of web services. There are a number of management components in this layer including the information services component (ISC) and the web services composition component (WSCC). The ISC advertises the capabilities and properties of the task agent to the federation management agent. The WSCC organizes and composes the web services according to dynamic requirements from the management agent.

The federation management layer deals with two kinds of issues: internal membership management and external business management. This module is composed of a number of agents, which have various functions. The internal management agent manages registrations of federation members and directs cooperation of members; the advertisement agent deals with service advertisement to update and provide the services available; and the transaction agent seeks service, negotiation and cross-federation cooperation.

To fulfill a new task, several task agents corresponding to related divisions of the task into the federation. If the internal agent federation cannot satisfy the requirements of this task, the transaction agent (TA) of the federation will look for qualified candidates from external companies. After negotiation, the internal management agent (IMA) of the federation will direct the cooperation of member agents. If the requirements of this task changes or some of the current member agents go wrong or cannot fulfill the subtasks, the TA will look for new candidates and the IMA will recruit alternatives to implement the task.

3.3 Proactive Self-healing System Based on Multi-agent Technologies

Park et al. proposed a system consisting of multi-agents that analyze the log context, error events and resource status in order to perform self-healing and self-diagnosis [PYL2005].

The system is comprised of several agents, which have a variety of functions in each phase of the self-healing process:

- Monitoring Agent (Monitoring Phase): monitors the resource, deals with errors or problems arising in the component that do not generate log events.
- Component Agent (Filtering and Translation Phase): gathers the information provided by the Monitoring Agent; filters error context from normal log context File; translates the filtered error context into the CBE format; takes immediate action corresponding to emergency situation.
- System Agent (Execution Phase): when the System Agent receives the CBE log, the Resource Collector gathers all the related information and delivers it to the Adaptation Module, which is in possession of the threshold values pertaining to the gathered resource information. According to the threshold value, a suitable policy is implemented.
- Diagnosis Agent (Analysis and Diagnosis Phase): analyzes the CBE log and the dependency of the components, and then diagnoses the current problem. The results of the diagnosis can be used to trigger automated reactions.
- Decision Agent (Decision and Feedback Phase): through the information delivered by the Diagnosis Agent, the decision agent determines the appropriate healing method with the help of the Policy DB.
- Searching Agent: used to search the vendor's website for the knowledge to solve the problem.

The approach consists of a 5-step process to heal its own internal problems by using a number of agents, which play various roles in the system. The procedure of self-healing can be described as follows:

- First, the Monitoring Agent, which runs as a single process, provides real time monitoring of error events.
- Second, if problems or errors arising in systems result in an emergency situation, the Component Agent can take proactive and immediate action.
- Third, the System Agent using threshold values of the system resources recognizes system situations and chooses an applicable policy on the Meta level which offers different healing strategies according to the components situation.
- Fourth, before translating the original log into the log of the Common Base Event (CBE) type, a filtering process is performed.
- Finally, in the event that there is no applicable method of healing, the Searching Agent searches the web server of the vendor in order to interact with the administrator.

This multi-agent based self-healing system enables a computer system to observe, diagnose and heal errors or problems. Especially, the technology that monitoring agent monitors error event and resource status the generation of the log on the fly can be employed in building multi-agent based Autonomic Systems.

3.4 Comparison and Discussion

Unity represents an important step in building Autonomic Systems based on Multi-Agent Systems. It focuses on investigating the internal behaviors of each autonomic element and properties of each element should possess. Each element must be capable of managing itself, handling problems locally and establishing relationships with the other autonomic elements whose services it uses or who use its services abide by its policies. Unity is a valuable platform for studying and testing ideas about Autonomic Systems. It provides us with some valuable and helpful information for our research.

An agent federation based approach was proposed to simplify the control of the internal and external coordination and transaction of web services. The autonomic service discovery, negotiation, and cooperation in an open, dynamic environment has been conducted successfully. However, this approach has limitation. It has not provided monitoring and treatment mechanisms to supervise the contract fulfill process when service provider and service consumer deal with a specific web service. Moreover, its centralized approach limits its applicability in Autonomic Systems.

The Multi-agent based proactive self-healing system minimizes the resources required through the use of a single process (Monitoring Agent) and select the appropriate healing policy by using a Meta Policy which offers different healing strategies according to the components' situation. The approach allows the system or computing device itself to recognize, identify and heal problems arising without depending on an administrator.

All these approaches provide a very useful source of ideas for developing Autonomic Systems. However, in contrast to these other efforts, the problem that we study in this thesis can be seen as a variation of them. We focus on investigating Mobile Multi-Agent Systems and attempt to propose a fully distributed approach based on Mobile Agents to build Autonomic Systems. Moreover, we try to implement a prototype of our system in a mobile agent platform which brings more benefits to develop Autonomic Systems.

3.5 Summary

Building Autonomic Systems based on Multi-agent Systems is becoming a well-received research practice. Some research projects focus on designing and implementing every autonomic element as Unity, some intend to group the coordinating and cooperating agents into federation to improve the flexibility like federated multi-agent systems, some try to achieve a self-healing system that monitors, diagnoses and heals its own problems based on multi-agent systems. These different approaches provide valuable experiences in decision-making, system architecture design and development process analysis and evaluation of mobile-agent based autonomic Multi-Agent Systems. The next chapter describes the architecture for a Mobile Multi-Agent Autonomic Systems that supports this approach.

Chapter 4 General Approach and Methodology of a Solution

The general approach of this research is to develop Autonomic Systems in terms of Mobile Multi-Agent Systems by investigating the theories and principles of mobile agent system design, and applying them to build Autonomic Systems. This chapter presents architecture based on the Mobile Multi-Agent technologies and a detailed description of each type of static agent and its functionalities within the architecture, which achieves some features such as autonomy, complexity-hiding and user's intervention reduction. Moreover, we describe how this architecture implements one of the most important characteristics of Autonomic Systems—self-healing.

4.1 Mobile Multi-Agent Autonomic Architecture

This architecture that is to be presented is based on a Mobile Multi-Agent System—everything in the system is abstracted through agents. The system is comprised of both static and mobile agents. Static agents provide resources and facilities to mobile agents. Mobile agents move between network domains taking advantage of these resources to fulfill their goals.

The overview of the architecture is illustrated in Table 4.1. This table categorizes agents as belonging to any of six static agent types based on their functionalities in the system:

Table 4.1 Static Agent Types of Multi-Agent Autonomic Architecture

Agent Types	Description
User Interface Agents	Present agent results to the user and solicit input from the user.
Middle Agents	Support the flow of information in systems. Assist in locating and connect the ultimate service provider with the ultimate service requester.
Task Agents	Coordinate, decompose, and delegate tasks received from the middle agent or from other task agents.
Resource Agents	Monitor and interact with data sources. Update external data stores.
Domain Agents	Supervise, manage and control the activities that occur within a domain.
Monitor Agents	Continuously monitor the state of each static agent within a given domain. Detect agent failure and initiate agent recovery procedures.

Apart from these static agent types mentioned in Table 4.1, a number of mobile agents will be launched and dispatched to the target host by any static agent. Mobile agents are equipped with a set of goals to enable static agents to coordinate and cooperate with each other to fulfill a user's goals.

In addition, both static agents and mobile agents are built on an agent platform, whose responsibilities range from creating, cloning, and dispatching to disposing of agents.

The architecture of the Mobile Multi-Agent Autonomic Systems is illustrated in Figure 4.1.

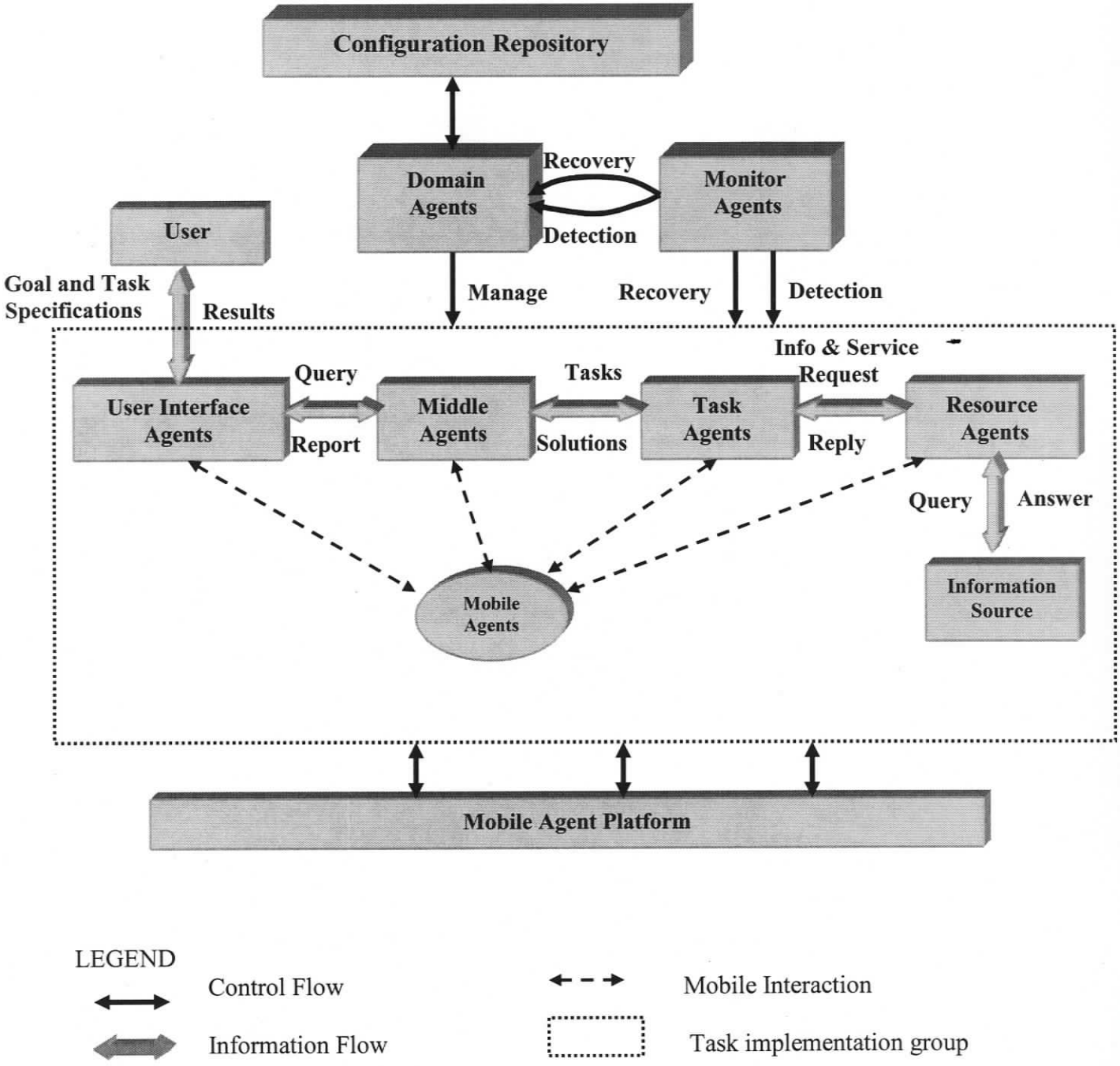


Figure 4.1 Architecture of Mobile Multi-Agent Autonomic Systems

The next eight sections take each component of the architecture and describe its general activities and purpose in the multi-agent infrastructure.

4.1.1 User Interface Agents

User interface agents interact with the user by receiving user specifications and delivering results. They are static agents that reside within a domain and provide a level of abstraction between the user and the details of the mobile agent framework. The user interface agent provides a graphical interface that links the user with other agents. It is the only agent that resides on the client computer that communicates with the user directly. User interface agents need to perform the following tasks:

- Collect relevant information from the user to initiate a task.
- Present relevant information including results and explanations.
- Organize and preprocess information returned from agents into a form that is suitable to the user.
- Ask the user for additional information during problem solving.
- Validate the user's input. Ask for user confirmation when necessary.
- Provide online help to the user.

Having the user interface presented through an associated user interface agent for a task hides the underlying information gathering and problem solving complexity. The user is free from having to know how to access and interact with a potentially large number of agents to fulfill a task.

4.1.2 Middle Agents

Middle agents provide the infrastructure for other agents. A typical instance of a middle agent is the Matchmaker [FSLH2005] [ZH2002], or Yellow Page agent. A middle agent is capable of performing the following tasks:

- Locate the appropriate service-providing agents based upon their published capability description, known as advertisements, when request agents submit a capability request to the middle agent.
- Modify service-providing agents' state of capability accordingly when they register into or leave a domain.
- Act as task management or task status tracking as processes are enacted, and their associated activities occur, there needs to be a means of tracking the state of the execution so that clients can determine when and whether they will complete.

Middle agents work as a bridge to match the request to a class of potentially suitable candidates, select the most qualified service provider from among those candidates, and assign a specific task to the agent to perform a particular function.

4.1.3 Task Agents

Task agents support decision making by formulating problem solving plans and carrying out these plans through querying and exchanging information with other agents.

Task agents have knowledge of the task domain, and other agents that are relevant to performing various parts of the task. A task agent performs most of the autonomous problem solving.

The main functions of task agents include:

- Receive user delegated task specifications from a middle agent or other task agents.
- Interpret the specifications and extracts problem solving goals.
- Formulate plans to satisfy these goals.
- Decompose formulated plans into discrete tasks.
- Coordinate with the appropriate task agents for plan execution, monitoring and results composition.

Task agents help users perform tasks, formulate problem-solving plans and carry out these plans. They encapsulate task-specific knowledge and use that knowledge as the criterion for requesting or performing services for other agents or users.

4.1.4 Resource Agents

Resource agents are static agents that exist within a domain to provide a level of abstraction between agents and the information resource to which they provide access. The purpose of a resource agent is to mediate access to a particular resource at a local level for an agent. The resource agent understands how to interrogate the resource, and understands the permission structures associated with it.

The main functions of the resource agents include:

- Be fully conversant in the protocols of the information resource to provide complete access. The information resource should be completely accessible by agents so that user intervention is not required.
- Provide a standard and generic set of services which all agents can utilize. For example, query, get, delete, add, and alter, and so on.
- Maintain the integrity and privacy of the resource.
- Update external data stores, such as databases.

Resource agents can provide intelligent access to a collection of information sources. They can be categorized as a single source if they only model one information source or multi-source if one resource agent represents multiple information sources.

4.1.5 Domain Agents

A domain agent is a static agent that supervises and coordinates the activities that occur within a domain. A domain is a logical boundary used to delimit nodes, agents and resources into manageable and distinct entities. The domain agent offers a number of services related to information resources and agents within a given domain. The main functions of the domain agents as follows:

- Provide authentication and validation checks on agents wishing to enter the domain. Agents that cannot be authenticated or fail validation checks are rejected.

- Provide a central registration area where static and mobile agents can register and advertise their resources and interests. Thus, a domain is also a meeting point which allows agents to gather and share information to resolve their goals.
- Mediate global access to information resources. Before an agent can access information resources it must obtain permission from the domain agent.

The domain agent is the initial point of contact within the domain. All message interactions between static and mobile agents are initially routed through the domain agent, which allows agents to communicate in an asynchronous fashion. However, once two agents have become aware of each other's presence via the domain agent, they can initiate direct and synchronous communication.

The domain agent is the key force within a domain and is ultimately responsible for ensuring that security is enforced within a domain, and facilitating agent communication.

4.1.6 Monitor Agents

The objective of a monitor agent is to continuously monitor the status of each agent, detect agent failures, and execute recovery. A monitor agent plays a vital role in the Multi-Agent System by making sure that all of the agents work as desired, and the system is implemented properly. Apart from the basic operations that a monitor possesses, the monitor agents are able to carry out recovery procedures to achieve one of the properties of the Autonomic Systems: self-healing. Therefore, a monitor agent needs to provide the following services:

- Continually keep track of the behavior of each agent within a given domain.
- Check whether an action timeout has occurred.
- Generate and dispatch a mobile agent to the target host to implement recovery the procedure.

Section 4.2 describes in more detail how a monitor agent achieves self-healing.

4.1.7 Mobile Agents

Mobile agents are the components within the architecture that can migrate between network nodes. The essential functions of mobile agents will be defined by the specific tasks that they are allocated. They have the following responsibilities within the architecture:

- Determine where to migrate to next by querying the local domain agent for a list of agents of which it is aware. The mobile agents can then use this information to contact each agent and determine which one offers a compatible set of information resources.
- Transmit the results of their findings and actions regularly to their parents on the host domain.
- Communicate with other static agents and other mobile agents to achieve their goals.
- Need be authenticated by the host domain of the mobile agent.

Mobile agents are the ultimate effectors of change within a multi-agent system, and as such, should be managed cautiously. The possibility for mobile agents to effect

changes or gain access to private information must be closely monitored by the relevant agent.

4.1.8 Mobile Agent Platform

The mobile agent platform supports the creation and deletion of agents and migration of mobile agents to interact with other agents located at various hosts. Mobile agent platforms have been developed by several institutions and organizations mentioned in chapter 2.

In this research, the IBM Aglets platform is chosen as our mobile agent platform. Aglets are one of the best-known mobile Java agent projects. It was designed to take advantage of the agent characteristics of Java [HCMKK2000] [GHMCKT2003]. At the same time, Aglets provide additional capabilities such as resource control, support for preservation and resumption of execution state [LO1998] [GHMCKT2003]. Aglets are popular for their ease of programming, functionality, robustness and good performance. For a comprehensive comparison and quantitative performance evaluation of the mobile agent platforms, refer to [MNNS2004] [SSMBS1999] [DS2000] [SDSL1999].

An Aglet is a Java object that has the ability to move (be dispatched) autonomously from one computer host to another. It can halt itself, ship itself to another computer on the network, and continue execution at the new computer. When the Aglet moves, it takes along its program code as well as the states of all the objects it is carrying.

Aglets are autonomous because once they are started they independently decide where they will go and what they will do [LO1998] [GHMCKT2003] [QS2004]. They can control their own lifecycle. They can receive requests from external sources, but each individual Aglet decides whether or not to comply with external requests. Also, Aglets can decide to perform actions, such as travel across a network to a new computer, independent of any external request.

An Aglet comprises three key abstractions—Aglet, Aglet Proxy and Context [LO1998] [GHMCKT2003] [QS2004]. The Aglet structure is illustrated in Figure 4.2.

- An aglet is a mobile Java object that visits aglet-enabled hosts in a computer network. It is autonomous, because it runs in its own thread of execution after arriving at host. It is reactive, because it can respond to incoming messages.
- Each Aglet Proxy is a representative of an Aglet. It serves as a shield that protects the Aglet from direct access to its public method. One Aglet sends messages to one another via the Aglet Proxy.
- A Context is an Aglet's container where Aglets execute. It is a static object that provides a means for maintaining and managing running Aglets in a uniform execution environment.

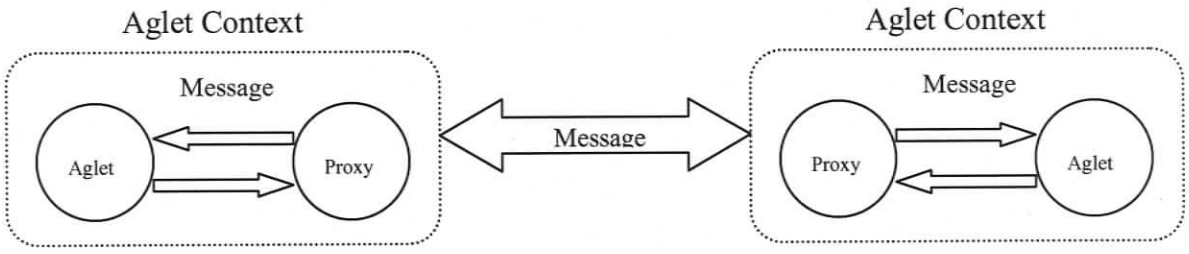


Figure 4.2 Structure of Aglet

An Aglet lifecycle involves the following fundamental operations (see Figure 4.3) [LO1998]:

- Creation: create an Aglet in a context, and its main thread begins executing.
- Cloning: clone an identical copy of the original Aglet in the same context, with current state but new identity.
- Dispatching: dispatch an Aglet from one context to another and the state goes with it.
- Retraction: a previously dispatched Aglet is pulled back from a remote host to the host on which it was created.
- Deactivation: an Aglet is put to sleep and its state is stored on a disk somewhere. Aglet execution is suspended.
- Activation: Aglet and its state are brought back to life from disk. An Aglet resumes the state it had before deactivation.
- Disposal: an Aglet dies and its state is lost forever.

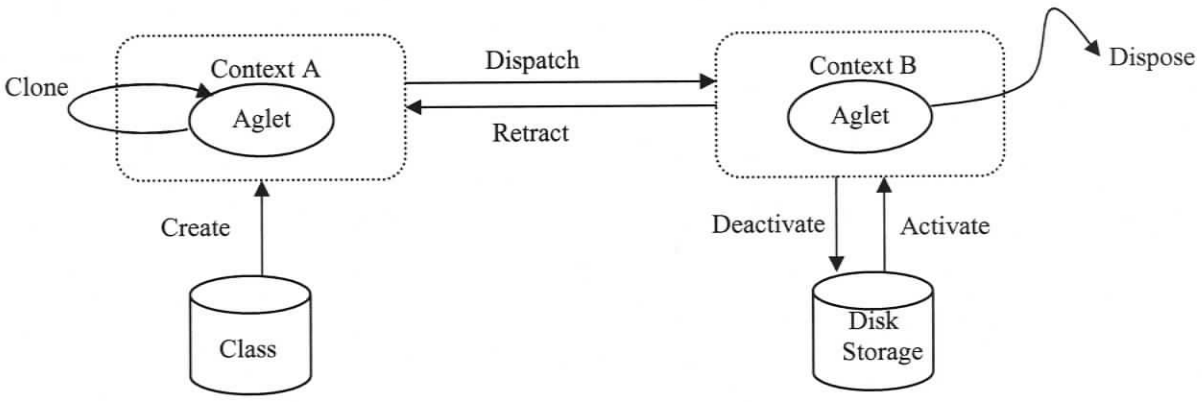


Figure 4.3 Lifecycle of Aglet

Before any of these events occur, the Aglet is notified of the upcoming event through a call to the appropriate callback method. For example, before an Aglet is created, the *OnCreation( )* method is invoked. We can override this method with one that initializes the state of the newly formed Aglet.

4.2 Self-healing

This Mobile Multi-Agent Autonomic Architecture achieves self-healing by incorporating mechanisms for self-monitoring and self-configuration at different levels of the system architecture. The event detection, correlation, and notification mechanisms are used as the basic building blocks for failure detection. Our design uses the notion of continuous periodic detection of each static agent and notification of a failure event until the failed components causing it are repaired.

The recovery mechanisms are designed to address the failures of individual static agents. The agents themselves may fail in unpredictable ways. They may incur partial failures or fail completely and stop communicating with other components.

Figure 4.1 illustrates how monitor agents of this system perform self-monitoring and recovery. They are primarily responsible for managing system-wide monitoring policies. The monitor agents achieve self-monitoring by subscribing to *AgentAlive* events from all static agents in the system. If no such event is received from an agent over a pre-defined number of consecutive timeout periods, it generates an *AgentFailure*. It keeps on generating such events until the agent is restarted and a heart-beat message is received.

If the monitor agent receives an *AgentFailure* message, it immediately creates and dispatches a Recovery Agent to the faulty agent host. Upon arriving at the target host, the Recovery Agent executes recovery action by creating a new agent to replace the failed one.

4.3 Summary

This chapter described the architecture of Mobile Multi-Agent Based Autonomic Systems. It consists of six types of static agents. They are User Interface Agents, Middle Agents, Task Agents, Resource Agents, Domain Agents and Monitor Agent. Mobile agents roam among these static agents and communicate and cooperate with other agents to properly implement the desired goals. The mobile agents can be created, disposed and migrated by a Mobile Agent Platform—Aglets, a mobile Java object. The architecture directly addresses the requirements of Autonomic Systems. It autonomously fulfills the user's goals without human intervention. Self-healing is achieved through the use of a heart beat monitor mechanism. The next chapter describes the design and implementation of an application, an electronic marketplace, based on the architecture described in this chapter.

Chapter 5 Design and Implementation of the Electronic Marketplace

In order to verify the effectiveness of the Mobile Multi-Agent Autonomic Architecture, an application—an electronic marketplace—is constructed. This chapter discusses design and implementation details of this application.

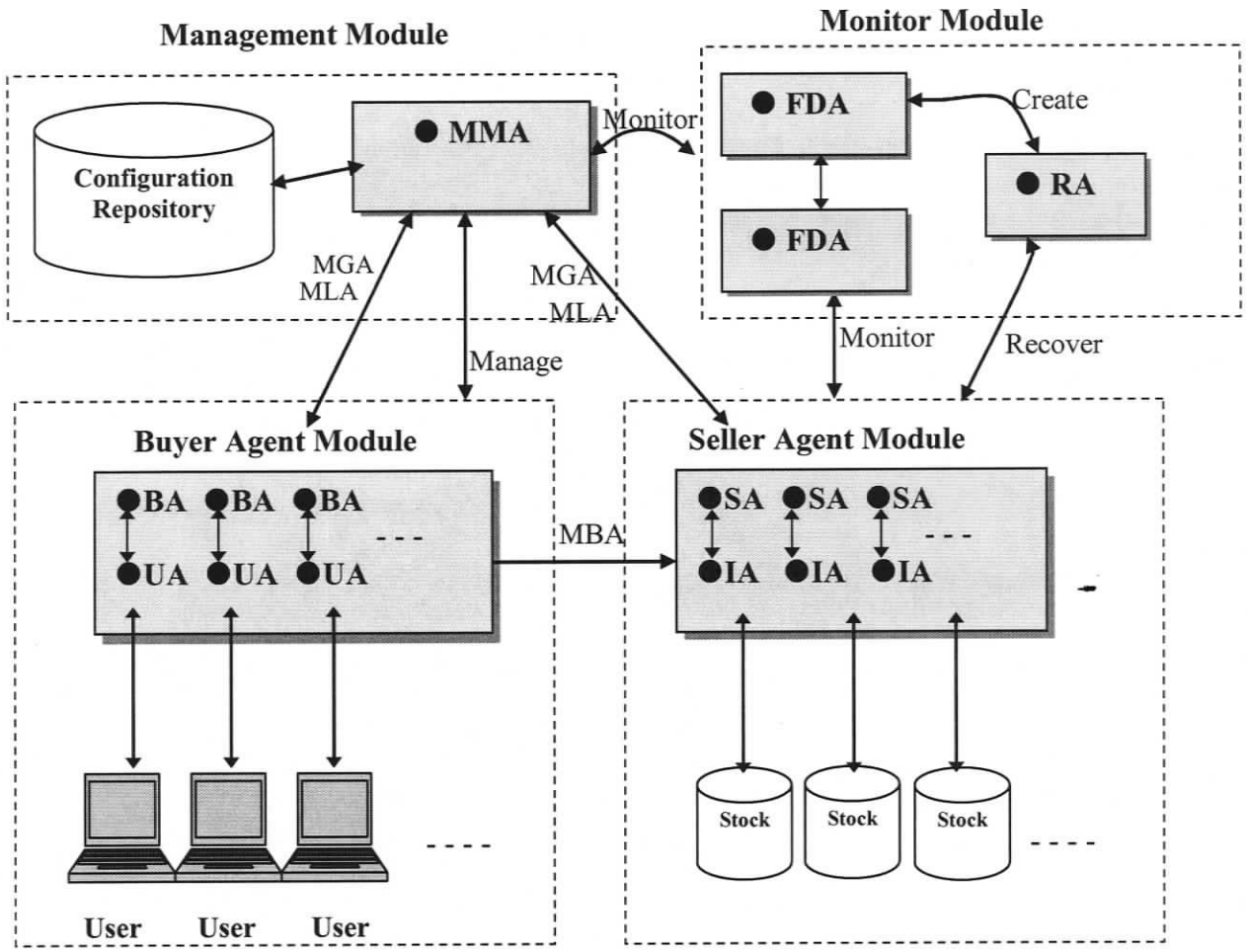
5.1 The Role of Agents in an Electronic Marketplace Application

Based on the proposed architecture depicted in the previous chapter, we modeled an agent environment—an electronic marketplace which is populated with economically motivated agents. The nature of this electronic marketplace allows the economic agents, which are classified as buyers and sellers, to enter or leave the market. Buyers and sellers are self-interested agents whose goal is to maximize their own profit. As shown in Table 5.1, our electronic marketplace consists of four main modules:

Table 5.1 List of Design Modules and Descriptions

Module Name	Description
Management Module	Manages the E-marketplace including all buyers and sellers in this market
Buyer Agent Module	Acts as a buyer to trade with sellers to obtain the best price
Seller Agent Module	Acts as a seller to sell products and realize profits
Monitor Agent Module	Continuously monitors the E-marketplace to detect if there is something wrong with any agent; recover it from failures.

Each module contains a number of static agents and/or mobile agents that through communication and movement, tie together the modules to achieve a user's goals (see Figure 5.1). The following sections describe these modules in detail.



LEGEND

BA – Buyer Agent
IA – Inventory Agent
RA – Recovery Agent
SA – Seller Agent
UA – User Agent

FDA – Failure Detection Agent
MBA – Mobile Buyer Agent
MGA – Mobile Register Agent
MLA – Mobile Logout Agent
MMA – Market Management Agent

Agent

Organizational relationship

Data Resource

Interaction

Figure 5.1 Major Modules of E-marketplace Application

5.1.1 Management Module

The Management Module has a Market Management Agent (MMA), whose responsibility is to manage its E-marketplace domain. An MMA has several abilities: (1) initiate an E-marketplace domain; (2) monitor Marketplace, Buyer Agents and Seller Agents; (3) manage Marketplace, Buyer Agents and Seller Agents; and (4) provide registration and authenticity capabilities.

In addition, there is a configuration repository to keep track of the status of all buyers and sellers in this E-marketplace. When a buyer or a seller enters the E-Marketplace, it will send a Mobile Registration Agent (MGA) to register with MMA, and when it logs out, it will dispatch a Mobile Logout Agent (MLA) to inform MMA of its exit. The MMA will add or delete an annotation in the repository when a buyer or seller agent enters or leaves.

5.1.2 Buyer Agent Module

The Buyer Agent Module contains a User Agent (UA), which provides a user interface that receives a user's goals and tasks. Each user is served by a Buyer Agent (BA). A BA will generate a Mobile Buyer Agent (MBA) to visit every seller to make bargains. After visiting all sellers and getting the best price, the MBA will create a Mobile Report Agent (MRA) to report the final purchase result to the seller. Moreover, there are two more mobile agents, a Mobile Registration Agent (MGA) and a Mobile Logout Agent (MLA), which are responsible for registering with the MMA and logging out of the E-marketplace for the BA.

User Agent (UA)

The User Agent is responsible for user system interaction. After the UA is initiated, it creates a graphic user interface and waits for the user to input needed products and the price range they are willing to pay. When a user has requested some goods, the UA sends a purchase query to the Buyer Agent to process the requirements. After the transaction is complete, the UA receives the purchase results and displays it on the user interface.

Buyer Agent (BA)

After the BA receives a query from the UA, it creates a purchase order and obtains a confirmation from the UA. The BA then creates a Mobile Buyer Agent and dispatches it to visit all sellers who provide the requested product. After the Mobile Buyer Agent comes back, the BA receives the purchase results and sends it to the UA.

Mobile Buyer Agent (MBA)

The Mobile Buyer Agent is created by a BA and then explores the network to negotiate with each Seller Agent. After it has visited all destinations, the MBA returns home and reports the results to the BA. The MBA disposes itself when it has accomplished its tasks.

Mobile Report Agent (MRA)

The Mobile Report Agent is generated by an MBA. When the MBA returns home, it starts to process the purchase results, and creates an MRA to send the final purchase results to the successful sellers. Each successful seller, who upon receiving a confirmation message from the MRA, will update its inventory, but unsuccessful sellers

are not notified. After the MRA has visited all the relevant destinations, it returns home and disposes of itself.

Mobile Registration Agent (MGA)

A BA creates a Mobile Register Agent when it enters the market. An MGA will go to an MMA site to register with the MMA. After successful registration, the MGA returns home and reports to its BA which will have the authority to check out all sellers in the market and trade with all available sellers.

Mobile Logout Agent (MLA)

The BA generates a Mobile Logout Agent when it is ready to exit the market. An MLA will go to an MMA site and inform the MMA that its parent BA exits. After passing an identification check, the MMA will send a confirmation to the MLA. The MLA then returns home and the BA will officially exit the market.

5.1.3 Seller Agent Module

Each company, which intends to join this market, should build a Seller Module. There are two static agents in a Seller Module.

Seller Agent (SA)

The Seller Agent is in charge of selling tasks. When an MBA arrives, the SA accepts the MBA request. The SA will initially check with an Inventory Agent to see if it has the needed products in stock. If the seller has goods available for sale, the SA negotiates with the MBA; otherwise, the SA will send a message to refuse the request. If a transaction is successful, it asks an Inventory Agent to update its inventory.

Inventory Agent (IA)

Each SA has its own Inventory Agent to keep track of the SA stock. The IA is used for receiving queries and updating messages from an SA. It sends query results back by creating an XML message and updates its inventory using XML Document Object Model (DOM) technology.

Apart from the above static agents, a Seller Agent Module contains an MGA and an MLA, which has the same function as the ones in the Buyer Agent Module to ensure the SA enters the market with authority and exits the market properly.

5.1.4 Monitor Agent Module

The Monitor Agent Module performs a set of functions which includes self-monitoring, detection and recovery. It periodically checks each static agent using a Failure Detection Agent (FDA) to send messages to every host in order to get the heartbeat from each agent, and generates a Recovery Agent to recover an agent if the FDA finds something wrong with that agent.

Failure Detection Agent (FDA)

This module includes a pair of Failure Detection Agents, executing on different hosts that monitor each other to make sure at least one FDA is continually monitoring the state of the system. If one of them fails, the other one will detect it and it will immediately create a new FDA to replace the dead one. Both FDA send messages to all

static agents in this market, for instance the MMA, SA and BA, to detect a heartbeat from them. If a response is received within a pre-defined number of timeout periods, the agent is not dead; otherwise, a Recovery Agent will be created.

Recovery Agent (RA)

The FDA creates a Recovery Agent if the FDA detects a failure within an agent, and dispatches the RA to the target host. Upon arriving at the host, the RA starts to implement a recovery procedure by creating a new agent to replace the dead one.

5.2 Agent Communication Language

In order to cooperate effectively in this system, agents are required to communicate with one another. When agents need to communicate, they must individually "understand" messages written in the Agent Communication Language (ACL).

5.2.1 KQML Performatives

As described in chapter 2, KQML is both a message format and a message-handling protocol to support run-time knowledge shared among agents. KQML focuses on an extensible set of performatives, which defines the permissible operations that agents may attempt on each other's knowledge and goal stores [FLM1997]. The main advantage of KQML is its ability to support a wide range of agent architecture with its extensible set of performatives [CD2002]. The performatives comprise a substrate on which to develop higher-level models of inter-agent interaction.

The following performatives are used for information and knowledge exchange in the electronic marketplace application.

Table 5.2 KQML Performatives in the E-marketplace Application

Performative Name	Description
accept-proposal	The action of accepting a previously submitted proposal to perform an action.
cancel	The action of canceling some previously requested action which has temporal extent.
cfp	The action of calling for proposals to perform a given action.
confirm	The sender informs the receiver that a given proposition is true.
failure	The action of telling another agent that an action was attempted but the attempt failed.
inform	The sender informs the receiver of a proposition.
not-understand	The sender tells the receiver that the sender did not understand the message that the receiver has just sent.
propose	The action of submitting a proposal to perform a certain action.
query-if	The action of asking another agent whether or not a given proposition is true.
refuse	The action of refusing to perform a given action, and explaining the reason for the refusal.
reject-proposal	The action of rejecting a proposal during a negotiation.
request	The sender requests the receiver to perform some action.

5.2.2 KQML Performatives Parameters

A KQML message consists of a performative whose associated parameters that include the real content of the message, and a set of optional parameters that describe the content in a manner that is independent of the syntax of the content language.

In this research, KQML performatives adopt the following parameters to exchange information among agents:

Table 5.3 KQML Performative Parameters in the E-marketplace Application

Parameters	Description
:sender	Identification of the sender's message.
:receiver	Identification of the intended recipient's message.
:language	Encoding schema of the content of the action.
:protocol	Identifier denotes the protocol that the sending agent is employing. The protocol serves to give additional context for the interpretation of the message.
:conversation-id	Expression is used to identify an ongoing sequence of communicative acts that together form a conversation.
:content	Content of the message; equivalently denotes the object of the action.

5.3 Agent Interaction Protocols

Social interactions, such as cooperation, coordination and negotiation, are a fundamental feature of multi-agent systems. All interactions between agents in this research are based on the exchange of messages according to a specific communication

protocol. Besides the Agent Communication Language (ACL), which is used to specify the individual messages that can be exchanged, another important issue is the Interaction Protocols (IPs). IPs specify a well-defined sequence of messages (each message refereeing to a speech act).

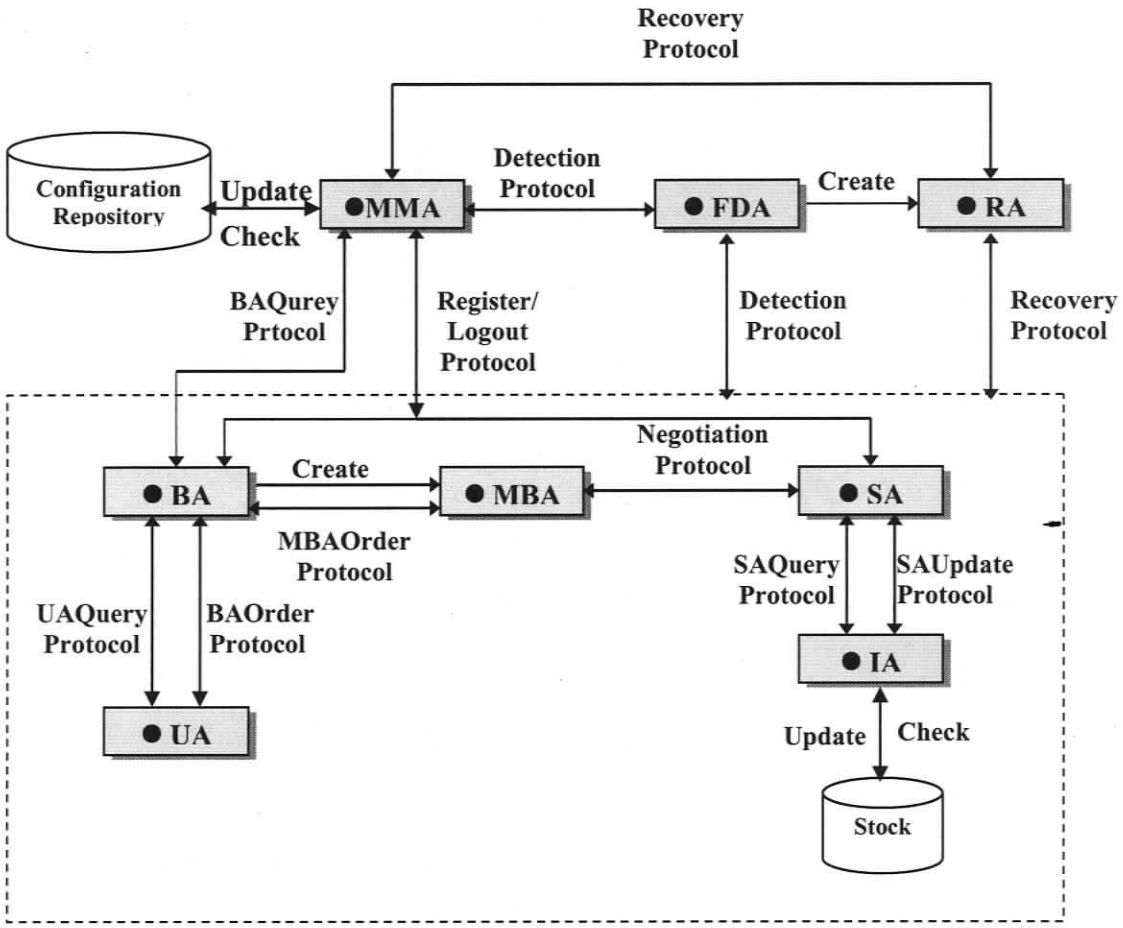
5.3.1 Protocol Design

The design of MAS involves the design of a communication protocol that will allow the individual agent to transmit and receive information from other agents. The design of the multi-agent communication protocol for this research is simplified in two ways:

1. Only certain agents are capable of communicating with each other. Using this structured design, the possible interactions between the agents are known and at each interaction, the type of information being sent will always be the same.
2. Although the communication protocol is designed for bi-directional communication, the information being sent from one agent to another is unidirectional. Information sent between agents can only move forward, with the acknowledgement being the only information sent back to the transmitting agent. Thus, the protocol is bi-directional, but information flow is unidirectional.

5.3.2 Interaction Protocols

Every interaction between two agents follows a defined protocol. Figure 5.2 represents all protocols in our E-marketplace application.



LEGEND

BA – Buyer Agent
IA – Inventory Agent
RA – Recovery Agent
SA – Seller Agent

FDA – Failure Detection Agent
MBA – Mobile Buyer Agent
MMA – Market Management Agent

Agent

Interaction

Task implementation group

Data Resource

Figure 5.2 Interaction Protocols in the E-marketplace Application

The following is a detailed description of each protocol.

- Registering/Logout Protocol—a BA or an SA initiates this protocol when they want to enter or exit the market by generating a mobile agent (MRA or MLA) and dispatching it to an MMA. Once the MMA receives the information and updates the configuration repository, it sends an acknowledgement to the mobile agent. After the mobile agent returns home, it will report to the BA or the SA and dispose of itself.
- Detection Protocol—an FDA uses this protocol to periodically send messages to the MMA, and all SAs and BAs to detect the state of each agent.
- Recovery Protocol—if an FDA does not receive any response from an MA, an SA or a BA within a predefined timeout period, it will create an RA to execute recovery using this protocol.
- UAQuery Protocol—after users indicate products and price range they are willing to pay, a UA sends a query to a BA. The BA then generates a purchase order and replies to the UA with confirmation.
- BAQuery Protocol—upon obtaining confirmation from a UA, a BA creates a mobile agent and sends it to the MMA site. After the mobile agent returns home with all the available sellers' information, the BA will choose sellers and dispatch an MBA to trade with each seller.
- BAOrder Protocol—after a BA gets the purchase results from an MBA, it sends final results to the UA and the UA reports to users by displaying purchase results on the user interface.

- MBAOrder Protocol—this protocol is designed for an MBA to send purchase results to a BA after it has visited all available SAs. The MBA sends a "request" message to the BA. The BA replies with "confirm" or "failure" when it finishes processing.
- Negotiation Protocol—as soon as an MBA arrives at a seller's site, it starts to negotiate with the seller. This includes two specific protocols, *MBANegotiation* Protocol and *SANegotiation* Protocol, to implement one-to-one negotiation (see 5.4.2 for details).
- SAQuery Protocol—this protocol is used to query inventory between an SA and an IA. When the SA wants to query the inventory, it sends a message to the IA. The IA will check its stock and reply to the SA with inventory information.
- SAUpdate Protocol—when an SA has sold some products, it will update its stock. This protocol is designed to update inventory between an SA and an IA. When the SA wants to update its inventory, it sends a "request" message to the IA. The IA replies with "confirm" or "failure" based on the update action.

5.4 Autonomous E-shopping

This section introduces the autonomous transaction processing in the E-Marketplace application. The steps of shopping without human intervention for buyers are described below.

5.4.1 The Steps of Shopping

- When a User Agent receives the "Go Shopping" message from a user action (i.e., (1) in Figure 5.3), it sends a purchase query to a Buyer Agent (i.e., (2) in Figure 5.3). The Buyer Agent immediately creates a purchase order, which is an XML file, and sends it to the User Agent to get confirmation.
- Upon obtaining confirmation from the User Agent, the Buyer Agent creates an MBA and dispatches it to the Market Management Agent (i.e., (3) in Figure 5.3). After the MBA passes the identity verification, the Market Management Agent will check the configuration repository based on the products that the buyer needs, and get the relevant information about all sellers and give the MBA the sellers' address (i.e., (4) in Figure 5.3).
- After the Buyer Agent receives from the MBA all the relevant sellers' addresses, it can choose the sellers with which to transact. Then the Buyer Agent sends the MBA to visit each seller (i.e., (5) in Figure 5.3).
- Upon arriving at a seller's site, the MBA starts to negotiate with the Seller Agent. Firstly, the Seller checks his own inventory through an Inventory Agent, if it has the needed goods, the Seller will accept the proposal from the MBA and begin to bargain with the MBA by using three negotiation strategies (see section 5.4.3 for details); otherwise, it will send a "refuse" message to the MBA (i.e., (6) in Figure 5.3). Section 5.4.2 introduces the detailed negotiation process.

- As soon as the MBA visits all Sellers, it returns home with the purchase results (i.e., (7) in Figure 5.3). After the MBA processes the results, compares all prices, and selects the best price, it will generate a Mobile Report Agent (i.e., (8) in Figure 5.3) and dispatch it to the relevant Sellers in order to tell them the final purchase result. The successful Sellers will update their inventories accordingly.
- After the Mobile Report Agent finishes its task, it will go back to its parent Mobile Buyer Agent, report to the MBA and dispose itself (i.e., (9) in Figure 5.3).
- The MBA reports the purchase results to the Buyer Agent and disposes itself. Finally (i.e., (10) in Figure 5.3), the Buyer Agent sends the purchase results to the User Agent (i.e., (11) in Figure 5.3) and the User Agent displays information for the user (i.e., (12) in Figure 5.3).

The inventory file and purchase order is in XML format. The Simple API for XML (SAX) is used for checking XML content while XML Document Object Model (DOM) is used for updating and modifying XML content.

A graphical user interface allows the user to input purchase order information and displays the purchase results. A seller agent interface is used to show inventory and selling results.

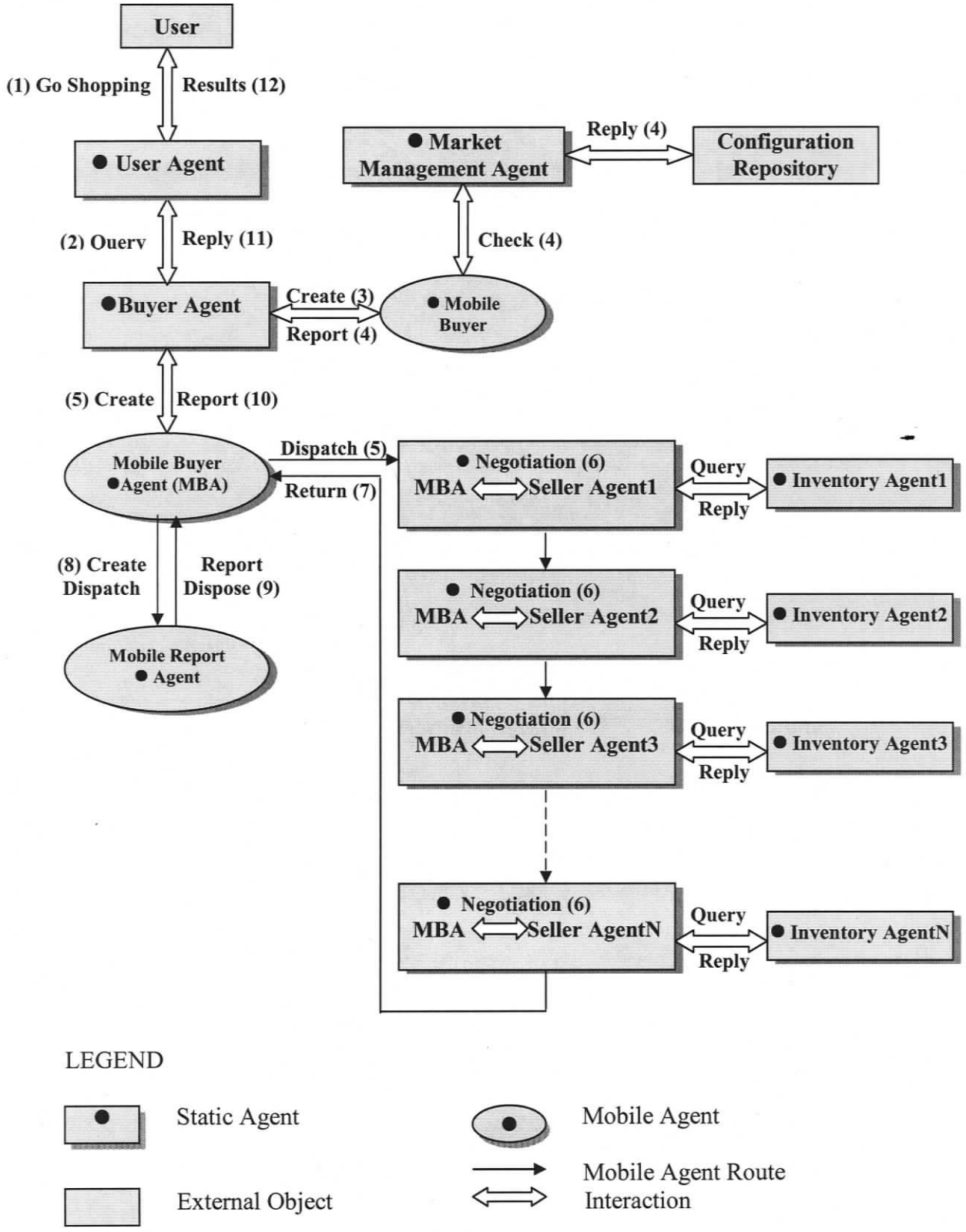


Figure 5.3 Autonomous Transaction Processing for E-shopping

5.4.2 One to One Negotiation

After an MBA arrives at a Seller site, it starts to negotiate one-to-one with the Seller. The purpose of the one-to-one negotiation mechanism is to allow the buyer to have more chances to get a better price by bargaining with all available sellers. One-to-one negotiation processes are described below.

- Upon arriving at a Seller site, the MBA starts negotiating by sending a "cfp" message.
- The Seller receives the "cfp" message and checks its inventory to see if there are enough products available in stock. The Seller replies to the MBA with sell check result. The reply message can be "accept-proposal" when the price is acceptable; it can be "refuse" when there is not enough stock, or it can be "not-understand" when there is an XML validation error; it can be "propose" when price negotiation is required; or finally it can be "reject-proposal" when the price is unacceptable.
- The MBA gets the order price as a reply and uses its strategy to determine if it can continue to negotiate with the seller. If the strategy permits more negotiation, the MBA creates a new proposal price using this strategy.
- The seller uses its strategy to compare the MBA order price and generates a new proposal price.
- This negotiation process continues until the buying price is greater than or equal to the selling price and then the proposal is accepted; otherwise it is rejected.

- If the MBA accepts a proposal, it will record the current agreement order along with the seller's local URL. After the MBA has traveled to all the destinations, it returns home.
- When the MBA returns home, it calculates the best price and dispatches a Mobile Report Agent to send a "confirm" message to the successful seller. For unsuccessful sellers, it sends a "cancel" message.
- Finally, the buyer gets the purchase results and the successful seller updates its inventory upon receiving the "confirm" message.

The sequence diagram of one-to-one negotiation is shown in Figure 5.4.

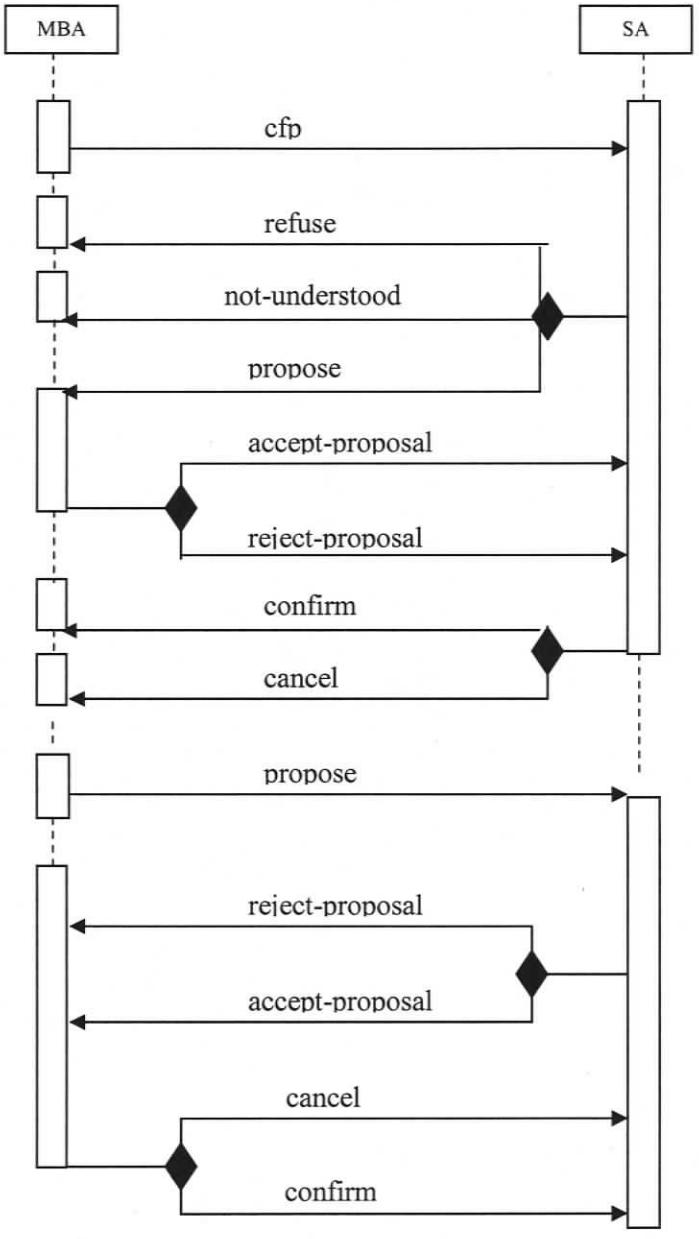


Figure 5.4 MBA and Seller One-to-one Negotiation

5.4.3 Negotiation Strategy

In this project, we implemented three different kinds of negotiation strategies when a mobile buyer agent bargains with a seller agent.

Each strategy implements two base methods, one is the *hasNextPrice( )* method which returns true if the price is between the open price and the reserve price, the other is the *getPriceValue( )* method which returns the next strategy price value.

The three strategies are Linear Strategy, Concede Strategy and Holdout Strategy. They use the following formula to calculate next price:

$$Bt = Bo + (Br - Bo) * (t/T)^k$$

$$St = So - (So - Sr) * (t/T)^k$$

where *Bo* is buy open price, *Br* is buy reserve price; *So* is sell open price, *Sr* is sell reserve price, *t* is a time counter, and *T* is default time without timeout.

When k = 1; the above formula represents Linear Strategy;

When k < 1; the above formula represents Concede Strategy;

When k >1; the above formula represents Holdout Strategy.

5.5 Self-healing Electronic Marketplace

This E-marketplace application achieves self-healing by incorporating mechanisms for self-monitoring and self-recovery at different levels of the system architecture.

A Failure Detection Agent (FDA) is primarily responsible for managing system-wide monitoring and detecting policies. It is replicated for fault tolerance. They continually monitor each static agent in this system.

Each static agent is equipped with an *AgentAliveDetector*, whose main functions fall into two broad categories: execution of local detector functions and event notification. It periodically checks the internal state of the static agent and generates *AgentAlive* heart beat events to indicate the health of the agent.

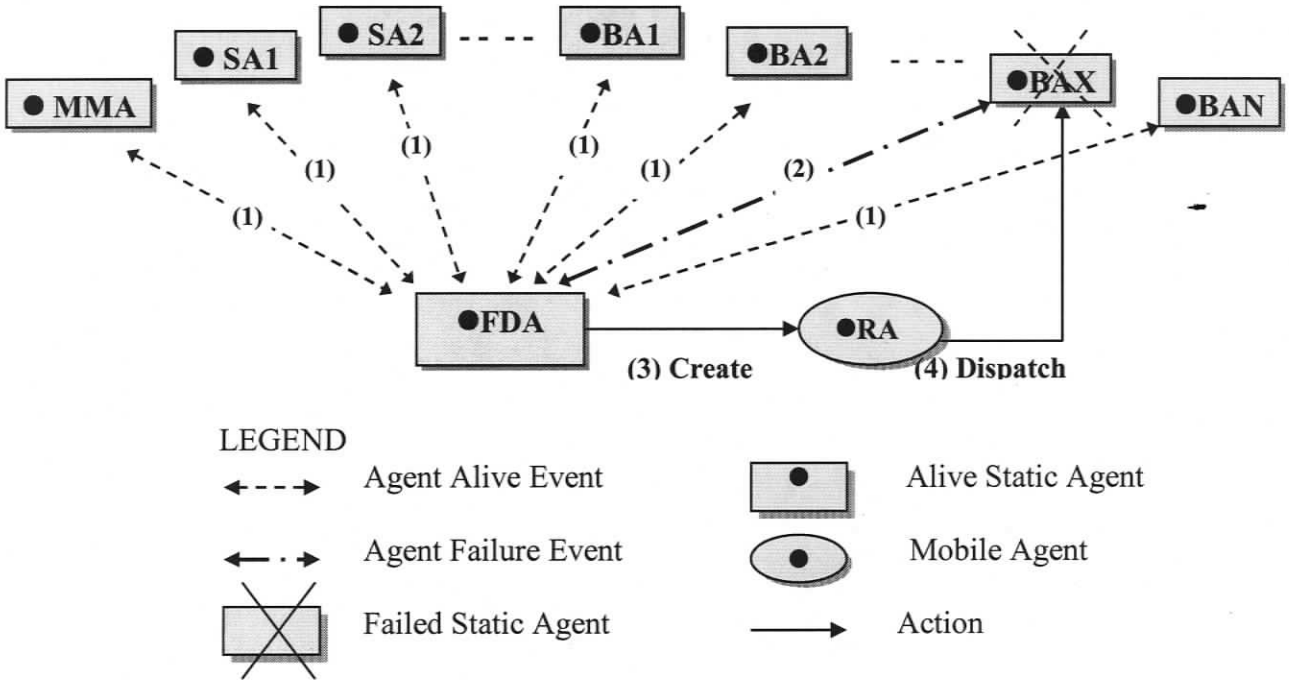


Figure 5.5 Recovery of an Agent in the E-marketplace Application

Each Failure Detection Agent subscribes to *AgentAlive* events from all static agents in the system (i.e., Step 1 of Figure 5.5). If no such event is received from an agent over a pre-defined number of consecutive timeout periods, it generates an *AgentFailure* event (i.e., Step 2 of Figure 5.5). It keeps on generating such events until the agent is restarted and a heart-beat message is received.

Once the Failure Detection Agent receives an *AgentFailure* event, it generates a Recovery Agent (i.e., Step 3 of Figure 5.5) and dispatches it to the target host (i.e., Step 4 of Figure 5.5). A Recovery Agent implements a recovery procedure at the failed agent site.

5.6 Summary

This chapter described the design and implementation of a Mobile Multi-Agent Autonomic Architecture application—an Electronic Marketplace. Based on the requirements and characteristics of a marketplace, the model primarily consists of four modules. Each module has its own objectives and functions. Mobile agents make a significant contribution to enable these modules to work together, coupled with the agent communication language and interaction protocols, bring multiple agents together to achieve the desired goals. This electronic marketplace application, in addition to performing E-shopping tasks that would normally be carried out by the user, continually monitors for any failure of any component and autonomously executes recovery, which enables the marketplace to be more reliable and robust.

Chapter 6 Evaluation

The ultimate goal of this research is to develop Autonomic Systems based on Mobile Multi-Agent Systems technology in order to make the process of building Autonomic Systems more systematic and less error-prone. In terms of mobile-agent based Multi-Agent Systems, an autonomic electronic marketplace application was implemented, which exhibits some basic characteristics of Autonomic Systems: autonomous, self-healing, self-managing, self-monitoring, complexity hiding and reduction of human intervention.

6.1 Comparison

This research proposes an architecture, which implements Autonomic Systems by taking advantage of distinguished characteristics of mobile multi-agent systems. The application to an electronic marketplace provides autonomy to current electronic marketplaces. The functional agents in this marketplace provide various services and a

defined transaction mechanism for mobile agents to execute transaction in a marketplace effectively. Typical characteristics of mobile agents include their ability to migrate at will, autonomy in their actions, a peer-to-peer personality and network independence from their original location. There are several advantages that emerge from the proposed architecture:

- Heterogeneity. Taking advantage of one of the most distinguishing features of mobile agents—heterogeneity in platform, this Mobile Multi-Agent system is generic enough to suit a wide rang of applications. Because a mobile agent is transport-layer independent, platform heterogeneity can be achieved by transmitting agents by byte code and interpreting the byte code at their destinations. Also, byte code is less bulky than source code so that it can be interpreted faster and can be authenticated easier than either machine code or source code.
- Robustness. The mechanism for self-recovery in the electronic marketplace is a mobile agent based system for monitoring multi-agent systems. It uses the monitoring system's inherent capabilities to detect its own component failures and notify a failure event until the failed components are repaired.
- Scalability. Each functional agent is responsible for different tasks, and cooperates to provide services in the marketplace. It is flexible so that we can increase or decrease the number and the kind of functional agents in the marketplace if necessary without significantly affecting system performance.

- Flexibility. All agents in the electronic marketplace can interact in new configurations "on demand," depending on the information requirements of a particular decision making task.

6.2 Development Experience

Agent orientation can contribute to Autonomic Systems in many ways. We have outlined a notion of agent from the viewpoint of characteristics of agents and mobile agents.

From the developer point of view, Multi-Agent Systems make it much easier to build Autonomic Systems. Agents provide a new way of managing complexity because they provide a new way of describing a complex system or process. Using agents, it is easy to define a system in terms of agent-mediated processes.

In an agent-based system design, the focus is placed on the behavior of each of these agents and communication between agents. The problem is made much easier because the level of abstraction is much higher and the programming problem becomes one of specifying agent behavior.

An agent-based approach to Autonomic Systems is useful and attractive because the various agents used inherently know how to do many things. For example, agents know how to communicate with other agents. The system developer does not need to design message-transferring protocols and message formats. The agent provides this capability as part of the basic agent mechanism. Agents have the inherent capability to build models

of their environment, monitor the state of that environment, reason and make decisions based on that state. All the software developer needs to do is simply specify what the agents do in any given situation.

Certainly, a multi-agent approach to Autonomic Systems has some drawbacks. Each machine must install a mobile agent server to execute, create and migrate mobile agents. For this research, every test machine needs to install the Aglets platform. In addition, the installation of agent server is not very easy: firstly, environment variables need to be set on your computer, and some parameters of the server need to be changed to make it more compatible with your machine. The security permissions given to the incoming agents must also be configured and the proper hooks necessary to allow the communication between the agents and the application must also be setup.

6.3 Good Practices and Lessons Learned

- Taking advantage of Aglets, Java objects that can move from one host on the network to another, supports dynamic and powerful communication that enables agents to communicate with unknown agents as well as well-known agents.
- KQML (Knowledge Query and Manipulation Language) was chosen as the communication language between the different types of agents. XML (Extensible Markup Language) plays an important role in exchange information. These languages make interactions among agents more effectively and easily.

- Mobile agents are naturally heterogeneous. They can travel from one host to another without considering the difference from both hardware and software perspective. Mobile agents provide optimal conditions for seamless system integration.
- This project shows that how agents assist people and act on their behalf, and some agents' properties, such as autonomy, communication and goal-driven behavior, could be used in other applications.

6.4 Summary

Compared to traditional approaches, mobile multi-agent technology is highly suitable for building Autonomic Systems. From the viewpoint of features that agents have, autonomy, proactivity, and goal-directed interactivity with their environment make it critically important to Autonomic Systems, coupled with mobile agents' properties; this approach is capable to improve system performance considerably. From the viewpoint of a developer, it reduces the design and development effort and enables developers to focus on a high-level system design. However, the drawback of the mobile agent system is that the installation of mobile agent platform is quite complicated.

Chapter 7 Conclusions

7.1 Research Summary

The increasing heterogeneity, dynamism and interconnectivity in software applications, services and networks lead to complex, unmanageable and unreliable systems. Coping with such a complexity necessitates to investigate a new paradigm namely Autonomic Systems. The traditional approaches to computer systems, which are often centralized and hierarchical, fail to adequately meet the challenges of building Autonomic Systems.

This thesis presented a new approach, a decentralized architecture for Autonomic Systems based on the notion of a mobile multi-agent system. The architecture is composed of six types of static agents and mobile agents, which migrate among these static agents. Taking advantage of the benefits of agents and mobile agents, the

architecture realizes a number of desired Autonomic Systems behaviors including autonomy, hiding complexity and self-healing.

We then presented a realistic prototype implementation—an electronic marketplace application based on the proposed architecture. The E-marketplace application consists of four main modules: management module, monitoring module, buyer agent module and seller agent module. Each module, which is composed of multiple static agents, plays different role in the E-marketplace to fulfill user's goals. Furthermore, employing agent communication language KQML, interaction protocols and negotiation strategies in terms of mobile agents' migration enables the static agents to cooperate with each other well in order to achieve autonomous E-shopping and self-healing within the E-marketplace application.

By comparing our system with current electronic marketplaces, we found that several critical qualities, such as efficiency, flexibility, reliability, scalability, and so on, of the system are greatly improved. Contrasting our approach with the traditional approach to build Autonomic Systems, the Mobile Multi-Agent Systems approach has definite advantages over traditional approaches from our development experience.

In general, our experience of developing mobile multi-agent systems to build Autonomic Systems is largely positive.

7.2 Contributions

We cannot claim that we have solved the problem of building Autonomic Systems based on Mobile Multi-Agent System technology completely. However, we believe that

our approach is a step in the right direction. The following list shows our main contributions to the research in this specific field.

- Investigate and analyze the benefits and drawbacks of utilizing agents to develop Autonomic Systems.
- Propose a general solution — the mobile multi-agent architecture, a six-type of static agent architecture containing a set of technical methods and practices.
- Design and implement a prototype — an electronic marketplace to validate the effectiveness of the mobile multi-agent architecture to achieve a set of desired autonomic behaviors.
- Collect some good practices and lessons we found in the process of our research and development.

7.3 Future Work

This research does not address all aspects of using Mobile Multi-Agent Systems for building Autonomic Systems. Several interesting but unexplored areas of research are outlined below.

7.3.1 Improving Self-optimization

In our implementation, the buying agents in our E-marketplace application bargain with the selling agents using three negotiation strategies: Concede Strategy, Linear Strategy and Holdout Strategy. However, this set of strategies is too fixed and lacks

flexibility. We would like to take into account the fact that multiple selling agents may offer the same goods with different qualities, and that selling agents learn to adjust price and alter the quality of their goods in order to maximize profit.

Toward this goal, we could make our marketplace more realistic and more interesting. We need to take the following issues into account:

- The quality of goods offered by different sellers may not be the same;
- A seller may alter the quality (in addition to the price) of its goods;
- Each buyer has some way to evaluate the goods it purchases, based on the price and the quality of the goods received.

7.3.2 Trust of Agents

The buying agents learn to avoid dishonest sellers and increase the satisfaction of their users by modeling the reputation of selling agents and focusing their business on those agents with whom they have established a certain degree of trust.

Establishing trust is one of the key factors when applying mobile agent technology to improve self-protection. We can protect mobile agents from the following categories: (1) An agent visits an untrusted server that may extract private information from or tamper with the agent. (2) An agent interacts with one of the agents in an attempt to extract private information from it and interrupts its execution. (3) Unauthorized third party threatens an agent. We would like to investigate existing security technology to solve most of these problems.

A comprehensive authentication, validation and access permission system is vital to ensure that malicious agents cannot do any harm to a system and that legitimate mobile agents can fulfill their goals.

7.3.3 Interface Agents

We intend to expand user interface agents to include a wider range of functions. They have the potential to be developed in a number of ways. Essentially, they could be implemented as an interface agent which sits along side the user and learns from their actions; in this manner, the user interface agent can take the initiative and autonomously launch mobile agents of its own to perform functions that it thinks will be of use to the user, dependent upon the actions that the user is performing or has performed or dependent upon changes in the user's environment.

7.3.4 Other Issues

Apart from the above questions, there are still some many more issues that can be explored and investigated.

- We aim to further our investigation in the area of multi-agent communication in order to develop a better understanding of the interaction between communities of agents. Research can also focus on improving the interface agents to learn from the interactions with the users and other agents. Specifically, certainly the knowledge-based approach would benefit from this research and assist in making Autonomic Systems more efficient.

- Currently, our E-marketplace application provides one way to let buyers and sellers transact using the one-to-one negotiation protocol. We would like to provide more transaction negotiation strategies in the future, such as shop directly and auction, to promote transaction activities.
- Find a better way of matching buyers and sellers in terms of a new function agent: a Broker Agent. An important function of the market would help users to find suitable ways to match buyers and sellers for example by using a dependency graph.

Bibliography

[Aglets1996] IBM Japan Research Group, Aglets workbench, http://aglets.sourceforge.net/

[Austin1975] J. L. Austin. *How to Do Things with Words (Second Edition)*, Harvard University Press, 188 pages, ISBN 0-674-41152-8, 1975.

[BBCM1998] M. Breugst, I. Busse, S. Covaci and T. Magedanz, "Grasshopper: A mobile Agent Platform for IN Based Service Environment," *Proceedings of 7th IEEE Intelligent Network Workshop,* pp. 279–290, Bordeaux, France, May 1998.

[BFL1996] Matt Blaze, Joan Feigenbaum and Jack Lacy, "Decentralized Trust Management," *Proceedings of the 1996 IEEE Conference on Privacy and Security,* pp. 164–173, Oakland, May 1996.

[BNS2004] M. Bema-Koes, I. Nourbakhsh and K.Sycara, "Communication Efficiency in Multi-Agent Systems," *Proceedings of the 2004 IEEE International Conference on Robotics & Automation*, pp. 2129–2134, New Orleans, USA, April 2004.

[CD2002] B. Chaib-Draa and F. Dignum, "Trends in Agent Communication Language," *World Congress on Computational Intelligence*, Vol. 2, No. 5, pp. 174–228, Hawaii, USA, May 2002.

[CKSW2004] D. M. Chess, V. Kumar, A. Segal and I. Whalley, "Work in Progress: Availability-Aware Self-Configuration in Autonomic Systems," *Proceedings of 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2004),* pp. 257–258, Davis, CA, USA, November 2004.

[CSWW2004] D. Chess, A. Segal, I. Whalley, and S. White, "Unity: Experiences with a Prototype Autonomic Computing System," *Proceedings of the First International Conference on Autonomic Computing (ICAC 2004),* pp. 140–147, New York, USA, May 2004.

[DS2000] M. Dikaiakos and G. Samaras, "A Performance Analysis Framework for Mobile-agent Systems," *Proceedings of the First Annual Workshop on Infrastructure for Scalable Multi-agent Systems, The Fourth International Conference on Autonomous Agents 2000,* Vol. 1887, pp. 180–187, Springer, 2000.

[FLM1997] T.Finin, Y. Labrou, and J. Mayfield, "KQML as an Agent Communication Language," *Software Agents*, J. M. Bradshaw, Ed., AAAI Press/The MIT Press, pp. 291–316, 1997.

[FSLH2005] E. Farazmand, A. Shokoufi, C. Lucas and F. Habibipour, "Using a Learning Mechanism in Matchmaking Process of Agents' Negotiation in Multi-Agent Systems," *Proceedings of the 19th International Conference on Advanced Information networking and Applications (AINA'05),* Vol. 1, pp. 59–64, Taipei, China, March 2005.

[GC2003] A. Ganek and T. Corbi, "The Dawning of the Autonomic Computing Era," *IBM Systems Journal*, Vol. 42, No. 1, pp. 5–18, 2003.

[GHC2004] J. Gao, J. Hu and J. J. Chen, "A Federated Multi-agent System: Autonomic Control of Web Services," *Proceedings of the Third International Conference on Machine Learning and Cybernetics*, Vol. 1, pp. 1–6, Shanghai China, August 2004.

[GHMCKT2003] I. Gorton, J. Haack, D. McGee, A. Cowell, O. Kuchar and J. Thomson, "Evaluating Agent Architectures: Cougaar, Aglets and AAA," *Software Engineering for Large-Scale Multi-Agent Systems*, pp. 264–278, Portland, Oregon, USA, 2003.

[GP2001] M. Griss and G. Pour, "Accelerating Development with Agent Components," *IEEE Computer Society*, Vol. 34, No. 5, pp. 37–43, May 2001.

[HCK1996] C. G. Harrison, D. Chess, and A. Kershenbaum, "Mobile Agents: Are they a good idea?" *Lecture Notes in Computer Science*, Vol. 1222, pp. 25–45, Springer-Verlag, ISBN: 3-540-62852-5, July 1996.

[HCMKK2000] D. Horvat, D. Cvetkovie, V. Milutionvie, P. Koeovie and V. Kovacevie, "Mobile Agents and Java Mobile Agents Toolkits," *Proceedings of the 33rd Hawaii International Conference on System Sciences*, pp. 3090–3099, Hawaii, USA, January 2000.

[Horn2001] Paul Horn, "Autonomic Computing: IBM Perspective on the State of Information Technology," *International Business Machines Corp. Autonomic Computing, http://www.research.ibm.com/autonomic/manifesto/*, October 2001.

[Jennings2000] N. R. Jennings, "On Agent-based Software Engineering," *Artificial Intelligence*, Vol. 117, pp. 277–296, April 2000.

[Jones2002] Anita Jones. "Grand Research Challenges in Information Systems Final Report," *Conference on "Grand Research Challenges in Computer Science and Engineering*," Virginia, June 2002.

[KC2003] J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *IEEE Computer,* Vol. 36, No. 1, pp. 41–50, January 2003.

[KGR2002] D. Kotz, R. Gary, and D. Rus, "Future Directions for Mobile Agent Research," *IEEE Distributed Systems Online*, Vol. 3, No. 8, pp. 8–14 August 2002.

[KSN2000] M. T, Kone, A. Shimazu, and T. Nakajima, "The State of the Art in Agent Communication Languages," *Knowledge and Information Systems,* Vol. 2, No. 3, pp. 259–284, August 2000.

[LO1998] B. Danny, Lange and Mitsuru Oshima, *Programming and Deploying Java Mobile Agents with Aglets,* Addison-Wesley, 225 pages, ISBN 0-201-32582-9, 1998.

[LO1999] D. B. Lange and M. Oshima, "Seven Good Reasons for Mobile Agents," *Communication of the ACM*, Vol. 42, No. 3, pp. 88–89, March 1999.

[MLC1999] D. S. Milojicic, W. LaForge, and D. Chauhan, *Mobile Objects and Agents Mobility: processes, computers, and agents,* Addison-Wesley, 651 pages, ISBN: 0-201-37928-7, 1999.

[MNNS2004] A. Mawlood-Yunis, A, Nayak, D. Nussbaum and N. Santoro, "Comparing Performance of Two Mobile Agent Platforms in Distributed Search," *Proceedings of the IEEE/XIC/ACM International Conference on Intelligent Agent Technology (IAT'04)*, pp. 425–428, Beijing, China, September 2004.

[Pour2003] Glida Pour, "Agent-Oriented Software Engineering (AOSE): its emergence as a cornerstone of enterprise software engineering education," *World Transactions on Engineering and Technology Education*, Vol. 2, No. 2, pp. 225–228, September 2003.

[PYL2005] Jeongmin Park, Giljong Yoo, and Eunseok Lee, "Proactive Self-Healing System based on Multi-Agent Technologies," *Proceedings of the 2005 Third ACIS Int'l Conference on Software Engineering Research, Management and Application (SERA'05),* Mt. Pleasant, MI, USA, IEEE Computer Society Press, pp .256–263, August 2005.

[QS2001] Tong-Seng Quah and Andreas Schmid, "Mapping of Aglets into E-speak," *The 2001 International Conference on Internet Computing,* pp. 25–28, Las Vegas, U.S.A, June 2001.

[QS2004] Wenyu Qu and Hong Shen, "Mobile Agent-based Execution Modeling," *Proceedings of the Fourth International Conference on Hybrid Intelligent Systems (HIS 2004),* pp.148–53, Kitakyushu, Japan, December 2004.

[Searle1969] J. R. Searle, *Speech Acts: An Essay in the Philosophy of Language*, Cambridge University Press, 199 pages, ISBN 0-521-09626 X Paperback, 1969.

[Searle1979] J. R. Searle, *Expression and Meaning: Studies in the Theory of Speech Acts,* Cambridge University Press, 183 pages, ISBN 0-521-31393-7 Paperback, 1979.

[SDPV2005] R. Steele, T. Dillon, P. Pandya, and Y. Ventsov, "XML-based Mobile Agents," *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05),* pp. 42–48, Las Vegas, Nevada, USA, April 2005.

[SDSL1999] G. Samaras, M. Dikaiakos, C. Spyrou and A. Liberdos, "Mobile Agent Platforms for Web-databases: A Qualitative and Quantitative assessment," *The Joint Symposium ASA/MA'99 1st International Symposium on Agent Systems and Applications (ASA'99), 3rd International Symposium on Mobile Agents (MA'99)*, pp. 50–64, USA, October 1999.

[SSMBS1999] L. M. Silva, G. Soares, P. Martins, V. Batista and L. Santos, "The Performance of Mobile Agent Platforms," *The Joint Symposium ASA/MA'99 1st International Symposium on Agent Systems and Applications (ASA'99), 3rd International Symposium on Mobile Agents (MA'99)*, pp. 270–271, USA, October 1999.

[SSPE2004] C. Spyrou, G. Samaras, E. Pitoura and P. Evripidou, "Mobile Agents for Wireless Computing: The Convergence of Wireless Computational Models with Mobile-Agent Technologies," *Mobile Networks and Applications*, pp. 517–528, Vol. 9, No. 5, Netherlands, October 2004.

[SV2000] Peter Stone and Manuela Veloso, "Multiagent Systems: A Survey from a Machine Learning Perspective," *Autonomous Robots*, Vol. 8, pp. 345-383, 2000.

[TCWD2004] G. Tesauro, D. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart and S. White, " A Multi-Agent Systems Approach to Autonomic Computing," *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004),* pp. 464–471, New York, USA, October 2004.

[Voyager1997] ObjectSpace, Voyager, technical overview, http:// http://www.recursionsw.com/voyager.htm.

[Weiss1999] G. Weiss, ed., *Multi-Agent Systems: A Modern Approach to Distributed Artificial Intelligence,* MIT Press, 609 pages, ISBN 0-262-23203-0, 1999.

[White1996] J. White, Mobile Agents, General Magic White Paper, 1996.

[Wla2001] I. Wladawsky-Berger, "*Advancing E-business into the Future: The Grid,*" Kennedy Consulting Summit 2001, New York, November 2001.

[Wooldridge1997] M. Wooldridge "Agent-based Software Engineering," *IEEE Proceedings on Software Engineering*, Vol. 144, No. 1, pp. 26–37, February 1997.

[WPWD1997] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet, "Concordia: An infrastructure for collaborating mobile agents," *Lecture Notes in Computer Science*, Vol. 1219, pp. 86–97, Springer, 1997.

[ZH2002] Long-Wen Zhao and Yi-Bin Hou, "Matchmaking Based on Task and Capability Description," *Proceedings of the First International Conference on Machine Learning and Cybernetics*, pp. 1301–1305, Beijing, China, November 2002.

Appendix A: Transaction Procedures

(1) UA sends purchase query to BA

(2) BA creates a purchase order and replies to UA

(3) UA sends a purchasing order confirmation to BA

(4) BA handles purchasing order from UA, and creates a MBA to travel

(5) MBA arrives at a SA, informs SA of its arrival

***** SA sends an inventory query message to IA. ****

***** SA receives n inventory reply from Inventory. ****

(6) MBA sends request cfp to SA

(7) SA receives a cfp from MBA

(8) The seller is negotiating with MBA

Seller get next sell price:6.0

(9) MBA receives a proposal from SA. The MBA is negotiating with Seller

MBA get next buy price:1.0

(10) SA receives a proposal from MBA

(11) MBA receives reject-proposal form SA

***** Dispatch the mobile agent to one destination: atp://uluru:4500/

(12) SA receives reject-proposal from MBA

***** Addr: atp://uluru:4500/ place:

No integrity checks because no security domain is authenticated.

(11) MBA receives accept-proposal from SA

***** Dispatch the mobile agent to one destination: atp://uluru:4500/

***** Addr: atp://uluru:4500/ place:

(12) SA receives MBA accept-proposal

***** Dispatch the mobile agent to one destination: atp://uluru:4500/

***** Addr: atp://uluru:4500/ place:

No integrity checks because no security domain is authenticated.

(13) MBA returns home

(14A) MBA process the purchasing result

*** Result seller url is: atp://uluru:5000/

*** Result seller url is: atp://uluru:5200/

(14B) MBA create a MRA to SA

***** Dispatch the mobile agent to one destination: atp://uluru:5000/

***** Addr: atp://uluru:5000/ place:

No integrity checks because no security domain is authenticated.

(15A) Mobile reply agent sends message to SA: atp://uluru:5100/

(15B) SA receives confirmation from MPA

***** SA sends an inventory update message to IA. ****

***** SA receives an inventory update confirmation from IA. ****

***** Dispatch the mobile agent to one destination: atp://uluru:5000/

***** Addr: atp://uluru:5000/ place:

No integrity checks because no security domain is authenticated.

(16) MBA sends purchasing result to BA

***** The purchasing result is ******

<?xml version="1.0" encoding="UTF-8"?>

<Order><Product><Name>Apple</Name><Quantity>1</Quantity><Price>1.568926313

445650

1</Price></Product></Order>

(17) BA receives purchasing report from MBA, and sends confirmation to MBA

(18) MBA disposed

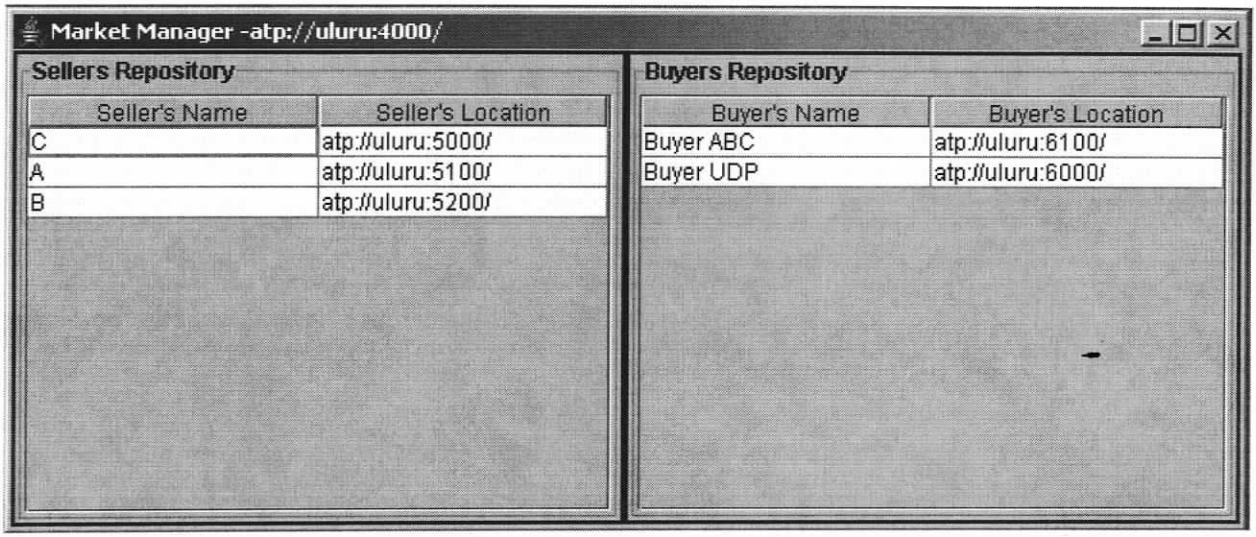
(19) BA sends purchasing result to UA

(20) UA process the purchasing result from BA

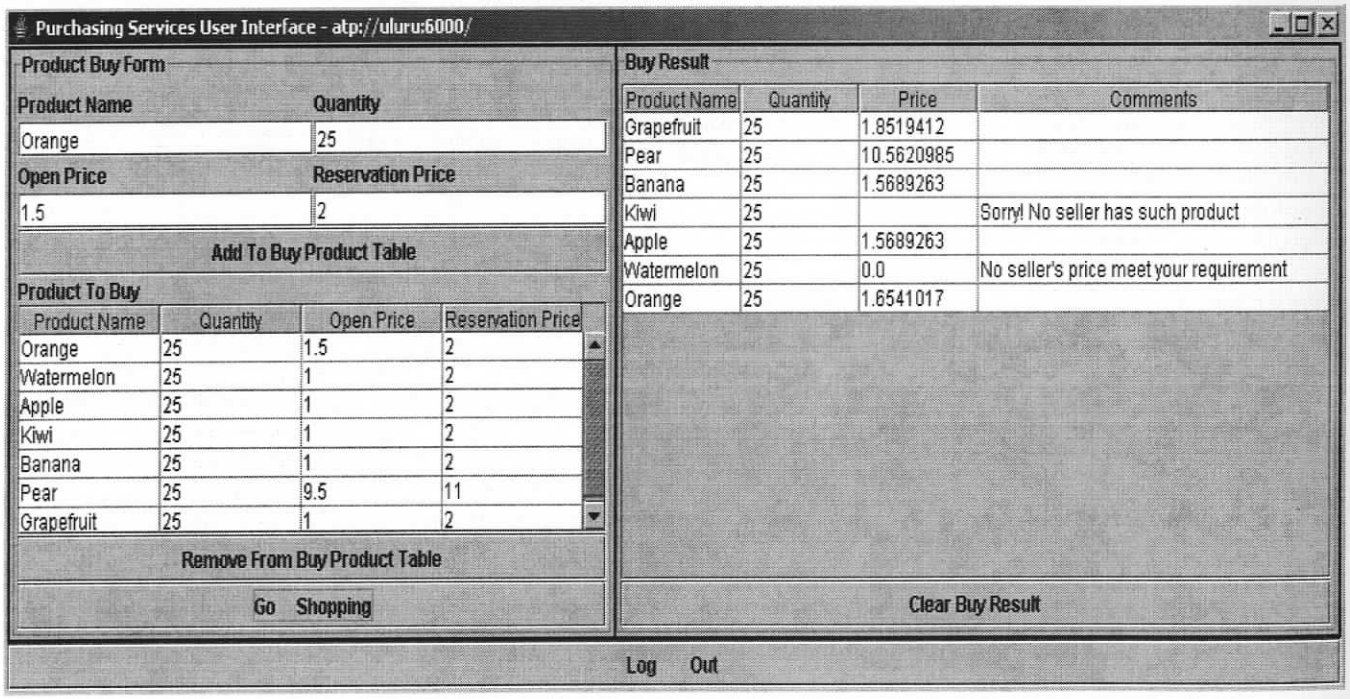
********* The transaction is completed. *********

Appendix B: User Interface

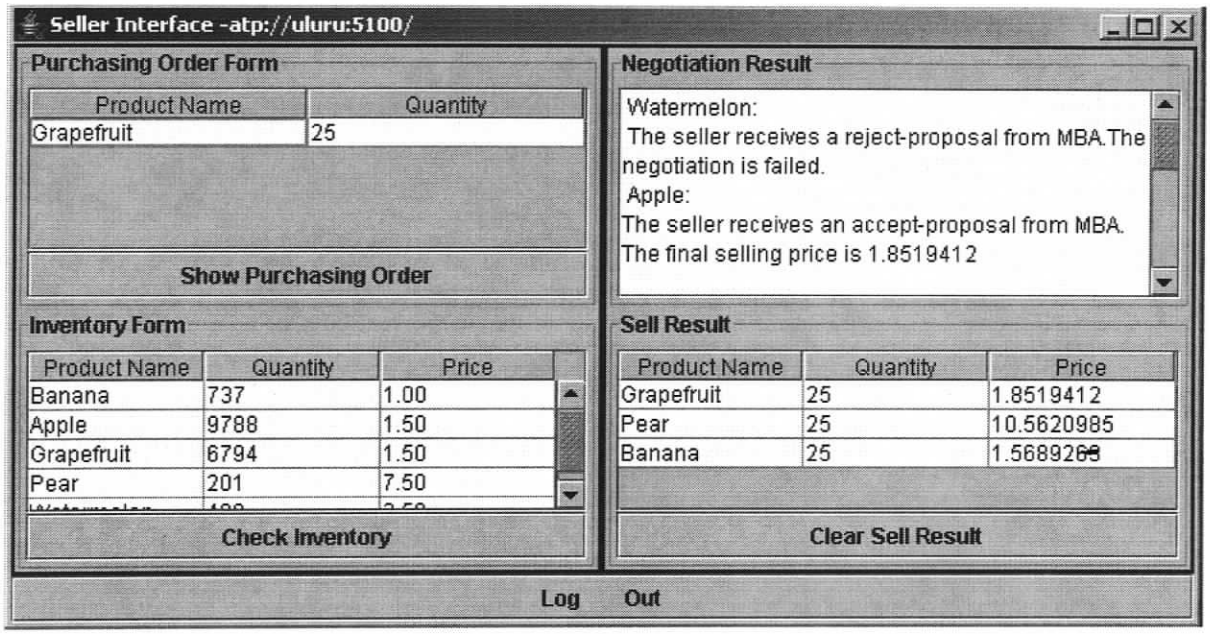
B.1 User Interface for Market Manager Agent



B.2 User Interface for Buyer Agent:



B.3 User Interface for Seller Agent



Appendix C: Selected Source Code

C.1 Source Code for User Agent

```
public class UserAgent extends SuperAgent{
        public UserInterface ui;
        private AgletProxy proxy;
        private URL originURL;
        private String buyerName;

        private String  orderContent = "";
        private Vector itemsToBuy;
        private double openPrice, reservePrice;
        private String productName = "";
        private int quantity;

        public void onCreation (Object init){
                context = getAgletContext();
                context.setProperty("UserAgent", getAgletID());

                proxy = context.getAgletProxy(getAgletID());
                originURL = context.getHostingURL();

                ui = (UserInterface)((Object[])init)[0];
                ui.setUserAgent(this);
        }

        public boolean handleMessage (Message msg){
                String convID = (String)msg.getArg(":conversation-id");
                if(findConversation(convID)){
                        Conversation conv = getConversation(convID);
                        conv.handleMessage(msg);
                }else{
                        System.out.println("No such conversation in UA.");
                }
                return true;
        }

        public void processPurchaseOrder(Vector products){
                itemsToBuy = products;
                Conversation conv = createConversation("UAQueryProtocol");
                conv.startConversation();
        }

        public void setBuyProduct(Vector item){
                productName = (String)item.get(0);
                quantity = Integer.parseInt(item.get(1).toString());
                openPrice = Double.parseDouble(item.get(2).toString());
                reservePrice = Double.parseDouble(item.get(3).toString());
        }
```

```
public void sendOrderToBA(Message msg ){
        Conversation conv = createConversation("UAOrderProtocol");
        conv.startConversation( );
}

public void showBuyResult(String xmlString){
        MySAXDriver driver = new MySAXDriver();
        Vector v = driver.checkOrder(xmlString);
        if( Double.parseDouble(v.get(2).toString()) == 0){
                v.add(3, "No seller's price meet your requirement");
        }else if ( Double.parseDouble(v.get(2).toString()) == -1){
                v.insertElementAt("", 2);
                v.add(3, "Sorry! No seller has such product");
        }else{
                v.add(3, "");
        }
        ui.showBuyResult(v);
}

public void deleteBuyer(String buyerName){
        this.buyerName = buyerName;
        createMDA();
}

private void createMDA(){
        try{
                Object args = new Object[]{buyerName, originURL, proxy};
                context.createAglet(getCodeBase(), "agents.MobileDeleteAgent", args);
        }catch(Exception e){
                System.out.println("Can not create a MDA.");
        }
}

 public String getContent(){
        return orderContent;
}

public double getOp(){
        return openPrice;
}

public double getRp(){
        return reservePrice;
}

public int getQty(){
        return quantity;
}

public String getProductName(){
        return productName;
}

 public Vector getItems(){
        return itemsToBuy;
}
}
```

C.2 Source Code for Seller Agent

```
public class SellerAgent extends SuperAgent {

        private int counter = 0;
        private SellerInterface ui;
        private String buyProductName;
        private int buyProductQty;
        private Strategy strategy[] = new Strategy[3];
        private String sellerName;
        private AgletProxy proxy;
        private double sellPrice=0, buyPrice=0;
        private AgletID MBAId;
        private String MBAConvID;
        private URL originURL;

        public void onCreation(Object init){
                context = getAgletContext();
                context.setProperty("SellerAgent", getAgletID());
                context.setProperty("Agent", getAgletID());

                proxy = context.getAgletProxy(getAgletID());
                if( (((((Object[])init) [0]).toString()).length())== 1){
                        originURL = context.getHostingURL();
                        sellerName = (((Object[])init) [0]).toString();

                        createMRA();
                }else{
                        originURL = (URL)((Object[])init)[0];
                        sellerName = (String)((Object[])init)[1];
                        //proxy = (AgletProxy)((Object[])init)[2];
                }

                ui = new SellerInterface(this, originURL.toString());
                ui.setVisible(true);
                ui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }

        private void createMRA(){
                try{
                        Object args = new Object[]{sellerName, originURL, proxy};
                        context.createAglet(getCodeBase(), "agents.MobileRegisterAgent", args);
                }catch(Exception e){
                        System.out.println("Can not create a MRA.");
                }
        }

        public void deleteSeller(){
                createMDA();
        }

        private void createMDA(){
                try{
                        Object args = new Object[]{sellerName, originURL, proxy};
```

```
                context.createAglet(getCodeBase(), "agents.MobileDeleteAgent", args);

        }catch(Exception e){
                System.out.println("Can not create a MDA.");
        }
}

public boolean handleMessage(Message msg){
        Conversation conv;
        String convID = (String)msg.getArg(":conversation-id");
        AgletID id = (AgletID)msg.getArg(":sender");

        if(findConversation(convID)){
                conv = getConversation(convID);
                conv.handleMessage(msg);
        }else if(msg.sameKind("inform")){
                try{
                        showBuyProduct(msg);
                        checkInventory();
                        MBAId = (AgletID)msg.getArg(":sender");
                        MBAConvID =  (String)msg.getArg(":conversation-id");
                }catch(Exception e){
                        System.out.println("MBA arrival exception.");
                }
        }else if (msg.sameKind("confirm")){
                if(msg.getArg(":protocol").equals("MRAReportProtocol")){
                        System.out.println("SA get registration confirmation.");

                        Message m = new Message("confirm");
                        m.setArg(":conversation-id", convID);
                        m.setArg(":sender", msg.getArg(":sender"));
                        m.setArg(":protocol", "MRAReportProtocol");
                        sendMessage(id, m);
                }else{
                        System.out.println("(15B). SA receives reply from MPA.");
                        msg.setArg(":conversation-id", msg.getArg(":conversation-id"));
                        msg.setArg(":sender", msg.getArg(":sender"));
                        msg.setArg(":portocol", "contract-net");
                        showSellResult();
                        updateInventory();
                }
        }else if(msg.sameKind("query-if")){
                System.out.println("SA send confirmation to detector.");

                Message m = new Message("confirm");
                m.setArg(":sender", msg.getArg(":sender"));
                m.setArg(":protocol", "MDADetectProtocol");
                sendMessage(id, m);
        }else {
                conv = createConversation("SANegotiateProtocol");
                conv.handleMessage(msg);
        }
        return true;
}

public void negotiateProposal( Message m){
        System.out.println("(8) The seller is negotiating with MBA.");
```

```
Vector item = (Vector)m.getArg(":content");
Object price = m.getArg(":buyerPrice");

try{
        if(price == null) {
                buyPrice = 0.0;
        }else {
                buyPrice = Double.parseDouble( m.getArg(":buyerPrice").toString());

        }

        for(int i = counter; i< 3; i++){
                sellPrice = strategy[i].getNextPriceValue();
                if(buyPrice >= sellPrice){
                        Message msg = new Message("accept-proposal");
                        AgletID mbaID = (AgletID)m.getArg(":sender");
                        String convID = (String)m.getArg(":conversation-id");
                        msg.setArg(":sender", getAgletID());
                        msg.setArg(":conversation-id", convID);
                        msg.setArg(":sellerPrice", sellPrice);
                        msg.setArg(":buyerPrice", buyPrice);
                        sendMessage(mbaID, msg);
                        addAccepNego();
                        break;
                }else {
                        if(strategy[i].getNextPrice()){

                                sellPrice = strategy[i].getNextPriceValue();
                                System.out.println("    Seller get next sell price:" + sellPrice);
                                Message msg = new Message("propose");

                                AgletID mbaID = (AgletID)m.getArg(":sender");
                                String convID = (String)m.getArg(":conversation-id");
                                msg.setArg(":sender", getAgletID());
                                msg.setArg(":conversation-id", convID);
                                msg.setArg(":sellerPrice", sellPrice);
                                msg.setArg(":buyerPrice", buyPrice);
                                sendMessage(mbaID, msg);
                                break;
                        }else if(counter == 2){
                                Message msg = new Message("reject-proposal");

                                AgletID mbaID = (AgletID)m.getArg(":sender");
                                String convID = (String)m.getArg(":conversation-id");
                                msg.setArg(":sender", getAgletID());
                                msg.setArg(":conversation-id", convID);
                                msg.setArg(":sellerPrice", sellPrice);
                                msg.setArg(":buyerPrice", buyPrice);

                                sendMessage(mbaID, msg);
                                addRejectNego();

                                break;
                        } else {
                                counter++;
```

```
                                        System.out.println("Change to another strategy to negotiate with MBA.");
                                        }
                                }
                        }
                }catch (Exception e){
                        System.out.println("Seller negotiate proposal exception.");
                }
}

public void addAccepNego(){
        ui.addNegotiation(" "+ buyProductName + ": " + "\n"+
                                        "The seller receives an accept-proposal from MBA. " +
                                        "The final selling price is " +
                                        new Double(buyPrice).floatValue()+"\n");
}

public void addRejectNego(){
        ui.addNegotiation(" "+ buyProductName + ": " + "\n"+
                                        " The seller receives a reject-proposal from MBA." +
                                        "The negotiation is failed." );
}

public void setStrategy(double price){
        strategy[0] = new HoldOutSellStrategy(price*3, price, 1.2);
        strategy[1] = new LinearSellStrategy(price*3, price);
        strategy[2] = new ConcedeSellStrategy(price*3, price, 0.8);
}

public void showBuyProduct(Message msg){
        Vector item = (Vector)msg.getArg(":content");
        buyProductName = (String)item.get(0);
        buyProductQty =  Integer.parseInt(item.get(1).toString());
        ui.showBuyProduct();
}

public void checkInventory(){
        Conversation conv = createConversation("SAQueryProtocol");
        conv.startConversation();
}

public void showInventory(LinkedList[] v){
        boolean hasProduct = false;
        int size = v[0].size();
        for(int i =0; i < size; i++){
                if((v[0].get(i)).equals(buyProductName )){
                        setStrategy(Double.parseDouble((v[2].get(i)).toString()));
                        hasProduct = true;
                }
        }
        ui.showInventory(v);
        if(hasProduct){
                Message m  = new Message("confirm");
                m.setArg(":sender",getAgletID());
                m.setArg(":conversation-id", MBAConvID);
                sendMessage(MBAId, m);
```

```
                hasProduct = false;
            }else{
                Message m  = new Message("refuse");
                m.setArg(":sender",getAgletID());
                m.setArg(":conversation-id", MBAConvID);
                sendMessage(MBAId, m);
            }
        }

        public void updateInventory(){
            Conversation conv = createConversation("SAUpdateProtocol");
            conv.startConversation();
        }

        public void showSellResult(){
            Vector item = new Vector();
            float price = new Double(buyPrice).floatValue();
            item.add(buyProductName);
            item.add(buyProductQty+"");
            item.add(price+"");
            ui.showSellResult(item);
        }

        public String getBuyProName (){
            return buyProductName;
        }

        public int getBuyQty(){
            return buyProductQty;
        }

        public String getType(){
            return sellerName;
        }
        public double getBuyPrice(){
            return buyPrice;
        }

}
```

C.3 Source Code for Buyer Agent

```
public class BuyerAgent extends SuperAgent{
        private String buyerName;
        private AgletProxy proxy;
        private URL originURL;
        private Vector sellersURL = new Vector();
        private UserInterface ui;
        private String converID, productName, openPrice, reservePrice, quantity;

        public void onCreation (Object init){
                context = getAgletContext();
                context.setProperty("BuyerAgent", getAgletID());
                context.setProperty("Agent", getAgletID());

                proxy = context.getAgletProxy(getAgletID());
                originURL = context.getHostingURL();

                if((((Object[])init)[0].toString().substring(0, 5)).equals("Buyer")){
                        buyerName = (((Object[])init) [0]).toString();
                        ui.setBuyerAgent(this, buyerName);
                }else{
                        ui = (UserInterface)(((Object[])init) [0]);
                        buyerName = (((Object[])init) [1]).toString();
                        ui.setBuyerAgent(this, buyerName);
                        createMRA();
                }
        }

        public boolean handleMessage (Message msg){
                Conversation conv;
                String convID = (String)msg.getArg(":conversation-id");
                AgletID id = (AgletID)msg.getArg(":sender");

                if(findConversation(convID)){
                        conv = getConversation(convID);
                        conv.handleMessage(msg);
                }else if (msg.sameKind("confirm")){
                        if(msg.getArg(":protocol").equals("MRAReportProtocol")){
                                System.out.println("BA get registration confirmation.");

                                Message m = new Message("confirm");
                                m.setArg(":conversation-id", convID);
                                m.setArg(":sender", msg.getArg(":sender"));
                                m.setArg(":protocol", "MRAReportProtocol");
                                sendMessage(id, m);
                        }
                }else if (msg.sameKind("inform")){
                        System.out.println("BA get all sellers info. from MCA.");
                        sellersURL = (Vector)msg.getArg(":content");

                        Message m = new Message("confirm");
                        m.setArg(":conversation-id", convID);
                        m.setArg(":sender", msg.getArg(":sender"));
                        sendMessage(id, m);
```

```
                createMBA();
        }else if(msg.sameKind("query-if")){
                if(msg.getArg(":protocol").equals("MDADetectProtocol")){
                        System.out.println("BA send confirmation to detector.");

                        Message m = new Message("confirm");
                        m.setArg(":sender", msg.getArg(":sender"));
                        m.setArg(":protocol", "MDADetectProtocol");
                        sendMessage(id, m);
                }else {
                        conv = createConversation("BAOrderProtocol", convID);
                        conv.handleMessage(msg);
                }
        }else {
                conv = createConversation("BAOrderProtocol", convID);
                conv.handleMessage(msg);
        }

        return true;
}

private void createMRA(){
        try{

                Object args = new Object[]{
                        buyerName,
                        originURL,
                        proxy,
                };
                context.createAglet(getCodeBase(), "agents.MobileRegisterAgent", args);

        }catch(Exception e){
                System.out.println("Can not create a MRA.");
        }
}

public void deleteBuyer(){
        createMDA();
}

private void createMDA(){
        try{

                Object args = new Object[]{
                                buyerName,
                                originURL,
                                proxy
                };
                context.createAglet(getCodeBase(), "agents.MobileDeleteAgent", args);

        }catch(Exception e){
                System.out.println("Can not create a MDA.");
        }
}
```

```
public void createMBA(){
        System.out.println("Create MBA to travel.");
        try{
                sellersURL.add(originURL);
                for(int i =0; i < sellersURL.size(); i++){
                        System.out.println(sellersURL.get(i));
                }
                Object args = new Object[]{
                                new ConcreteItinerary(originURL,sellersURL),
                                proxy,
                                converID,
                                productName,
                                quantity,
                                openPrice,
                                reservePrice
                };
                context.createAglet(getCodeBase(), "agents.MobileBuyerAgent", args);
        }catch(Exception e){
                System.out.println("Can not create a MBA.");
        }
}

public void createMCA(Message msg, String convID){
        System.out.print ("(4). Create MCA to find all sellers.");

        converID = convID;

        Vector content = (Vector)msg.getArg(":content");
        productName = (String)(content.get(0));
        quantity =  (content.get(1)).toString();
        openPrice =  (content.get(2)).toString();
        reservePrice =  (content.get(3)).toString();

        try{

                Object args = new Object[]{
                                buyerName,
                                originURL,
                                proxy,
                };
                context.createAglet(getCodeBase(), "agents.MobileCheckAgent", args);

        }catch(Exception e){
                System.out.println("Can not create a MCA.");
        }
}


public String createPurchasingOrder(Vector v){
        MyDOMDriver DOMDriver = new MyDOMDriver();
        String xmlContent = DOMDriver.createPurchasingOrder(v);
        return xmlContent;
}
}
```

C.4 Source Code for MBA Negotiation Protocol

```
public class MBANegotiateProtocol extends Protocol {
        private MobileBuyerAgent agent;
        private String convID;
        private int counter;
        private Strategy strategy[] = new Strategy[3];
        private double sellPrice=0, buyPrice=0;

        public MBANegotiateProtocol(MobileBuyerAgent agent, String convID){
                this.agent = agent;
                this.convID = convID;
                counter = 0;
                setStrategy();
        }

        public void handleMessage(Message msg){
                if(msg.sameKind("refuse")){
                        System.out.println("(11).MBA receive refuse propose from SA.");
                        agent.goNextDestination();
                }else if(msg.sameKind("propose")){
                        System.out.print("(9).MBA receive a proposal from SA.");
                        negotiateProposal(msg);
                }else if(msg.sameKind("accept-proposal")){
                        System.out.println("(11).MBA receive accept-proposal from SA.");
                                                agent.putResultTable(buyPrice + "");
                        agent.goNextDestination();
                }else if(msg.sameKind("reject-proposal")){
                        System.out.println("(11).MBA receive reject-proposal form SA.");
                        agent.goNextDestination();
                }
        }

        public void start(){
                Vector item = new Vector();
                item.add(agent.getProductName());
                item.add(agent.getQty() + "");

                System.out.println("(6). MBA sends a request cfp to SA.");
                Message m  = new Message("cfp");
                m.setArg(":sender", agent.getAgletID());
                m.setArg(":conversation-id", convID);
                m.setArg(":content", item );
                m.setArg(":protocol", "contract-net");
                agent.sendMessageToSA(m);
        }


        public void negotiateProposal(Message m){
                System.out.println("The MBA is negotiating with Seller.");

                Object price = m.getArg(":sellerPrice");
                try{
                        if(price == null) {
                                sellPrice = 100.0;
```

```
                    } else {
                        sellPrice = Double.parseDouble(m.getArg(":sellerPrice").toString());
                    }

                    for(int i = counter; i< 3; i++){
                        buyPrice = strategy[i].getNextPriceValue();
                        if(buyPrice >= sellPrice){
                            agent.putResultTable(buyPrice + "");
                            Message msg = new Message("accept-proposal");

                            msg.setArg(":sender", agent.getAgletID());
                            msg.setArg(":conversation-id", convID);
                            msg.setArg(":sellerPrice", sellPrice);
                            msg.setArg(":buyerPrice", buyPrice);

                            agent.sendMessageToSA(msg);
                            agent.goNextDestination();
                            break;
                        }else {
                            if(strategy[i].getNextPrice()){

                                buyPrice = strategy[i].getNextPriceValue();
                                System.out.println("    MBA get next buy price:" + buyPrice);
                                Message msg = new Message("propose");

                                msg.setArg(":sender", agent.getAgletID());
                                msg.setArg(":conversation-id", convID);
                                msg.setArg(":sellerPrice", sellPrice);
                                msg.setArg(":buyerPrice", buyPrice);
                                agent.sendMessageToSA(msg);
                                break;

                            }else if(counter == 2) {
                                Message msg = new Message("reject-proposal");

                                msg.setArg(":sender",agent.getAgletID());
                                msg.setArg(":conversation-id", convID);
                                msg.setArg(":sellerPrice", sellPrice);
                                msg.setArg(":buyerPrice", buyPrice);
                                agent.sendMessageToSA(msg);
                                //agent.removeConversation(convID);
                                agent.goNextDestination();
                                break;
                            } else {
                                counter++;
                                System.out.println("Change to another strategy to negotiate with
                                seller.");
                            }
                        }
                    }

                }catch (Exception e){
                    System.out.println("MBA negotiate proposal exception.");
                }

            }
        }
```

C.5 Source Code for Seller Negotiation Protocol

```
public class SANegotiateProtocol extends Protocol {
        private SellerAgent agent;
        private String convID;

        public SANegotiateProtocol (SellerAgent agent, String convID){
                this.agent = agent;
                this.convID = convID;
        }

        public void handleMessage(Message msg){
                if(msg.sameKind("cfp")){
                        System.out.println("(7). SA receives a cfp from MBA.");
                        msg.setArg(":conversation-id", msg.getArg(":conversation-id"));
                        msg.setArg(":sender", msg.getArg(":sender"));
                        msg.setArg(":protocol", "contract-net");
                        agent.negotiateProposal( msg);
                }else if(msg.sameKind("propose")){
                        System.out.println("(10). SA receives a proposal from MBA.");
                        msg.setArg(":conversation-id", msg.getArg(":conversation-id"));
                        msg.setArg(":sender",msg.getArg(":sender"));
                        msg.setArg(":portocol", "contract-net");
                        agent.negotiateProposal( msg);
                }else if (msg.sameKind("accept-proposal")){
                        System.out.println("(12). SA receives MBA accept-proposal.");
                        agent.addAccepNego();
                }else if (msg.sameKind("reject-proposal")){
                        System.out.println("(12). SA receives reject-proposal from MBA.");
                        agent.addRejectNego();
                        //agent.removeConversation(convID);
                }else if(msg.sameKind("cancel")){
                        System.out.println("(15B). SA receives cancellation from MPA.");
                        agent.removeConversation(convID);
                }else if(msg.sameKind("confirm")){
                        System.out.println("(15B). SA receives confirmation from MPA.");
                        msg.setArg(":conversation-id", msg.getArg(":conversation-id"));
                        msg.setArg(":sender", msg.getArg(":sender"));
                        msg.setArg(":portocol", "contract-net");
                        agent.showSellResult();
                        agent.updateInventory();
                        //agent.removeConversation(convID);
                }
        }
}
```