

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]



Université d'Ottawa • University of Ottawa

A Platform for Mobile Agent-based Data Access, Retrieval, and Interaction

by

Benjamin W. Falchuk, *B.Math., M.Sc.*

**A Thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of**

**Doctor of Philosophy
in Electrical and Computer Engineering**

**Ottawa-Carleton Institute of Electrical and Computer Engineering
School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa
December, 1998**

©1998, Benjamin W. Falchuk, Ottawa, Canada



National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-46518-7

Contents

Abstract	v
Acknowledgments	vi
1. Introduction	1
1.1 Motivation	1
1.2 Summary of Contributions	4
1.3 Limits of this Research	6
1.4 Outline	7
2. Background & Related Work	9
2.1 Introduction	9
2.2 Background	9
2.2.1 Origins	10
2.2.2 Potential Advantages of the Mobile Agent Approach	12
2.2.3 Potential Disadvantages of the Mobile Agent Approach	13
2.3 Related Work	13
2.3.1 Agent Systems, Architectures, and Java-based Platforms	13
2.3.2 Agent Communication	16
2.3.3 Mobile Agent Behavior Specification	20
2.3.4 Comparison at a Glance	22
2.4 Agent Issues not Addressed in this Thesis	24
2.4.1 Security	24
2.4.2 Artificial Intelligence	26
2.4.3 Object Serialization	27
2.4.4 Distributed Object Technology and Platforms	28
2.5 Summary	29
3. Communications – Inter-Agent and Inter-AEE Protocols	31
3.1 Introduction	31
3.2 An Inter-Agent Protocol - <i>AgentTransact</i>	32
3.2.1 A Shared Data Ontology	32

3.2.1.1 Extending and Sharing	39
3.2.2 Speech Acts and Conversational Modes	40
3.2.3 <i>AgenTransact</i> Message Format and Grammar	46
3.2.4 An Example	48
3.3 An Inter-AEE Protocol - <i>AgenTransfer</i>	50
3.3.1 Extending ATP/0.1 Functionality	53
3.3.2 An Example	66
3.4 Summary	66
4. Agent Execution Environment and Platform	68
4.1 Introduction	68
4.1.1 Summary of ObjectSpace's Voyager Platform	68
4.2 A Layered Agent-Stack	70
4.3 The Agent Execution Environment	72
4.3.1 Example Scenarios	79
4.4 The Mobile Agent, Facilitator and Blackboard Mechanisms	81
4.4.1 Supporting Structures	85
4.5 Example AEE Trace	87
4.6 Summary	87
5. The Agent Inception System	89
5.1 Introduction	89
5.1.1 A Platform-independent Itinerary Format	91
5.1.2 Dividing and Consolidating the Itinerary	92
5.2 Component Functionality and Description of AIS Managers	92
5.2.1 Class Hierarchy of AIS Components	102
5.2.2 Example Scenarios	105
5.3 Creating LiveAgents and Itineraries Visually	108
5.3.1 Minimal Specification Requirements for Mobile Agents	108
5.3.2 The Icon Set	111
5.3.3 Techniques for Icon Combinations for Itinerary Generation	113
5.3.4 Examples	118
5.4 Summary	121

6. Conclusions	125
6.1 Limitations	126
6.2 Suggestions for Future Research	127
References	129
Appendix A – Pseudo-code	135
Appendix B – Publications based on this Thesis	153

Abstract

Mobile software agents have emerged as a promising paradigm for computing at both the implementation and conceptual levels. While some will argue that the paradigm is not entirely “new”, it is nonetheless more flexible and dynamic than the client-server one. In the mobile agent model the program (e.g. agent) is sent to, and runs on, remote machines and so operates closer and more interactively with the remote resource. This flexible paradigm is well suited to many applications ranging from electronic Internet commerce, to military computing and telecommunication and therefore its exploration and exploitation are paramount to both industry *and* academia. Designing and abstracting the implementation of a platform using software agents and exposing the paradigm to the user via powerful graphical user interfaces is the general goal of this thesis. While both network bandwidth and user search time can be reduced when the mobile agent is employed for information retrieval on such a platform, enabling this platform requires protocols at the agent language, ontology, transfer, and application levels. To these ends this thesis proposes: (i) an “agent-stack” and protocols that enable automated mobile agent-based digital document access, retrieval, and interaction. *AgenTransfer* for inter-server transfer, *AgenTransact* for inter-agent communication, and *AgenTask* for human-agent communication, (ii) the functionality of an agent execution environment that hosts mobile agents, and (iii) a unique and powerful graphical user interface suite which can be used by novice end-users to quickly and easily program and manage mobile agents and their itineraries. An accompanying implementation of the main components proposed in this thesis serves as both proof-of-concept of the ideas and as an opportunity for analysis.

Acknowledgments

I would like to thank my supervisor, Dr. Ahmed Karmouch, who not only provided funding, hardware, and software but who also gave me the structured freedom to work on research that genuinely interested me while guiding me through a fruitful and enjoyable Ph.D. program. I enjoyed the time spent in his laboratory and the camaraderie with the other researchers. I also acknowledge my Ph.D. committee members – Dr. Luis Orozco-Barbosa and Dr. Dorina Petriu – for their comments, questions and criticisms that helped to shape this thesis into its final form. Thanks are due to my parents, as usual, for their consistent support throughout this pleasant ordeal. Finally, I would like to acknowledge the Telecommunications Research Institute of Ontario (TRIO) and the Natural Science and Engineering Research Council (NSERC), both of which provided funding that helped promote and encourage this research.

Chapter 1

Introduction

1.1 Motivation

With an abundance of data resources and vast amounts of multimedia information available on-line, it is often tedious and bandwidth wasting for users to search manually for desired information. The global Internet and the WWW, HTTP, and HTML [BER93] make it *possible* to find a huge amount of information on-line, ranging from personal home pages to up-to-the-minute news and sports information. Web browsers such as *Navigator* and *Explorer* make the browsing process simple but do not solve the information retrieval problem; Web-browsing users are very often overloaded with possibly relevant information. As client-server (C-S) tools these browsers put the onus on the *user* to continually follow hyperlinks to find desired information, and furthermore, although search tools and catalogs exist (e.g. *Lycos*TM [LYC98] and *Yahoo*TM [YAH98]), it can still be time-consuming, bandwidth wasting, and tedious to find information. The mobile agent paradigm helps alleviate these problems. In such a paradigm the program moves *to* the data, and advantages over client-server computing, in which computation occurs *across a network*, are realized. Additionally, the asynchronous nature of the paradigm allows users to migrate or log out while agents continue to work remotely on the users behalf. The potential complexity of mobile agent operation requires that mechanisms exist on several levels to coordinate these activities. Such mechanisms, including tools and protocols, are the main topic of this thesis.

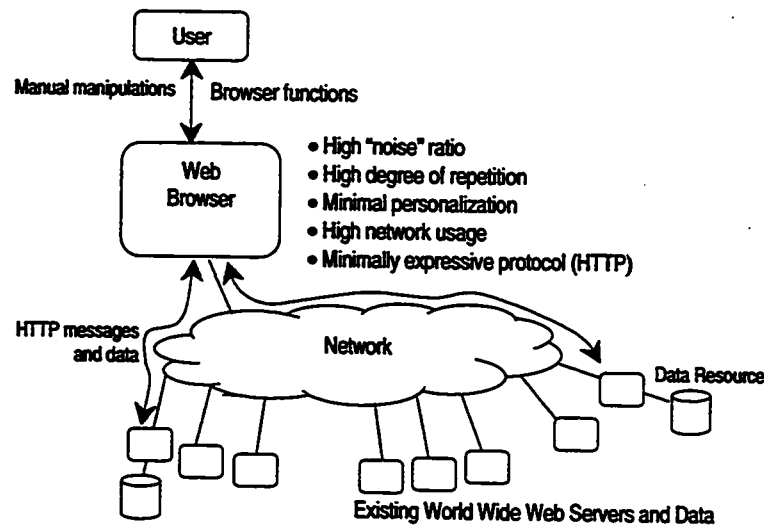


Figure 1.1 Existing information platform

Figure 1.1 illustrates the architecture of large Intranets or the Internet, focusing on on-line information, access, and retrieval. Users browse through distributed information using a Web client that messages with remote Web servers. Inherent in this process is a high degree of repetition, noise data, and network usage, and low degrees of personalization and protocol expressiveness.

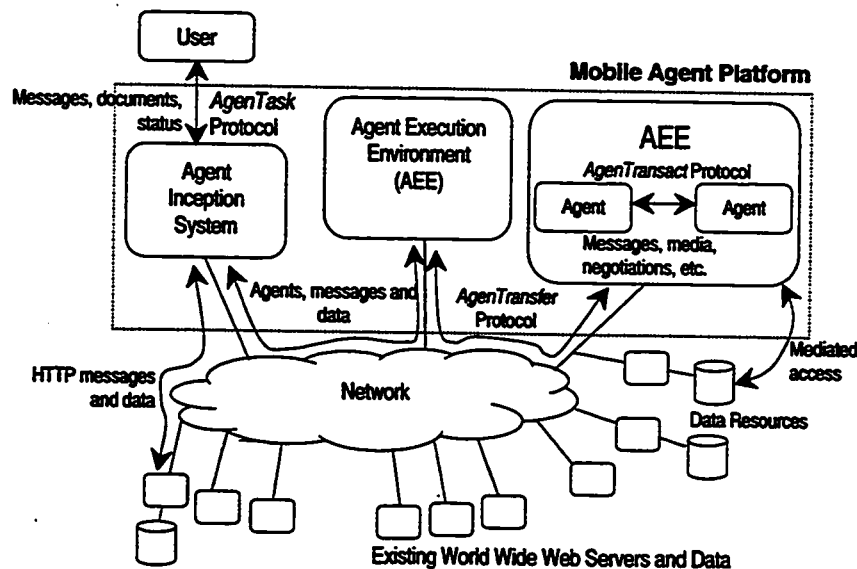


Figure 1.2 Proposed platform architecture and contributions. Such a platform both empowers and liberates the end-user and stimulates electronic data exchange and commerce.

Figure 1.2 illustrates the proposed platform in which mobile agents originate from a user-controlled *Agent Inception System*, migrate between *Agent Execution Environments*, and both interact with and collect data on behalf of a user. Such a platform enables personalized electronic commerce, automated information retrieval, and a host of other applications including automated software upgrades and seamless messaging for mobile users. On this platform, the mobile agent becomes an advanced, personal, intelligent, and timesaving “tool”.

The mobile agent-model extends the basic notions of both Postscript [ADO85] and Remote Evaluation (REV) [STA90] in the sense that the agent “object” is sent to a remote “device” for execution. Analogously, Postscript programs are sent to a printing device and REV procedures sent to remote machines. Four main concepts are integral, we believe, in this model. (1) *Agents* - programs or scripts, compiled or not, that represent the user. (2) *Agent execution environments* (AEE's) - virtual machines in which agents can run. (3) *Resources* - CPU cycles, memory, disk space, services (including databases, WWW data, etc.), and (4) *Protocols* - timing, syntax, and semantics that allow agents to intercommunicate. An informal definition of a mobile agent is as follows: *a program that is able to migrate from node to node on a network under its own control for the purpose of completing a task specified by a user. The agent chooses when and to where it will migrate and may interrupt its own execution and continue elsewhere on the network.* Note that Web Spiders, Robots [CHE96], and search engines (e.g. LycosTM) are *not* mobile agents by this definition. A precise definition of “agent” has not yet been universally accepted (see [MAE94, FRA97, MAG97, FIP98]), although Maes' definition is useful. She defines “interface agents” as: “...computer programs that employ artificial intelligence techniques to provide active personal assistance to a user with computer-based tasks” [MAE94]. Other characteristics frequently considered necessary for “agenthood” (sometimes referred to as “agenticity”) are:

- *Asynchronous* - migration, messaging, and termination occur asynchronously.
- *Peer-to-peer messaging and multiple roles* - Agents may act as both client and servers in their lifetimes
- *Collaborative* - An agent uses information and data provided by other agents.
- *Task and environment-awareness* - The agent “understands” the task it must solve and the environments in which it executes.
- *Ability to delegate.*

Many aspects of the mobile agent paradigm make it an appropriate solution to information retrieval problems. To solve those types of problems using this paradigm we require an implementation of a suite of protocols and GUI's. This suite should enable a mobile agent to be constructed by a human, launched onto a network, and enable agent-migration between environments (AEE's) for the purposes of satisfying a computer-based task. The prototypical task entails negotiating for, and acquiring multimedia information (e.g. video, images, HTML, etc.) which may also require mobile agents to share information with other agents, help other agents, divide the work amongst clones, etc. The mobility aspect of agents, while empowering the paradigm, creates a need for more intricate protocol semantics and AEE's. Multimedia aspects of data require that agents "understand" the qualifying factors related to the naming, storage, acquisition, and transfer of different data-types.

In this thesis, we focus on the functionality of the components that make up an end-to-end mobile agent platform to enable data collection and interaction amongst non-malicious mobile agents and non-malicious service agents and hosts. This thesis mainly establishes:

- An *Agent Inception System* for mobile agent creation, management, and manipulation via a unique drag-and-drop graphical notation.
- An *Agent Execution Environment* (AEE) which hosts mobile agents, serves resources, and manages agent interactions. The AEE is a part of a proposed layered *agent-stack*.
- Protocols for inter-agent and inter-AEE communications, called *AgentTransact* and *AgentTransfer* (resp.)

The platform, tools, and protocols are hereafter referred to as *AgentSys* [FAL97a, FAL97b, FAL97c, FAL98a, FAL98b, FAL99]. The remainder of this chapter provides an overview of the contributions of this thesis, followed by an outline of the rest of this document.

1.2 Summary of Contributions

This section overviews the original contributions and results of this thesis that will be presented in greater detail in subsequent chapters of this document. This thesis contributes to the field of mobile software agents in a vertical slice in the sense that our proposals exist in each of the various layers that, together from top to bottom, constitute a mobile agent system. In summary, the contributions (discussed in subsequent paragraphs) are as follows:

- An AIS and an IMT that allow customization and programming of distributed itineraries
- An agent-stack and execution environment
- Inter-agent and inter-environment protocols, syntax, and semantics •

Closest to the user and in the highest conceptual layer, we propose the *Agent Inception System* (AIS). The AIS is a small but functional set of cooperating mobile agents running on a single (usually home-base) machine that serves to persist, launch, create, and manage all aspects of mobile agents that are either user-customizable or configurable. The AIS exposes several GUI's to the end user that allow the above operations to occur. Of special note is the *Iconic Modeling Tool* (IMT) that allows users to create mobile agent itineraries by simply "drawing" them on a large canvas. The AIS is also proactive, serving as the gateway to the World Wide Web, retrieving Web-accessible data, and spidering Web servers as well as popular Web catalogs (YahooTM) and indices (ExciteTM). There has not yet been an end-user system proposed that both exposes the mobile agent paradigm to the end-user and performs the on-demand launching of mobile agents as well as accessing Web data. Furthermore, the IMT provides a unique solution to the problem of how users without programming skills can customize and program mobile agents. The IMT is generally superior to existing (moderately comparable) tools such as *AgentSheets* [REP95], *Interaction Diagrams* [RON97], and *Agent Prototyper* [TAK96]. In this thesis we call the visual protocol for task description *AgentTask*. Both the AIS and IMT were implemented and tested in the lab using Java JDK1.1.5 and Voyager [VOY97].

In order to enable competing, component-based implementations of the *AgentSys* platform, its protocols and functionality were grouped into independent layers a la the Open Systems Interconnected layered communication suite [ISO84]. In this thesis, we propose an *Agent Stack* that clearly separates the components required for a communicating agent platform. The stack consists of the *User/Service-entity*, *Facilitator/agent-entity*, and the *Agent-transfer-entity* layers. Each layer provides an Application Interface (API) to the next higher layer. Furthermore, in order to allow the execution of mobile agents on machines of arbitrary operating system and architecture, we propose a generic *Agent Execution Environment* (AEE). The AEE consists of a Java Virtual Machine (JVM) as well as a number of static agents who manage different aspects of mobile agent execution beyond what the JVM and other agent environments (such as the

Voyager [VOY97] server) provide. Together, these smallish, limited-purpose agents manage agent conversations, cargo, ontologies, privileges, and resources via inter-communication and mutual service provisioning. The Agent Stack and AEE represent unique contributions to the area of mobile agent protocol design and execution management.

Finally, this thesis proposes two simple but effective protocols ; one that enable a network of distributed AEE's to send agents, messages, and cargo amongst themselves and the other to enable mobile agents to intercommunicate for the purpose of data acquisition, sharing, and negotiating. We named the former protocol *AgentTransfer* and the latter *AgentTransact*. *AgentTransfer* is based upon an existing proposal by IBM but extends both the syntax and semantic to allow a great deal more functionality, previously unsupported. *AgentTransact*, based upon speech-act theory, uses a set of named, well-known performatives (speech-acts that induce some action) to enable meaningful exchanges of information between mobile agents. It is implemented in the same spirit as the *Knowledge Query and Manipulation Language* (KQML) [FIN97] but adds performatives that allow more explicit media and document gathering and exchange. We also propose an ontology with four main branches: network, agent, document, service. The terms in this ontology can loosely be thought of as the “nouns” of inter-agent discourse while performatives can be thought of as the “verbs”.

1.3 Limits of this Research

This thesis provides solutions to several open research issues relating to mobile software agents and their protocols and interfaces. In the process, however, assumptions have been made and other key issues have been purposely overlooked so as not to limit the development of this work. Consequently, some legitimate questions regarding mobile agent platforms remain open and unaddressed due to time and resource limitations. These issues include:

- *Agent security* – Securing agents from malicious hosts and visa versa is an ever-growing ongoing research area amongst agent-researchers. Digitally signing bytecodes and the secure exchange of privileges, media, and other sensitive data would seem to be paramount to the success of a mobile agent platform (e.g. see [FAR96, GRE98]). This thesis did not address agent security at a low-level.

- *Artificial Intelligence and Symbolic Logic* – Whether or not a procedurally implemented mobile agent that does *not* use AI techniques and does *not* carry a knowledge base of logic symbols can solve problems with the same ease as AI-based agents has not been documented. Declarative agents have been implemented however, and have had various degrees of success (e.g see [SYC98, BRA97b]). AI techniques for agents and symbolic logic for agent message-contents were not explored as a part of this thesis. Agents implemented as part of this thesis prototype were done so procedurally.
- *Distributed Object Technology* – The emergence and acceptance of the CORBA [COR95], DCOM [DCO98], and recently *Grasshopper* [GRA98] distributed object platforms has enabled programmers to completely disregard object location and system. Some of these platforms are also beginning to offer sophisticated agent services. The study of how a mobile agent system would behave on top of such an object platform was not undertaken as a part of this thesis.

Chapter 2 discusses and surveys other important agent technologies that were not studied in-depth as a part of this thesis. Chapter 6 specifies future work and suggests how some of these technologies can be integrated with *AgentSys*.

1.4 Outline

In chapter two, we provide background information on mobile agents and survey some important related work. We discuss the origins and advantages of the mobile agent paradigm and examine related work in the agent communications, behavior specification, and standards domains. In chapter three we begin presenting the contributions of this thesis. The inter-agent protocols called *AgentTransact*, and the inter-environment protocol called *AgentTransfer*, are detailed. These two protocols are unique and allow a community of agents to exchange information amongst themselves and their environments. In chapter four we describe platform required for mobile agent activity and propose a layered agent-stack to distribute the protocols. We also propose the *Agent Execution Environments* (AEE's) in which agents execute. These environments can be thought of as "the next Internet servers" and we detail a minimum specification of the functionality and operation of such environments. In chapter five, we propose the *Agent Inception System* (AIS). AIS's exist at user's computers and manage all

aspects of local agent persistence, itinerary creation, and management. They also provide a “live” window through which remote mobile agents exchange messages with their users. An essential component of the AIS is the novel *Iconic Modeling Tool*, which allows computer illiterate end-users to visually program mobile agents. Finally, in chapter six we conclude, present limitations of this research, and outline possible future work. In Appendix A, the pseudo-code for all major implemented classes is presented. In Appendix B, a list of refereed publications based on this thesis research is provided.

Chapter 2

Background & Related Work

2.1 Introduction

In 1994 Maes defined *interface agents* as “computer programs that employ artificial intelligence techniques to provide active personal assistance to a user with computer-based tasks” [MAE94]. These types of agents are generally thought of as personal assistants that perform local tasks such as E-mail filtering. The intrinsic distributed nature of the Internet has made it clear that distributed communication and mobility aspects of agents are extremely important. Agents should be able to communicate and co-operate with both local *and* remote agents and should be able to migrate to remote hosts to operate closer to physical data stores. A natural network-bandwidth reduction materializes in this case, as do other advantages. This chapter provides background information on mobile agents and compares and contrasts other related work to the contributions of this thesis.

2.2 Background

Maes’ definition of interface agents illustrates effectively how *agents* differ from *objects* and *applications*. In particular, note the adjectives “active” and “personal”. Agents can be distinguished from *programs* with respect to their degree of autonomy, goal-orientation, and other characteristics [FRA97]. Many examples of agent-based applications, including Internet-based ones, have arisen in recent years. Several are cited in [IEE97, JEN98] as well as a broad

discussion of agents, Internet-based and otherwise. Classes of applications suited to mobile agents tend to include Internet information retrieval, electronic commerce, and network management, telecommunications management, and automated software-upgrades. Large telecommunications companies, including Nortel and AT&T, are all considering how mobile agents may add value to existing and future products. Research revolving around mobile agents and distributed object platforms for telecommunications is very active [CHA97, BRE98, BON98, DIL98]. One implementation that combines mobile agents and a distributed object model is called *Grasshopper* and is MASIF-compliant (see [GRA98]).

2.2.1 Origins

In the client-server model [UMA93] there are two main entities: The *server* is the service-provider which typically waits for well-formed requests to come onto the communications port that it monitors. The *client* is the service-consumer that sends a particular request message to the server when it needs a service performed. If the request does not conform to the syntax of the server it is not understood and discarded. Remote Procedure Call (RPC) [BIR84, STO94] implements the client-server model. RPC is a programming tool in which the client uses uniquely named, clearly defined remote procedures as if they were local. A run-time client-stub detects this and makes a network connection to the server-stub that validates the formulation of the request. The remote server performs the procedure and passes the results back to the client. Typically, the client blocks until the server returns results or an error message. Other implementations of the C-S model include distributed SQL databases, SUN's NFS file-system, and LAN print services [STA94, UMA93]. Although widely adopted, the C-S model can be limiting. Some of its limitations are its strict typecasting, lack of dynamism (e.g. with respect to node failure and server load), lack of interactivity, and lack of customization or personalization.

The mobile agent-model builds upon and extends the basic concepts of client server computing. It is often thought of as an "evolution" of both Postscript [ADO85] and Remote Evaluation (REV) [STA90]. A Postscript file is in effect a program that is sent to a remote printer device; the program instructs the printer how it should be printed. A mobile agent is sent to a remote "execution device" and informs that device what services it would like. REV allows procedures to be sent to a remote machine for execution, providing that all variables are provided at the

server or within the procedure. A mobile agent is essentially a mobile (possibly) procedural object (in the programming sense). Unlike in REV, agent “objects” may interact dynamically with other remote objects to read variables. Mobile agents may also migrate to other devices beyond the original destination device, whereas Postscript and REV entities hop only once and terminate. Many researchers now agree that the three (albeit arguable) main contributing technologies to mobile agents are [JEN98]:

- Artificial intelligence
- Object-oriented programming and concurrent object-based systems
- Human-computer interface design.

Nonetheless, many other sister technologies enable mobile agents, including operating system design, concurrent programming and distributed systems.

Agents are often confused with other entities. In our view, agents are different than other related technologies in the following ways:

- *Applications* – Agents are smaller than classic applications (e.g. word processors, spreadsheets)
- *Objects* – Agents are more autonomous than objects. For example, any object can invoke another’s public method while an agent may discriminate against the caller and not invoke its method (e.g. one reason might be that the caller is a competitive agent).
- *Applets* – Agents are more flexible than applets since applets may migrate only once (from source to destination browser) and can only communicate with the source server.
- *Daemons and control systems* – Agents are flexible to system changes and exhibit proactive (and sometimes) sociable behavior. Daemons and control systems do not exhibit this behavior.
- *Web Robots and Spiders* – Agents can communicate and cooperate with other agents – often cross platforms - by way of standard agent-message wrappers (e.g. KQML) whereas robots and spiders cannot. Furthermore, unlike agent servers, HTTP servers cannot host executing scripts or programs. Therefore, robots and spiders could not migrate even if they wanted to.
- *Push Technology* – Agents are conceptually higher than push. Push technology is a novel data transport mechanism in which the *server* initiates the data transfer to the client – typically, the *client* initiates such a transfer.

To summarize with a typology and taxonomy of software agents we incorporate useful figures from [BRA97]. Figure 2.1a is a typology of agents based on their abilities to cooperate, learn and the autonomy. Figure 2.1b is an agent taxonomy that makes it clear that this field is a diverse but rooted research area.

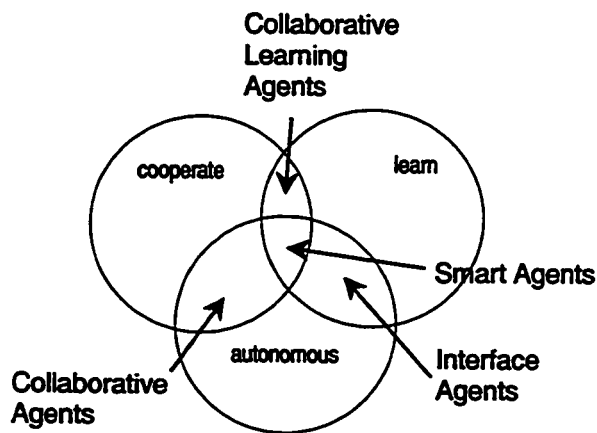


Figure 2.1a). An agent typology based on attribute dimension [BRA97]

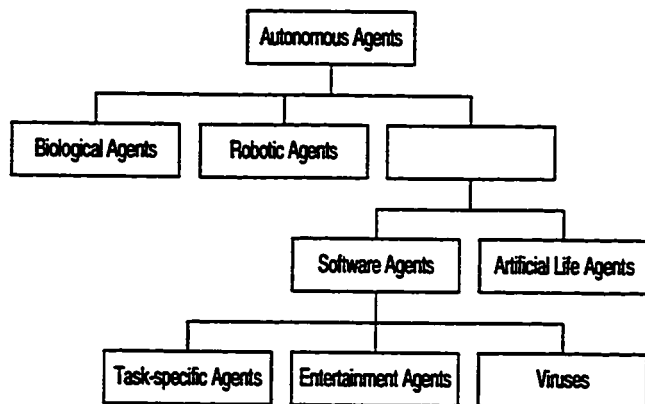


Figure 2.1b). An agent taxonomy [BRA97].

2.2.2 Potential Advantages of the Mobile Agent Approach

Mobile agents do not require network connectivity with remote services in order to interact with them and network connections are used for one-time transmissions of data (the agent and possibly its *state* and *cargo* [FOR97]) and then closed. Results in the form of data do not necessarily return to the user using the same communications trajectory, if indeed results are expected at the node. Alternatively, the agent may send itself to another intermediate node and take its partial results with it, eventually delivering results back to the user (whose address the agent knows). Potential advantages of the mobile agent-model are as follows: It uses less bandwidth by filtering out irrelevant data (based on user profiles and preferences) at the *remote site before* the data is sent back. Ongoing processing does not require ongoing connectivity with the remote server. It saves computing cycles at the *user's* computer (since most of the computing occurs on remote computers). It is more efficient than C-S since the processing occurs closer to the data. It frees the user to log out or migrate since the agent's life is independent of the user's session. Furthermore, the dynamic nature of mobile agents implies that

if an information server is down or loaded, the agent may migrate to one that is up, or more lightly loaded. General discussions of mobile agents, their theories, applications, and promise can be found in [JEN95, MAG97, FRA97, SHO97, JEN98, PHA98].

2.2.3 Potential Disadvantages of the Mobile Agent Approach

Mobile agents are not a panacea for information retrieval, or, for that matter, any other agent-suitable application. As this thesis will illustrate, agents require additional protocols and overhead that may or may not be cost-effective. Based on the experiences of this research, we can say that some of our implementation-specific disadvantages of the agent approach are:

- All participating hosts require execution environments for agents
- If the agent-paradigm is exposed to users a learning curve must be scaled
- With few large-scale implementations it is unclear how agent systems will scale

In [WOO98], Wooldrige and Jennings provide a comprehensive list of caveats for agent development. Selected disadvantages/caveats include (paraphrased from [WOO98]):

- Creating applications for the technology (agents) and not the other way around.
- Ignoring software engineering processes results in bad design and limited extensibility
- Forgetting to exploit related technologies can “sandbox” agents
- Overburdening the agent system with experimental AI techniques makes it cumbersome
- Too many agents are a computational burden, while a system with too few agents does not exploit the advantages of multi-agent system.

Further discussion and debate of such topics may be found in [CHE94, CHE95, JEN95, FRA97].

2.3 Related Work

Due to the rapid pace of agent milestones and the novelty of the paradigm, a relatively wide range of systems, platforms, tools and languages apply in some way to the contributions of this thesis. This section discusses the most important of those.

2.3.1 Agent Systems, Architectures, and Java-based Platforms

[MAE94] was one of the first documented interface-based learning agents. In this project a static agent observes and receives feedback from the user in order to make decisions about and

perform E-mail, calendar and news filtering operations. The basic quantifiers *competence* (how reliable is the agent's knowledge?) and *trust* (how comfortable is the user in giving the agent a particular task?) are recognized as important features. By interpreting the user's feedback, the agent can become increasingly helpful and competent. There are four basic ways an agent can learn: by watching the user, by user-feedback, by examining examples given by the user, and by exchanging information with other agents. Anthropomorphic characters are used to convey the state of the agent to the user. For example, the agents graphical icon has a furrowed brow when it is "working", a smile when it is "gratified", and a mouth agape when it is "surprised". As it is an interface agent, it does not move from the user's machine so no insights into mobility issues can be gained from this project. However, it serves as a valuable early benchmark for future interface agents and remains a widely referenced system.

The Telescript [WHI94] programming language was created by General Magic Inc. to run on the Magic Cap O.S. (for hand-held devices). The language directly supports mobile agent activities, including the *meet* command that transports an agent and initiates a dialogue with another. Telescript is an object-oriented language designed to support commercial transactions by buyer and seller agents. Essential to this are: *places* (computer hosts that offer services), *agents* (represent service providers or users), *travel* (allows agents to move arbitrarily on the network), *meetings* (allow two agents to meet and communicate), and *permits* (describe authorities). General Magic has recently introduced products called *Tabriz*, which, among other things, allows Internet-based access to Telescript agents, and *Odyssey* [ODY98], a Java-based mobile agent-programming platform. The Tacoma system [TAC95] is another such O.S.-based mobile agent-system that uses analogous abstractions such as *briefcases* and *folders*. The basic Telescript model and entities are still used today by many agent systems although the language itself is not widely used on PC's. The model does not have a sophisticated inter-agent communication language or a graphical environment in which users may create mobile agents. The emergence of Java [SUN94] has made early General Magic systems almost obsolete.

The AgentTCL [RUS97] project aimed to reduce agent-migration to a single Tool Command Language (TCL) [OUS94] command, provide communication among agents, provide effective security, support multiple platforms, and be in the public domain. To achieve these goals the

TCL interpreter had to be extended with a more sophisticated module that captured the internal state of a TCL script executing within it. The state information - consisting of a global variable table, the names and bodies of procedures, call frames for active procedures and a command stack - must be “packaged” by the interpreter, sent along with the rest of the agent script, and re-installed at the destination host. AgentTCL represents an important contribution since it implements a complex agent-migration feature on a relatively simple language. AgentTCL agents do not yet have an extensive set of communication primitives nor a Java implementation though that is a future research agenda. The developers claim that they are developing a GUI for agent visualization.

Java-based agent programming platforms are emerging quickly with the August 1997 release of SUN's JDK1.1 [SUN94]. Examples are Mitsubishi's Concordia [MIT98], IBM's Aglets Framework [IBM97], and ObjectSpace's Voyager [VOY97]. These platforms all provide support for mobile agent activity and collaboration in a networked computing environment. They all also provide an agent server, JDK1.1 compliance, and easy object migration and class loading - furthermore, they are all available for the PC environment [KIN97]. These platforms differ mainly in the subtlety of the Java interfaces they offer and the extent to which agents may collaborate and be managed by the system.

The Knowledgeable Agent-oriented System (KAoS) architecture for agents [BRA97b] enables the exchange and management of technical information and services in an aerospace application domain. Software agents run on top of this proposed distributed architecture. The goal is to provide a standard set of communication primitives in a transport-neutral fashion. Agents communicate with each other within only with their own “domains”. To do so between domains, agents must message with a *Proxy Agent* who passes the messages on to the recipient. To interact with physical or external entities, agents utilize *Mediation Agents*. Finally, a *Service Broker Agent* receives *advertisements* of services and matches agent queries to recipients. KAoS agents use KQML-like speech-acts within conversation sequences (e.g. policies) that are *hard-wired*, as opposed to *emergent*. Conversation policies implement multi-state interactions between agents such as *inform*, *offer*, and *conversation for action*. The KAoS platform is solid due to its extensibility and flexible conversation policies. Overall KAoS is more powerful than

KQML and its conversation policies on par with *AgentSys*. In this thesis however, we propose more intuitive and useful policies that enable the exchange of profiles, media, and authentication data – granted, some of these are tailored specifically to the application we have in mind. KAOs does not propose the structure of agent execution environments nor does it propose a graphical management or agent inception system.

The InfoSleuth system [NAD97, JAC96] is a Java-based architecture for agent-based retrieval of information on heterogeneous information resources. Main novel technologies and contributions in InfoSleuth are (1) the enterprise model to capture and map to database schemas, (2) knowledge mining and representation techniques, and (3) distributed processing techniques. A data meta-model implementation uses *frames*, *meta-models* and *ontologies* (from top to bottom resp.) to precisely organize descriptions of resources. Special purpose agents such as *monitors*, *brokers*, *KQML agents*, *resource agents*, and *ontology servers* are used in a variety of ways to satisfy user-queries. KQML and KIF are the basic wrappers used for messages and knowledge. Unlike *AgentSys* however, InfoSleuth does not focus on application-level transfer of agents or address physical or graphical management of agents.

The SIMS project [KNO97] proposes the organization of multiple information agents, how each one should structure its knowledge, the agent communication protocol, the query processing module, and the agent-learning techniques. Information agents form an interconnected network. Each agent in the network has specific knowledge about the domain that it resides in or is a information repository. An information goal is expressed in terms of the agents domain model and is then reformulated for other information agents into their domain-level terms. Queries between agents are expressed in LOOM. The SIMS platform is a robust application-independent platform whose main contributions are the sophisticated agent-network and domain models. It would be an interesting exercise to map *AgentTask*, the highest *AgentSys* protocol level, onto a SIM agent implementation.

2.3.2 Agent Communication

In order for agents to exchange data or use services of any kind, they must somehow communicate with each other. In a client-server implementation such as RPC, communications

typically consist of a named procedure as well as the parameters of operation. For more complex entities such as agents to communicate, more sophisticated protocols are required. The Knowledge Query and Manipulation Language (KQML) [FIN97], developed by the ARPA Knowledge Sharing Effort, is a language and set of protocols that allow programs to interact with other programs. It offers flexibility through communication-act content-language independence. Using balanced parentheses a KQML message consists of: (i) a performative that identifies the kind of message, (ii) various other parameters which describe the content (e.g. recipient-id), and (iii) the content of the message. KQML can be used effectively as a speech-act language for agent-systems [JAC96], so long as the agent community agrees on the sub-set of performatives (e.g. *ask-one*, *stream*, etc.) to adopt. The Knowledge Interchange Format (KIF) is often used in conjunction with KQML to code the content of declarative (e.g. PROLOG) messages. KIF is a prefix version of the language of first order predicate calculus. Using KIF terms, one can build sentences, rules, and definitions; for example, (`> (sin theta) (cos phi)`). [GEN97]. A related formalism called an ontology is required for multi-domain agents. An ontology is an *explicit specification of a conceptualization* and is the vocabulary with which queries and assertions are exchanged [GRU93]. *Ontolingua* [GRU93], a language for creating ontologies, can be used to describe bibliographic data (for example) in a fashion similar to the object-oriented design of programs. Ontolingua allows ontologies to be compatible with multiple representation languages. By translating commands written in KIF to various other forms, agents may be able to access a wide range of different types knowledge bases.

The KQML exchange format has gained considerable momentum as an agent communication language. This is due to both its flexibility and power as well as to a large number of KQML-based agent implementations, platforms and prototypes that prove its utility. Some examples follow. [GEN97] describes an agent framework on which KQML-based agent applications may be built. [SYC96] describes RETSINA, a collection of cooperating, distributed agents that inter-communicate via KQML. [BRA97b] describes KAoS, an architecture upon which agents manage aerospace information. The inter-agent communication language in KAoS is based, in spirit, on KQML though semantic changes have been made. Finally, [NAD97] describes the InfoSleuth System in which, again, the spirit of KQML has been used and slightly modified to ensure a more effective application (see also [HOO98]).

In [PIT95] a community of agents communicates in order to arrange for a review of an organizational document. Agents represent potential reviewers and a negotiation procedure occurs as each agent has its own agenda. Message passing between agents is based on speech-act theory - logic is used to express the protocol specifications for the three most basic specifications: *inform*, *query*, and *command*. Other protocols required for the specific organizational examples include *propose*, *offer*, *request*, *accept*, *inform*, and *cancel*. Each protocol lends itself to a finite state machine representation so that the space of possible messages and responses to each is completely determined by the protocol's logical specification. In [TAK95] agents are applied in a travel-plan application. This work studies the role of multiple ontologies and sharing of ontologies. When agents communicate, their messages are forwarded to mediators called *facilitators*. The facilitator determines an appropriate recipient of the message by examining a server that logs the complete ontology. When multiple ontologies are used, *translator* agents must analyze incoming messages and retrieve conversion formulae from the server, compose a new message in the out-going ontology, and then continue to wait for further requests. In [LUK97] an ontology-based Web agent is presented that shops for and compares prices of products. These systems illustrate the importance of agent communications but are not mobile agent implementations. [LUK97] is a spider-like agent that does not physically migrate and has no formal inter-agent communication – rather, it communicates with Web servers via HTTP.

It is very often the case that mobile agents interact indirectly with other mobile agents or resources through a *facilitator agent*, *broker*, or *information agent* (e.g. [GEN97, KNO97, BRA97b]). This is also the case in this thesis. [GEN97] provides one of the most rigorous definitions of such an agent, stating that in order to be one the agent must offer services that include: white pages, yellow pages, direct communication, content-based routing, translation, problem decomposition, and monitoring. Facilitator agents in this thesis have *environmental* roles and *mobile agent-based* roles; they monitor and log the environment as well as being the source of meta-information about media, agents, servers, etc.

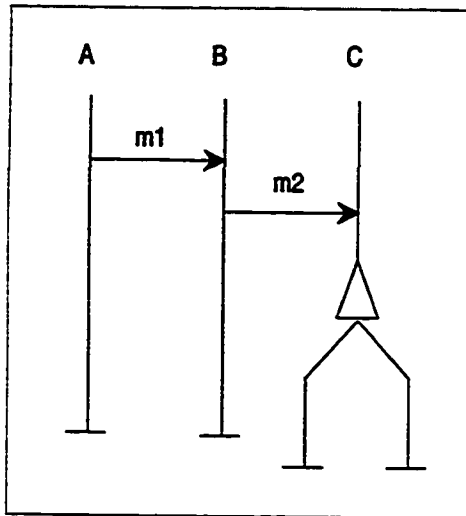


Figure 2.2a). A simple Interaction Diagram (ID). A, B, C are agents; m1, m2 messages. Agent C's computational flow is *split* into two.

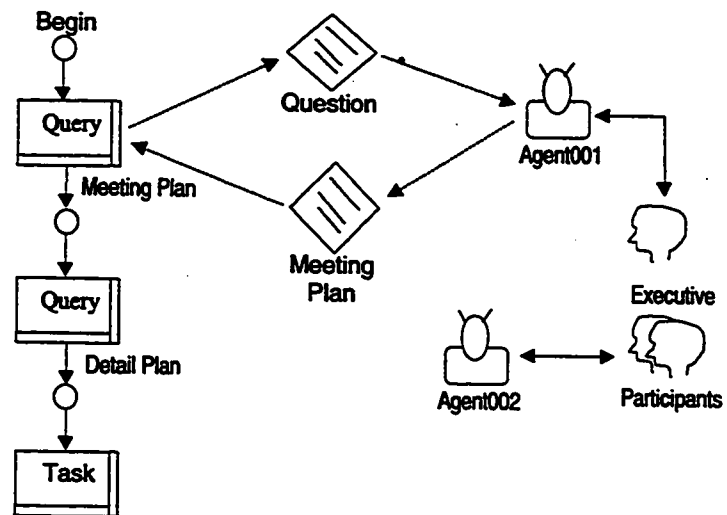


Figure 2.2b). A portion of an agent model graph from the Agent Prototype tool. The graph represents an agent workflow and can accommodate queries and physical actions such as speaking to the user.

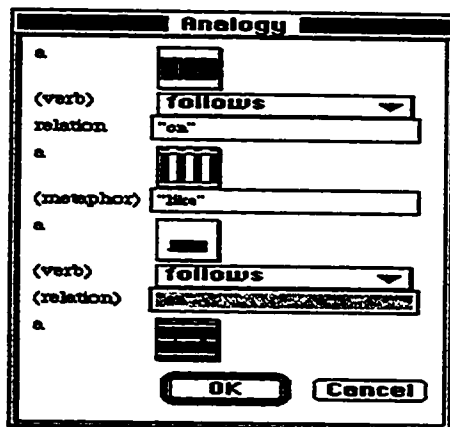


Figure 2.2c). Creating behavior for a train in Agentsheets using the behavior of a car as an analogy.

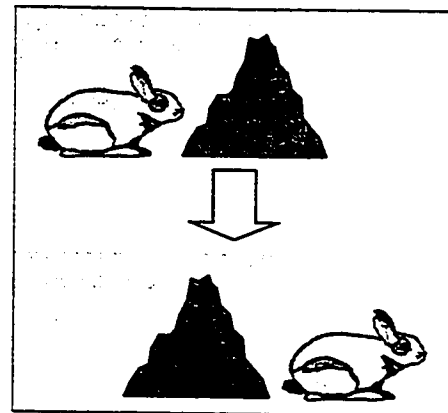


Figure 2.2d). KidSim *before-and-after* rule. If conditions are encountered matching the *before* rule (e.g. blocked) it makes the appropriate transformation (e.g. jump over obstacle).

2.3.3 Mobile Agent Behavior Specification

Specifying the behavior of a mobile agent is similar to, but more difficult than, writing a program. Many aspects of the location and interactions of a mobile agent are not known until run-time – the fact that location is an issue to consider distinguishes building agents from building static programs. Furthermore, mobile agents should be particularly accessible to novice users who should not need to write any sort of programming code at all. This implies an emphasis on visual, graphical user-mobile agent interaction.

The important *Iconic Modeling Tool* (IMT) component of the *Agent Inception System* (AIS) addresses weaknesses in other existing tools. [RON97] introduce a timing-like Interaction Diagram (ID) on which the messaging aspects of agents can be visualized (figure 2.2a)). The diagrams represent agent messaging and processing within a multi-agent system. The graphical notation is based on six fragments – *begin*, *send*, *split*, *end*, *receive*, and *join* – each with a simple visual representation.

ID fragments may be combined to create semantics that are more complex. *Computational flow fragmentation* lists fragments with respect to agent-order, whereas *message flow fragmentation* lists with respect to messages or tasks. ID's can be used for the analysis of computational sequences and communication and can even reveal the presence of anomalies such as race-conditions. The practical timing-diagram paradigm used by ID's is also used by the IMT. Since ID's only model messaging, however, they could not model more complex mobile agent scenarios. For instance, new types of both arrows and fragments would be required to distinguish between the following subtly different message-types: local agent-to-agent, remote agent-to-agent, remote cargo-to-agent, remote agent-to-Web server, remote agent-to-local-user, and so on. Furthermore, mobile agents can implement other complicated scenarios that could not easily be modeled using ID's - such as the establishment of remote monitor-agents that wait for trigger conditions before firing responses.

[TAK96] tackles the problem of visualizing workflow using an icon set (figure 2.2b). An Agent Prototype (AP) tool can be used to create agents that, for example, set up meetings between their human users. First an outline or state diagram is created; this diagram models basic user

interactions. Then a screen layout is generated; each state in the state diagram has a screen layout consisting of rectangular fields on the screen in their relative positions. Some field data is supplied at this point. Finally, actions are associated with each state in the state diagram. Actions are ordered for execution using Petri-nets and the result is a working agent; anthropomorphic characters are used in the interface. Agent model diagrams represent non-automatable actions (e.g. human intervention) as queries. The AP uses about 25 generic icons to represent concepts such as *resources*, *agents*, *users*, and conversational entities such as *querying*, *telling*, and *asking*. Other icons exist for *playing* animations or *speaking*. Graphical models resemble directed graphs with icons and text at each node. It is apparent, however, that the IMT is more effective than the AP since it allows visualization of agent mobility, cargo mobility, messaging, monitors, purchases, and other complex aspects of mobile agents.

Agentsheets [REP95] is a rule-based end-user programming environment that is actually usable by novice users (e.g. children). It employs tactile programming in which program components have both visual and interactive interfaces (figure 2.2c). Agentsheets can be used to create SimCity-like worlds, games, and cellular automata. To program agents, one defines the *conditions* (i.e. perceivable situations) and the desired *actions* (i.e. new states). This can be done visually in Agentsheets since simple graphical elements represent, for example, the “if-then” construct. Language components (condition-list, rule, action-list) can actually be dragged onto agents, thereby equipping the agent with the desired behavior. All levels of an Agentsheets program, including the simulation, agents, rules, and primitives, may be shared amongst programmers via the Web. Agents in Agentsheets may speak, play sounds, or open URL’s but are not mobile or distributed by nature and thus the modeling environment does not address many of the challenges that the IMT faced. However, the visual environment is strong and could conceivably be integrated with concepts from our IMT to create a very effective mobile agent programming environment.

Finally, Smith et al describe KidSim [SMI97], a language-less programming environment in which dynamic graphical rules govern the behavior of characters in simulations. A KidSim simulation session includes a clock, simulation objects (agents), and a rule editor. Its typical users are children. Graphical rewrite rules and programming by demonstration are used to teach

agents. A simple example is a before-and-after rule in which the child draws a “before scenario” which, if matched during simulation, triggers a transformation that results in the “after scenario” (figure 2.2d). KidSim is a very powerful and flexible tool for simulations and games, however, it is unlikely that the “programming by demonstration” paradigm could be as effective within the dynamic and complex world of distributed information. This is especially true because agent actions are both more abstract (e.g. helping another agent) and complex (e.g. negotiating) than jumping over obstacles. [DEC97] introduces an architecture for *information agents* that accept poorly specified goals, utilize reusable behaviors, and includes message polling, service advertising and self-cloning. However, the visual editor for this architecture is not specified. The Specification and Description Language (SDL) [SDL92] and Use Case maps [BUH97] are general systems design tools which may have a role to play in specifying *some* dimensions of agent behavior. However, while these tools can model complex scenarios, they lack a “friendly” look-and-feel. Finally, [BIM95] and [FAL95] present visual specifications for temporal logic and documents (respectively). Some of these techniques may be useful in the mobile agent context where agent behavior, like asynchronous document behavior, is not known until run-time. In any case, they will need extended semantics within the mobile agent paradigm.

Overall, the IMT is a response to still-open questions regarding the creation and management of mobile agents and their itineraries, and fills an enormous need for a simple and intuitive end-user tool for Itinerary modeling within the mobile agent paradigm. There has not yet been a tool proposed that uses computer graphics to intuitively and effectively address the challenges of mobile software agent behavior specification.

2.3.4 Comparison at a Glance

This section provides a tabular representation of some related works. Some of these have been discussed in detail already while others have not. The table headings are an illustrative set rather than an exhaustive set. Table 2.1a) and 2.1b) illustrate a comparison. Through this table, it can be seen that *AgentSys* and its components make specific inroads into agent-oriented research. Very few other research projects propose a top-to-bottom solution and implementation of an agent platform.

AgentSys (AEE, AIS, protocols..) [FAL98a, FAL98b, FAL98c]	Yes - limited. Specific security solutions are not proposed in this thesis.	Yes - agents migrate. Even components of AEE and AIS may migrate if desired.	Yes - via the AIS Web Access Mgr. Users can do quick-searches. Arbitrary Web servers can be 'spidered'. Yahoo™ and Excite™ are "wrapped" in Java.	Yes - AgenTransact. The inter-agent syntax is based on KQML, yet adds new performatives and functionality.	Yes - The world is carved up into manageable pieces. Ontological terms are inserted into speech-acts or sent across network. Extensible. Flexible.
Agent TCL [RUS97]	Yes - limited sandbox model.	Yes.	Not specified.	Not specified but multiple interpreters may be used. Mainly TCL-based.	No.
COOL [BAR95]	No.	N/a	N/a	Yes - theoretically-based language	Yes - integrated.
RETSINA [SYC96]	Not specified.	No - not presently.	Yes.	Yes.	Yes.
SHOE [LUK97]	No.	No.	Yes.	No.	Yes.
InfoSleuth [NAD97, JAC96]	No.	No.	Yes - possible.	Yes.	Yes.
SIMS agent [KNO97]	No.	No - not specified.	Not specified.	Yes. KQML, LOOM.	Yes.
Agentsheets [REP95]	N/a	No.	Not specified but probably possible.	Not specified.	No.
Agent Prototyper [TAK96]	N/a	No - models message flow only.	No - but could conceivably have such integration.	Yes - limited. Models the transfer of queries, etc.	No.
KidSim [SMI97]	N/a	No - limited distributed task modeling	Yes - limited.	No - rule-based behavior only.	No.
Use-Case Maps [BUH97]	N/a	Yes - limited but can be used to model distributed tasks.	N/a	Yes - can model agent comm.	No.
SDL [SDL92]	N/a	Yes - limited.	N/a	N/a	N/a
Other Agent Middlewares [VOY97, ODY97]	Yes - varies between systems.	Yes.	Yes - varies.	Yes - varies.	No.
Web-Spiders [CHE96]	No - not generally needed.	No.	Yes.	No.	Sometimes.
Interface Agents [MAE94]	No - generally not an issue since mobile code not involved.	No.	Yes - many interface agents filter Web data and/or e-mail.	Sometimes.	No.
Java Applets [SUN98]	Yes - the "sandbox" approach limits potential "damage".	Yes - but limited. Single migration only.	Yes.	Yes - limited. Communicates with Web-browsers, and the originating host.	No - application specific implementations.

Table 2.1a) Comparison between agent systems, tools, or components and AgentSys components.

AgentSys (AEE, AIS, protocols...) [FAL98a, FAL98b, FAL98c]	Yes – via <i>Status Mgr</i> , <i>LiveAgent Mgr</i> , AIS	Yes – <i>Iconic Modeling Tool</i> provides a simple, powerful drag-and-drop GUI for agent creation.	Yes – <i>Cargo Agent</i> at AEE's <i>Media Mgr</i> at AIS. Cargo options in <i>AgentTransfer</i> spec.	Yes – via the <i>Status Mgr</i> . Queries and cmds may be sent to remote agents.
Agent TCL [RUS97]	No – but is a future research area.	No – but is a future research area.	Not specified.	Not specified.
COOL [BAR95]	No.	No.	No.	No.
RETSINA [SYC96]	Not specified.	No.	Yes – but limited since agents do not migrate.	Yes – limited.
SHOE [LUK97]	N/a	Yes.	Yes.	Yes.
InfoSleuth [NAD97]	Yes – limited interface.	No.	Yes.	No.
SIMS agent [KNO97]	No.	No.	No.	Not specified.
Agentsheets [REP95]	Yes.	Yes – <i>Agent Builder</i>	No.	Yes.
Agent Prototyper [TAK96]	Yes.	Yes.	Yes – limited. Models query and data transfer.	Yes.
KidSim [SMI97]	Yes.	Yes.	N/a	N/a
Use-Case Maps [BUH97]	No – not specified.	Yes – available, but not for agent-specific applications or issues.	N/a	N/a
SDL [SDL92]	No – not specified.	Yes – available, but not for agent-specific applications or issues.	N/a	N/a
Other Agent Middleware	No – not specified.	No.	Yes.	Yes.
Web-Spiders	Yes.	No.	Yes.	No.
Interface Agents	No.	Yes.	Yes.	Yes.
Java Applets	No.	Yes.	No. Not Specified.	No. Not specified.

Table 2.1b) Continues table 2.1a)

2.4 Agent Issues not Addressed in this Thesis

While this thesis proposes application-level protocols to enable a mobile agent platform for information retrieval, it does not attempt to propose new solutions for *all* of the required “sister” technologies required for the long-term success of such a platform. This section outlines some of the agent issues in research that are not a part of the contributions of this thesis. In some cases, in fact, this thesis assumes the presence of such sister technologies (e.g. authentication, digital signatures, distributed object brokers, etc.).

2.4.1 Security

Since Java [SUN94] is the most prevalent language for agent-implementations it is worth while to summarize Java and its security limitations. Java is truly an enabling technology for agents, enabling class mobility, network communications, and security features. This is not to mention

the write-once-run-anywhere pledge. Java source is compiled into bytecodes and executed by Java Virtual Machines (VM) which are register oriented machines easily implemented in either hardware or software. Java counters security threats in a number of ways including by eliminating pointers and performing compile-time checking of types, casting, access, and over/underflows. Furthermore, custom *Class Loaders* can sandbox bytecodes to eliminate threats. This technique, combined with name-spaces, makes it impossible for bytecode to spoof built-in classes.

Java also offers a Security Manager that can raise exceptions when requested operations are not pre-authorized. Java security is a continually evolving technology and one should refer to [SUN98] for the latest information.

In general, mobile agents and hosts may attack each other in several ways including [GRE98]:

- *Damage* – e.g. files on the hosts systems are damaged, or the agent is damaged by the host
- *Denial of service* – e.g. the host refuses to return the agent
- *Breach of privacy or theft* – e.g. undesired access to sensitive information
- *Harassment* – e.g. repeated attacks by an agent on a host
- *Complex attacks* – e.g. Trojan horse-type attack, etc.

Combating these attacks requires the implementation of a security technique. [GRE98] and [PHA98] provide comprehensive descriptions of these techniques that we only summarize here (from [GRE98]). Hosts can be protected from malicious agents in the following ways: (1) *Authentication credentials*: In this scheme agents are digitally signed to validate their sender, their transit path, etc. This does not guarantee, however, the harmless execution of this agent. (2) *Access-level monitoring and control*: A reference monitor grants permissions to agents based upon their authenticating credentials. The philosophy here is that mobile agents that cannot do anything harmful cannot cause harm. (3) *Code verification*: This process involves finding any illegal instructions within an agents code. (4) *Limitation techniques*: Limiting an agents time, range, or duplication limits. Protecting a mobile agent against its environment is a subtly different issue and can be accomplished via fault-tolerance and encryption-based techniques.

The Aglet Security Model (ASM) [KAR97], designed for the Aglets Workbench [IBM97], provides protection of the host aglets and the protection of other aglets. The ASM uses a policy database wherein principals and privileges are combined into rules, and owner-specified preferences that allow aglet owners to establish security preferences that must be honored by remote contexts. Further discussion of mobile agent security can be found in [DEA97, GRE98, and PHA98].

This thesis does not propose new alternatives to protecting agents from hosts or visa versa. While the *AgentSys* protocols enable the exchange of privileges and authentication they do not propose the interpretation of such entities. Such interpretation and computation could be a future research sub-program of *AgentSys*.

2.4.2 Artificial Intelligence

The Artificial Intelligence (AI) community has provided a wealth of research and innovation for many years. AI has tended to isolate its important components rather than work on grand unifying concepts, thus the sub-fields of planning, reasoning, learning, computer-vision, etc. have developed in relative isolation. In the early 1980's, however, intelligent agents and their enabling concepts began to garner more attention [JEN98] and had a mild unifying effect.

A significant milestone in AI as it pertains to software agents came when Brooks proposed the *Subsumption Architecture* (SA) (see [BRO91]). With this architecture Brooks was supposing that trying to build "human level" intelligence into systems from scratch was not as wise as first creating simpler behaviors and building up to more complex ones. The SA was designed to provide the functionality displayed by lower level life forms, including insects. Through a combinations of simple machines with no central control, no shared representation, slow switching rates, and low bandwidth, more complex behaviors could be built. SA's are highly reactive and operate better then planning-oriented systems do in dynamic environments. To implement an SA, augmented *Finite State Machines* are used, adding both time and inhibition and suppression between modules, as well as distributed control. The SA paradigm could be used to build intelligent software agents though Brooks has mainly focused on physical mobile robots. Distribute AI (DAI) is a more complex sister technology of AI. In DAI knowledge

bases, agents, and information can be distributed on an information network. Thus DAI combines the theories of AI with the complexities of distributed information and computing. The RETSINA platform [SYC96] is an example of such a system where agents and information sources are distributed on an *infosphere*, and interchange knowledge using the KQML message format.

Differing from SA architecture is *Practical Reasoning* (PR) architecture. Such architecture is based upon the kind of pragmatic reasoning that humans do (see [JEN98, MAG97]). Human behavior can be thought of as emerging from the interaction between various attitudes like beliefs, desires, intentions, etc. Agents using the belief-desires-intention (BDI) practical reasoning model have beliefs, desires, and intentions. Beliefs correspond to information about the environment, desires to options or potential actions, and intents are committed desires. Beliefs are continuously updated while desires are filtered to determine new intentions.

Agents implemented in this thesis do not use symbolic logic or a declarative programming language (e.g. Prolog) but rather are implemented procedurally using Java. However, nothing in the *AgentSys* protocol suite prevents the use of declarative programming or AI techniques. [GEN97] notes that for the declarative approach to be effective, the statements that are exchanged must originate from a sufficiently expressive, yet compact, language. It is also noted that both procedural and declarative agent implementations have their advantages and disadvantages. Procedural implementations can represent delayed or persistent goals and execute directly and efficiently. However, they are unidirectional and difficult to merge. Declarative implementations must use a compact language to be efficient but leverage the power of declarative statements of a wide variety [GEN97].

2.4.3 Object Serialization

Object serialization refers to the conversion of an object and its state into a byte stream in such a way that the stream can later be converted back into a copy of the object. The conversion back to the object is called *deserialization* and can occur either on the same virtual machine (e.g. system) or on another [SUN98]. When an object is serialized its class is identified but not serialized along with it. Rather, the system that deserializes is responsible for locating and

loading the parent class (e.g. via a remote JAR file). Sun's Java initiative describes an *Object Serialization Specification* [SUN98].

Serialization of objects is a low-level technical research program on its own and a topic into which this thesis does not delve. With the adoption of the Voyager platform [VOY97] this thesis is spared the need of proposing an agent-specific serialization scheme. While agents in this thesis are objects and *do* migrate, Voyager middle-ware implements a custom serialization of objects, exposing only the *moveTo()* method to the programmer. Invoking an agent's *moveTo* method, for example *moveTo(123.43.22.1,atAEE)*, initiates the following sequence of events: interrupts the agents thread of control, serializes the object and its serializable instance variables, ships the object to the named IP address, deserializes it at the receiving end, restores the instance variable values, and executes the named method *atAEE()*.

2.4.4 Distributed Object Technology and Platforms

The Common Object Request Broker Architecture (CORBA) [COR95] and, for example, the DCOM [DCO98] architecture, are key technologies for future mobile agent systems. In general, these architectures provide [MAG97]:

- Component-based service realization and reuse
- Arbitrary distribution of components
- Location transparent communication
- Interoperability between systems

CORBA objects are defined in an Interface Definition Language (IDL) which specifies their exposed interfaces – IDL's are also manifested as objects. The standard IDL then, is the window to access object services or common facilities, and effectively separates the interface from the implementation of the object. When a client object wishes to invoke a remote objects method the invocation is caught at run-time by the IDL stub. The request is sent to a remote object adapter stub over the interconnection network (TCP/IP, etc.) and the method is invoked. CORBA effectively flattens a distributed object system from the programmers point of view and enables interoperability between object systems. A CORBA implementation is more flexible than an RPC-based one. Object mobility is not directly supported in CORBA.

The OMG recently began a push towards a *Mobile Agent System Interoperability Facility* (MASIF)(formerly the *Mobile Agent Facility* [CHA97])[BRE98] in order to bind the attractive advantages of the mobile agent paradigm with distributed object systems. Grasshopper [GRA98] is the first such MASIF-conformant intelligent agent platform. It enables [BRE98]:

- Creation of autonomous moveable agents with a few lines of code
- Transparent location of agents
- The creation of environments that are open to any other MASIF or CORBA compliant ones

The end-result of these features is the ability to create electronic marketplaces and Intelligent Networks (IN). A Grasshopper agency consists of services including support for (i) Agent execution, (ii) management, (iii) security, (iv) unique identification, (v) transaction, and (vi) communication. Further information on the impact of mobile agent technology on communications systems evolution can be found in [BRE98].

This thesis recognizes the emerging acceptance of CORBA and Grasshopper and neither specifies nor rules out the usage of such distributed object architectures. Since the *AgentSys* protocols reside on the application layer, such architecture could be used to add services and functionality. In other words, the platform proposed in this thesis could theoretically sit on top of CORBA or Grasshopper. It should be noted that to some extent the Voyager platform provides analogous flexibility to the agents that we create in this thesis. Such agents can be messaged with transparently to their location (by using forwarders) and methods or remote objects can easily be invoked as if local (using an RMI-like paradigm). Voyager objects, and hence agents implemented in this thesis, can also communicate with standard CORBA-compliant ORB's.

2.5 Summary

Despite a broad range of agent and mobile agent tools and protocols in the research community there has been little agreement on the ones which will compose the basis for world-wide agent sustenance. Multiple proposals are under scrutiny for standardization (e.g. KQML and FIPA [FIP98]) and competing companies all want their mark on standards (e.g. IBM, General Magic, Microsoft). Regardless, computer end-users seem to have a general lack of understanding of the mobile agent paradigm and its power. What is required is an end-to-end top-to-bottom system

that will display the power and elegance of this paradigm to end-users and thus create a market for the next generation of agent tools and standards. This thesis establishes the platform, protocols, and basic tools for creating mobile agent applications.

Chapter 3

Communications – Inter-Agent and Inter-AEE Protocols

3.1 Introduction

A mobile agent is like a businessperson travelling from country to country. Not only does it need the correct passports and authorities to enter each country it also needs the intelligence to speak in the *native* language or in some “universal” language to accomplish the mission. In this chapter, we introduce the essential elements required to allow a community of possibly heterogeneous agents to talk about and exchange data and messages, help each other, and migrate from node to node. The main topics are:

1. *AgentTransact* – an inter-agent communication protocol consisting of a standard set of speech-acts, novel conversational modes, and a basic ontological grammar.
2. *AgentTransfer* – an inter-AEE transfer protocol for sending agents, cargo, ontologies, remote messages, and slaves unambiguously between AEE's. *AgentTransfer* is an extension to the IBM ATP/0.1 protocol.

3.2 An Inter-Agent Protocol - *AgenTransact*

In order for agents to communicate, they require ontological, messaging, and conversational protocols. Together, these three classifications form the “layers” that compose *AgenTransact* which in turn allow agents to engage in state-full high-level conversations.

3.2.1 A Shared Data Ontology

Data, agents, networks and the services that they provide are complex and multi-faceted. If the goal is to have an agent system in which mobile agents roam a network where services are offered and transactions made, then it is crucial to have what is referred to as *an ontology*. As defined in [GRU93] an ontology is, “...a common vocabulary in which shared knowledge is represented [that]...associates names of entities...with human-readable text describing what the names are meant to denote, and the formal axioms that constrain the interpretation and well-formed use of these terms.” An ontology is a type of grammar that remove ambiguities between communicating parties – this includes human-agent and agent-agent modes. [KAC98] proposes a taxonomy of ontologies that helps to position this work:

1. *Application* – information structures used to build an application system
2. *Domain* – specific to a particular artifact and generalizes over particular application tasks in that domain
3. *Basic technical* – generalize over particular artifacts and provide a view usually related to a physical process
4. *Generic* – a top-level category used to “carve-up” the world.

The ontology we propose within this thesis fits best into the *application* category. Each individual “branch” of the *AgentSys* ontology will be illustrated as a tree of terms. Each term has parameters just as object-oriented classes have class variables. The “tree of terms” concept is essential to the overall operation of inter-agent messaging since ontological terms are used as parameters of messages, as are their parameters. The ontological commitment is useful because:

- As specified by the *AgenTransact* protocol, high level ontological terms may be replaced by lower, more specialized, ones. For example, “media” may be replaced by either “text” or “video” since “media” is an ancestor of both.
- It allows the system to be used in other application domains so long as a new ontology is defined and adopted.

- It “carves up” the domain into essential conceptual “pieces”.

In this thesis we introduce an application-ontology based upon the four critical entities in an agent-system: *networks*, *services*, *agents*, and *documents*. This scheme offers a clear classification of data and agents and forms the basis of messages and conversations. One of the important elements in this ontology is that it accommodates HTML documents. This is because despite the sophistication of the agent-paradigm, Web services remain an omnipresent component. We also assume the presence of more sophisticated documents called asynchronous (or time-based) multimedia documents (see [FALC95]).

Network level (figure 3.1)

The network that agents travel upon is represented as collection of servers, connections, and network entities. Each server must declare a unique name, IP-address and a service name. Servers are the physical places where agents go and there are several types including web servers serving HTML documents. One special type of server hosts agents in an *Agent Execution Environment (AEE)*. Agents travel via connections (sockets) to which particular ports are associated. Network nodes (computers) have names, IP-addresses, and a platform-type. This ontology liberates agents to ask for particular servers and facilitators and lets them understand the basic network entities. At the same time, it reduces network entities and primitives from the OSI Network layer to simple abstractions.

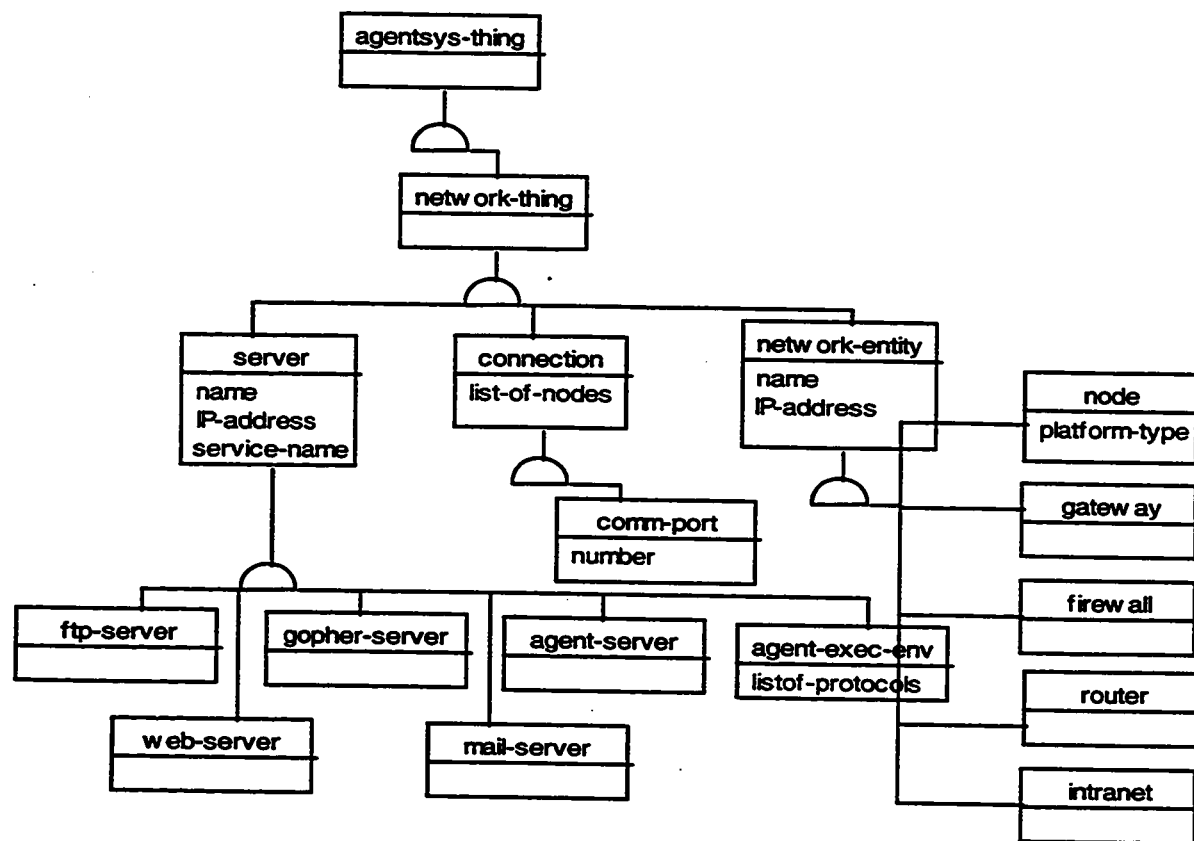


Figure 3.1 Network-level ontology

Agent level (figure 3.2)

The agent-level ontology strictly categorizes and describes the entities related closely to the agent itself. A mobile agent consists of agent-content; a header, an executable script, cargo (e.g. media that it is “carrying”), a task-type, and some untyped (possibly private) part. Furthermore, if the agent is a parent it lists its children and vice-versa. The human users’ identification is also contained in the agent. Since agents engage in conversations, they also encapsulate conversation logs, managers, speech-acts, and the conversation states that make up complete dialogues between agents. Each stage in a conversation, called a *state*, is initiated by a speech-act. Each stage has a unique identification and the meaning of speech-acts is context dependent. The semantic for an entire conversation is called a *protocol*. A unique feature is that agents carry with them a *bag* of protocols representing the different conversation dialogues that they know how to engage in. If two agents that don’t share a protocol wish to communicate, it is possible that one of them will provide it to the other (the *ask_for_protocol* conversation implements this exchange). This is possible because conversations can be represented as finite state diagrams.

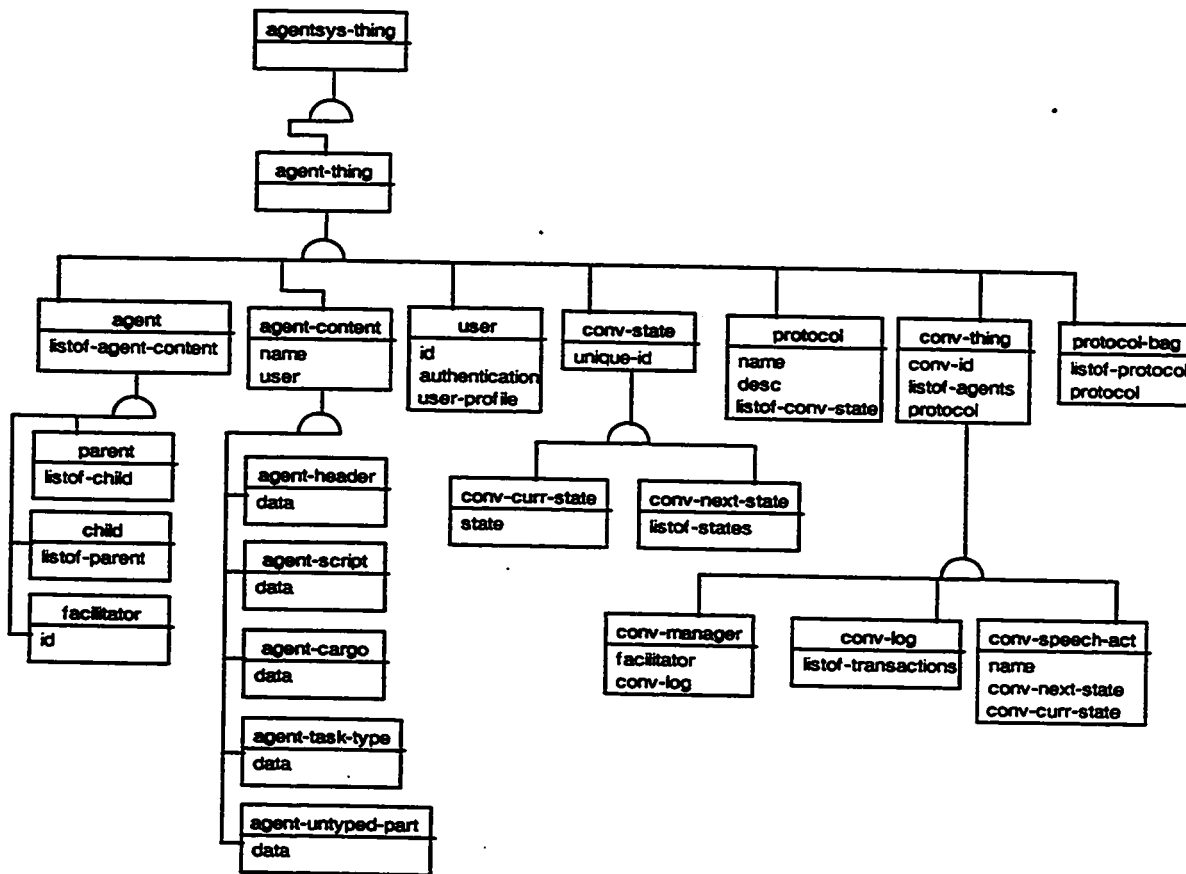


Figure 3.2 Agent-level ontology

Service level (figure 3.3)

Agents roam the network in search of services that can help them satisfy their task description. A service can be one of many different types: search-engine, data-retrieval, meeting-room, agent-execution environment, etc. and each service has a unique identifier. A resource is similar to a service but more closely linked to the physical media. For instance, there are several types of resources such as *document*, *media*, and *catalog* and one *service* may offer one or more of these resources. Document resources serve compound documents to agents, media resources serve binary or mono-media, and catalog resources provide meta-indices of other resources as well as pointers and references.

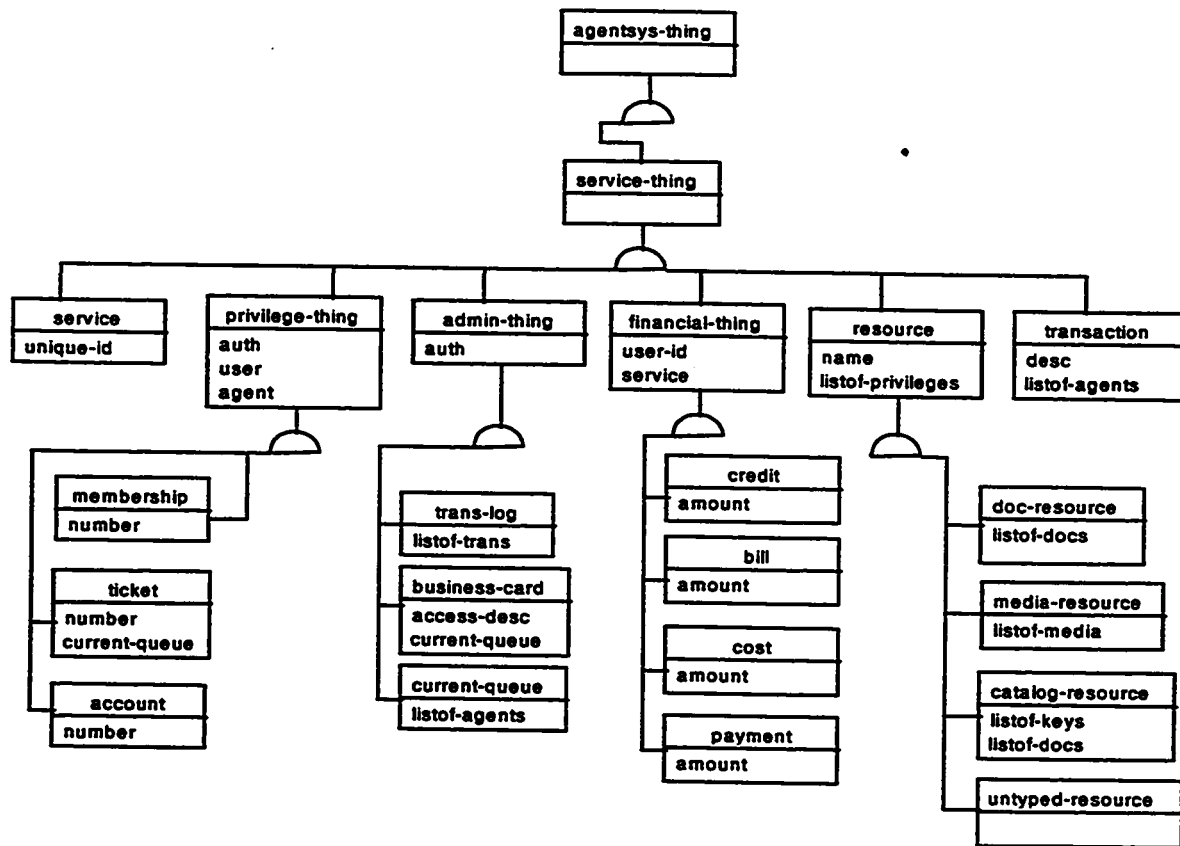


Figure 3.3 Service-level ontology

Closely related to services are privileges, administration, and finance. Agents can access services only if they have or are granted privilege. Minimally, privilege consists of authentication and full identification of user and agent. More specifically, services may grant tickets, memberships or accounts to agents, each of which provides access to and service from facilitators and service agents. When services are rendered, financial abstractions are utilized. Bills, credit, costs, and payments are all used by services to express financial issues to agents and human users without ambiguity. Transaction logs also exist which describe all transactions made by service. Other abstractions include the service's business card describing access privileges, and the current queue that lists the current queue of agents awaiting service.

Document level (figure 3.4)

The document level describes and classifies media and documents. For example, all media have a name, a size, and some raw data that composes them. Media may be compound or mono (e.g. text, image, video, audio, binary, etc.). Web documents composed in HTML are one type of

document generally marked up in a standard way (e.g. with header information surrounded by `<HEAD>` and `</HEAD>` and body information by `<BODY>` and `</BODY>`). Web documents can be accessed and manipulated by agents. They also have a number of optional components within the body such as links, in-lined images, in-lined applets, and other in-lined objects. Asynchronous documents are also supported. These types of documents are typically time-based and play themselves back to the user [FAL95]. They may also have temporal inter-media relationship requirements that HTML does not support (e.g. a text box containing English subtitles in-synch with an adjoining video in which a man speaks in French). Asynchronous documents have a more complex meta-information distribution consisting of titles, authors, length in minutes, and lists of chapters, sections and pages. Documents are thought of as lists of chapters that point to a number of sections, which in turn point to paragraphs.

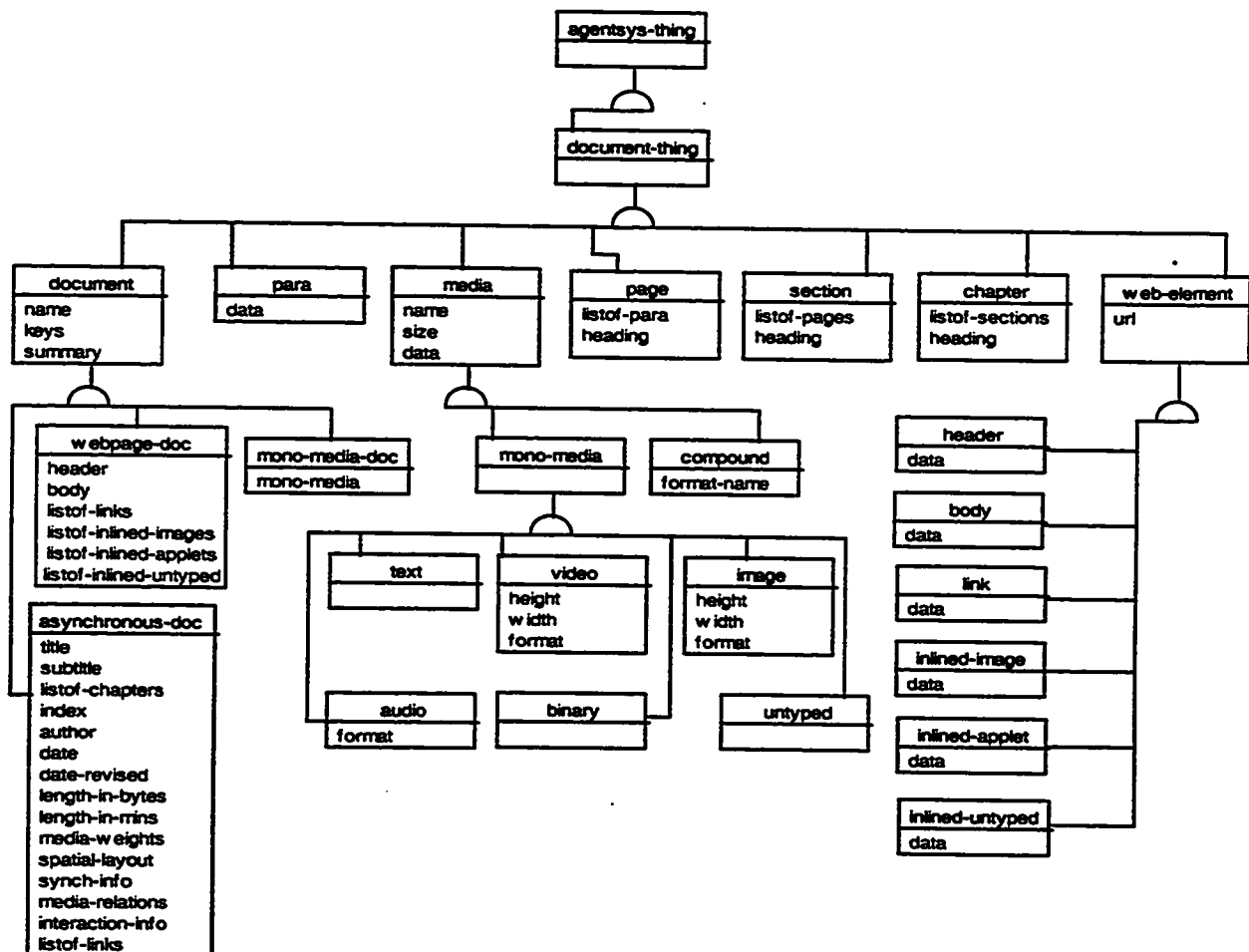


Figure 3.4 Document-level ontology

Note that *network-thing*, *agent-thing*, *document-thing*, and *service-thing* (figures 3.1-3.4) are all children of *AgentSys-thing*, thus forming a complete ontology. Some examples illustrate their functionality. The network ontology allows agents to talk about servers by name or IP-address and lets facilitators be aware of connections and ports that send agents from one site to another. The agent ontology allows one agent to ask another agent, without ambiguity, about its task-type or its human user's name. It also lets agents manage their own "grammar" and speech-acts and log their conversations. This allows an agent ask a facilitator, "Has any agent ever been here trying to negotiate for a document about Paramount Studios?" It also allows one agent to teach a protocol to another agent. The service ontology allows agents to talk about a number of different classifications of services, acquire privileges, and make transactions. Credit and cost can also be negotiated for. The document ontology describes media and documents letting agents exchange and talk about the content and format of data that is to be negotiated for. Both Web documents and asynchronous documents are represented. Note that the ontology is only a grammar for *talking about* asynchronous documents and does not impose or restrict any particular implementation of these documents.

Ontologies do not restrict or limit the implementation of the data store or dictate the way that data should be stored. The actual resources in an agent-system are heterogeneous and can be thought of as a federated database system with no global schema. Particular ontologies must be mapped onto the physical resources on a per-resource basis. A facilitator may have to map a request for a *webpage-doc.body* onto an HTTP GET method, onto a file system, or to a request for a *mono-media* onto an FTP protocol or a relational database. The proposed ontologies provide the basic grammatical units that are used in logically higher layers of inter-agent dialogue. The design of this ontology is novel because: (1) the division of entities into the four major classifications (agents, services, networks, documents) makes it easier for ontologies to evolve and to be partitioned, (2) the simple set of terms semantically spans the set of operations that the mobile agent will be capable of, and (3) the hierarchic organization provides extensibility and reuse. Examples and discussion are provided elsewhere in this chapter.

3.2.1.1 Extending and Sharing

Extension (e.g. specialization) and reuse would seem to be inherent advantages of a hierarchically arranged, object-implemented ontology. While software agents running on *AgentSys* can request and receive ontological terms using specialized messages it is not yet proposed in this thesis how an extended ontology can be dispersed amongst the whole agent community. The following parameters *are* established:

- *Naming* – This thesis proposes that agents request ontologies and terms by name. The naming scheme, however, has not been identified nor imposed. This simple but effective solution makes agents declare their ontology (e.g. *AgentSys1.1*) to the party with which they communicate. The party *must* respond with an *AgentSys1.1*-valid term.
- *Location* – Locations of ontologies must also be specified by agents.

If ontology *O* is updated and extended from version *x* to version *y*, how can agent *A* use this new version? The answer depends on *A*'s implementation. A declaratively implemented agent may have the ability to react autonomously to new ontological terms, and infer meaning from them. For the most part, this kind of inference is extremely difficult. We would propose a scheme in which ontologies are archived at a centralized location (or distributed with update and control) and stored by versions. When new versions are available, agent systems that subscribe to changes (e.g. the KQML publish-subscribe messages could be used [FIN97]) are informed of changes. Changes in the ontology would force changes to the agent classes that wish to use new terms. The extent of these changes would depend upon the nature of the agent implementation.

To illustrate the challenge of dynamic understanding of ontological terms, consider ontology *P1.1* with a term *media*. In *P*, the term *media* has two child terms; *text* and *image*. Now let us say that *P* is extended to become *P1.2*, and a new media type *web-banner* added under the term *media*. If agent *A* knows of and uses ontology *P1.1* and does not know about *P1.2* then the term *web-banner* means nothing to it. This thesis does not propose methods regarding how *A* might infer meaning from new terms in the extended ontology *P1.2*. One could imagine that it might be possible to do so, however, with a little bit of extra logic, rules, or code. For example:

- Agent *A* knows that web-banner is a type of media and a sibling of text, image, etc. Therefore, wherever the user specifies only a media preference (e.g. “give me any media on this topic”), agent *A* knows that web-banner could be used.
- Agent *A* can use ontological relationships to infer meaning via rules or extra logic. For example, if there also exists a relationship PART-OF (webpage-doc, web-banner), agent *A* might use this fact to its advantage.

In summary, the dynamic use of new ontological terms is very difficult and would require extra semantics. This thesis does not propose a system in which agents use extended terms dynamically, rather, they use ontologies strictly by name and location.

3.2.2 Speech Acts and Conversational Modes

AgentSys uses a small set of speech-acts that are the building blocks of inter-agent conversations. A speech-act consists of a performative and content. For example, `tell(agent1,k)` is a speech-act, `tell` is the performative and `agent1,k` is the content. Speech-acts may have different meanings in different contexts; for instance, an *ACK* message in response to a counter proposal may indicate acceptance, whereas an *ACK* in response to a document component may mean, “send more information”. The main goals of the conversation policies (protocols) established here are to:

- Allow agents to authenticate one another
- Allow for the inter-agent exchange of: protocols, documents, user-information, and other queries
- Enable meaningful negotiations to occur between agents and facilitators
- Allow agents to help each other solve tasks in a number of ways

Table 3.1 illustrates the policies and their functionality.

Primitive verbs in the *AgentSys* system appear, for the most part, within conversation policies. They are *ACK*, *decline*, *reject*, *suggest*, *inform*, *accept*, *propose*, and *ctr_propose* (see [FIN97]). Conversation policies can be represented by state diagrams as shown below. *ACK* is a verb typically used to acknowledge a transfer or to indicate that things have gone smoothly. *Decline* and *reject* are used to terminate conversations. Decline means, “I can do this but I choose not

to”, while reject means, “I can’t do this request, because of a semantic error or because of insufficient authorization”. *Suggest* is a way of passing data or instructions from one agent to another in an informal manner. This is unlike *propose* and *ctr_propose* (counter-propose) which are formal offers that occur in negotiation procedures. *Inform* allows agents to tell each other facts or data while *accept* is a context-sensitive ACK.

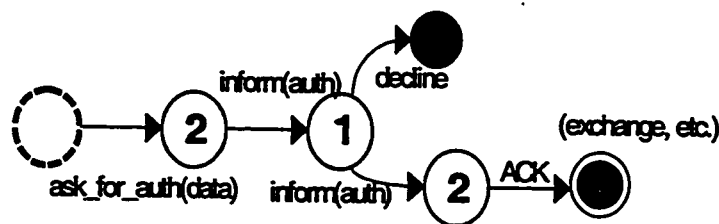
Conversation policy	Functionality
exchange_docs	each agent provides a document to the other
help_with_doc	one agent suggests media that “fits” into another’s document
talk_to_webdoc	an agent asks for representative data from a W3 document
talk_to_synchdoc	an agent asks for representative data from an arbitrary document
query_user_info	ask for information about another agent’s user
ask_for_protocol	exchange a protocol representation
provide_agent	ask for an agent that might be able to help with a problem
negotiate_for	engage in an interactive negotiation for data
ask_for_auth	two agents authenticate each other
tell_auth	one agent demands authentication from another
help_coop	an iterative collaboration between two agents
help_one	one agent helps another
suggest_server	a remote facilitator that might be helpful is suggested
provide_connection	a connection is provided to a remote computer
acquire	associate the named media with the agent – e.g. a “purchase”
receive_msg	receive the next message queued for the agent

Table 3.1 Conversation policies

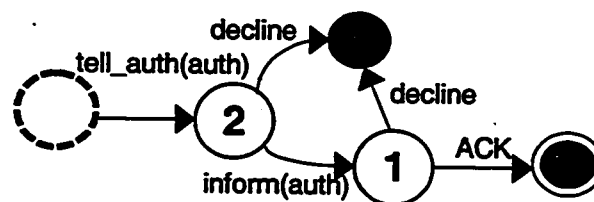
The conversation policies are as follows: *exchange_docs* lets agents query their current holdings of documents and exchange relevant ones with each other. *Help_with_doc* (*hwd*) is used when an agent asks a facilitator to help it achieve media for a document task - this process is iterative. *Help_coop* and *help_one* are similar - the former lets two agents exchange information and suggestions in order to help each other complete a task while in the latter only one agent is in need of help. *Ask_for_auth* is used when two agents wish to authenticate each other, while *tell_auth* is used when one agents demands the authentication of another without having to authenticate itself. *Query_user_info* (*qui*) allows one agent to discover the other’s human-user, profile, and current task. *Ask_for_protocol* (*afp*) is an iterative process in which one agent asks for and receives the representation of a new protocol. *Talk_to_webdoc* and *talk_to_asynchdoc* are conversations that let an agent query a document through the facilitator. The agent can determine the documents’ relevance before negotiating for it. *Provide_local_agent* (*pla*) lets an agent engage with the facilitator while the facilitator suggests other local agents that might have

information to help it with its task. *Negotiate_for* is an iterative process in which two agents continually propose and counter propose until an agreement is reached. *Suggest_server* allows the agent to ask the facilitator for an alternative server and/or facilitator that might have relevant data, and *provide_connection* is the process by which a facilitator grants a socket to an outgoing agent. *Acquire* is the message sent to the facilitator that effectively carries out the “purchase” or selection of a media. The named media in the message is placed (symbolically) into the mobile agent’s local data store where it becomes a part of the cargo. A mobile agent uses *Receive_msg* when it wants to receive the next queued message for itself. The mobile agent therefore controls the “speed” of the conversation using this message.

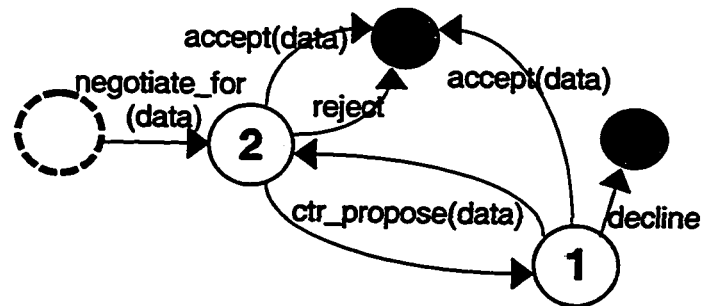
The state diagrams are shown below (not all parameters are shown – the full specification is section 3.2.3). In these diagrams, the dashed circle is the *start state* and the dark circles are *end states* (the dark double circle is a compound end state that implies further conversation). The other circles are *intermediary states* with the receiving entity shown in numeric form (there are two agents communicating at any given time). The arrows are transitions and the text is the verbs in the conversations that invoke those transitions. Since many conversations require the transfer of possibly sensitive data owned by the human user, authentication is important. A *tell_auth* or an *ask_for_auth* conversation to establish credentials and/or authentication precedes almost all conversations.



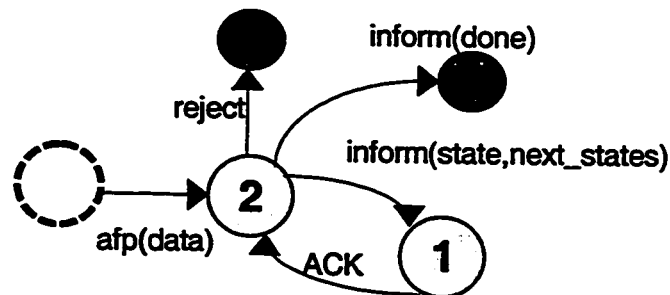
Ask_for_auth : one agent (e.g. facilitator) asks another for its authentication and then provides its own



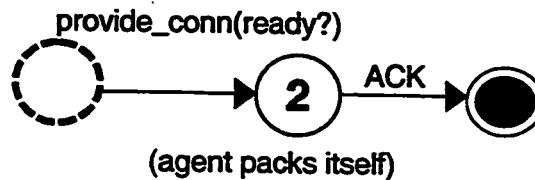
Tell_auth : one agent (e.g. user agent) tells another its authentication.



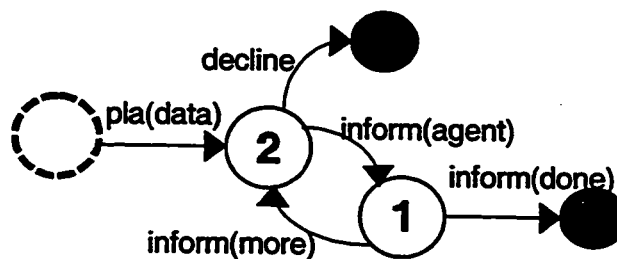
Negotiate_for : two agents engage in counter-proposing.



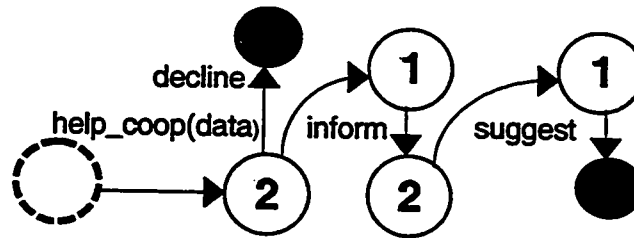
Ask_for_protocol (afp) : an agent requests the initiation of a protocol representation



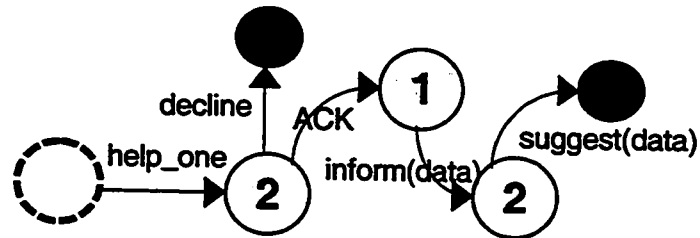
Provide_connection : one agent asks another to be sent to another node



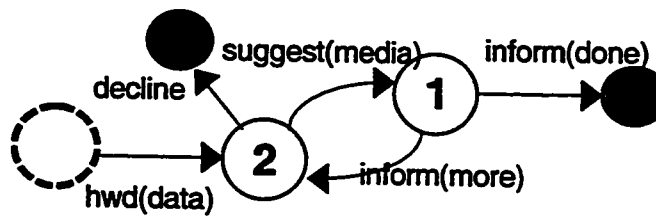
Provide_local_agent : a request to provide a handle to another local agent that might be useful.



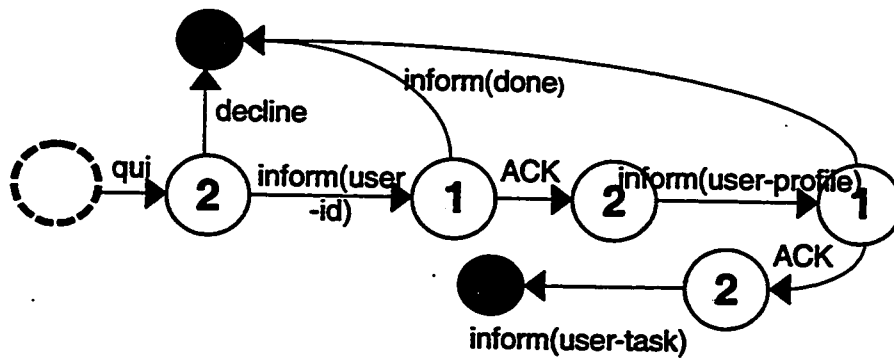
Help_coop : two agents first exchange task-info and then each suggest data



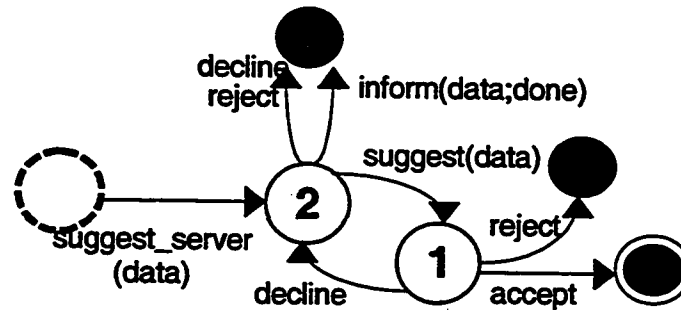
Help_one : one agent asks for a suggestion from another



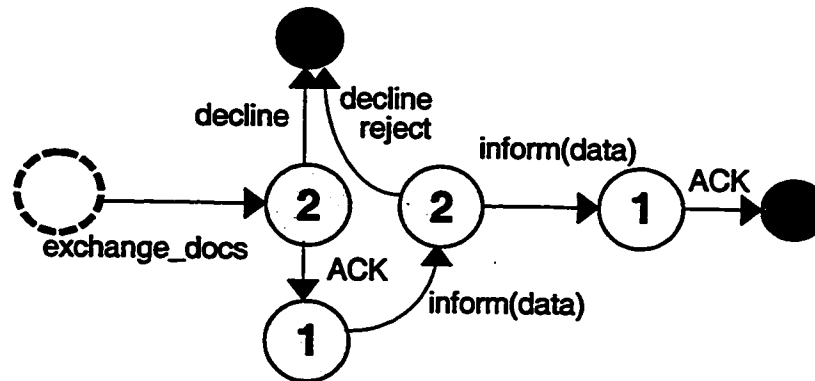
Help_with_document (hwd) : one agent asks another to help it “fill in” a document with relevant media



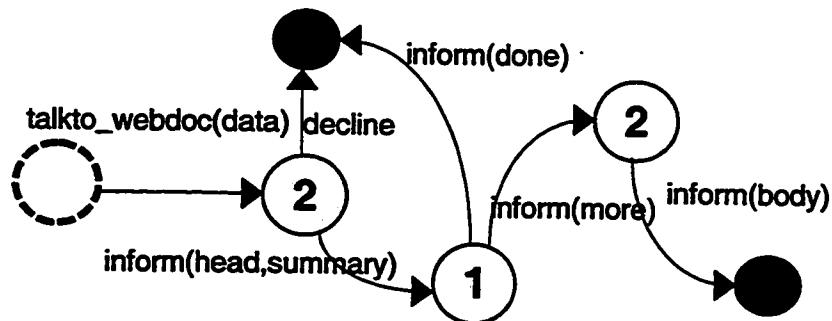
Query_user_info (qui) : an agent (e.g. facilitator) asks for a mobile agent’s user information



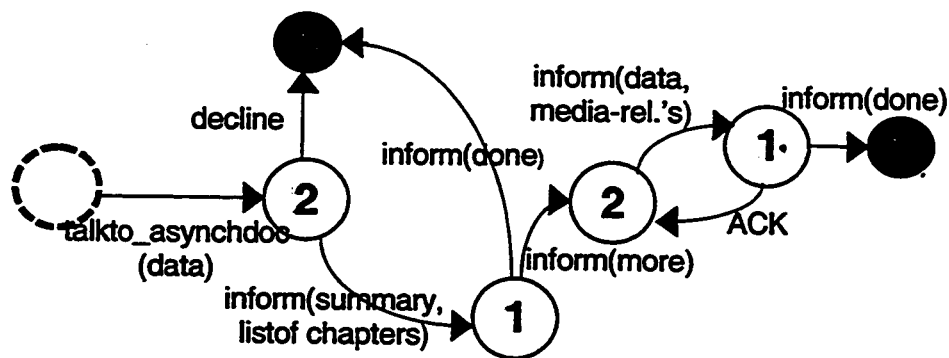
Suggest_server : an agent asks another to suggest a remote server where it may find relevant media



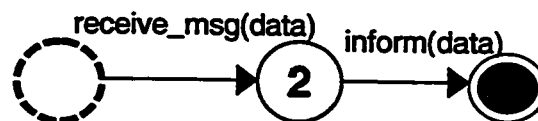
Exchange_docs : two agents exchange documents



Talkto_webdoc : an agent 'talks' to a WWW document by asking for its head and body.



Talkto_asynchdoc : an agent “talks” to an asynchronous document by obtaining its critical data



Receive_msg : an agents asks for the next queued message for it. This is how the conversation “steps” along



Acquire : a media is selected for addition to a mobile agent’s cargo using this message (e.g. a “purchase”)

3.2.3 AgenTransact Message Format and Grammar

Messages are sent between agents in a format similar to the KQML message format. In this format, there are parameters, a performative, and content. The parameters specify meta-information related to the message, including the originator and what the response must be. The performative indicates the speech-act, and the content of the message contains further parameters and data that qualify the speech-act (e.g. specifying the server name in a request for a connection). In the following specifications the terms in bold indicate string literals that are used as parameters. These strings, however, may be replaced by any other term from the *AgentSys* ontology that has the given term as a predecessor. The format of messages is based on the KQML format and is as follows:

```

Performative(
  :ontology <string>
  :sender <agent>
  :receiver <agent>
  :in-reply-to <string>
  :reply-with <string>
  :content ( ... )
)

```

The *Performative* is the message name, replaced by one of several messages (see below). The *ontology*, *sender*, *receiver*, and *content* parameters are self-explanatory. The *in-reply-to* and *reply-with* parameters essentially allow the creation of “labels” that messengers may use to keep messages in context. The message grammar is as follows (terms in bold are literals):

Message = CommunicativeAct “(” MessageParms “,” MessageContent “)”.

```

CommunicativeAct = "ACK"
                  "decline"
                  "reject"
                  "suggest"
                  "inform"
                  "accept"
                  "propose"
                  "ctr_propose"
                  "exchange_docs"
                  "help_with_doc"
                  "talk_to_webdoc"
                  "talk_to_synchdoc"
                  "query_user_info"
                  "ask_for_protocol"
                  "provide_agent"
                  "negotiate_for"
                  "ask_for_auth"
                  "tell_auth"
                  "help_coop"
                  "help_one"
                  "suggest_server"
                  "provide_connection"
                  "acquire"
                  "receive_msg".

```

MessageParms = RequiredComponents OptionalComponents.

```

RequiredComponents =   MessageSender “,”
                      MessageReceiver

```

```

OptionalComponents =   [ “,” MessageOntology ]
                      [ “,” MessageInReplyTo ]
                      [ “,” MessageReplywith ]

```

```

MessageSender = “:sender” SP String.
MessageReceiver = “:receiver” SP String.
MessageOntology = “:ontology” SP String.
MessageInReplyTo = “:in-reply-to” SP String.
MessageReplywith = “:reply-with” SP String.

```

```

MessageContent = “:content” “(” MessageContentEncoding “)”.
MessageContentEncoding = String | CharSequence | ByteSequence.

```

SP = <US-ASCII SP, space (32)>.

The messages that are most important (and complex) are the ones which start conversations (as opposed to those like *ACK* and *inform* which are general purpose). These important messages are typed below. Terms in bold are ontological terms and may be replaced with any single terms that are direct descendents in the ontology hierarchy. The message contents are those data within the parentheses. Note that some messages need not carry any further data within the content portion of the message because the required message components absorb this additional data.

```
ask_for_auth :content (privilege-thing, string UserId)
tell_auth :content (privilege-thing, int AuthNumber, string UserId)
negotiate_for :content (document|media|financial-thing|privilege_thing,
string ObjectName, int Amount)
ask_for_protocol :content (string Name)
provide_connection :content (server, string ServerName, boolean TakeCargo,
boolean HiPriority, string NetProtocol)
provide_agent :content (media|document, string TaskDesc, vector ListofKeys)
help_coop :content (media|document, string TaskDesc, vector ListofKeys)
help_one :content (media|document, string TaskDesc, vector ListofKeys)
help_with_doc :content (media|null, string TaskDesc, vector ListofKeys)
query_user_info :content (null)
suggest_server :content (media|document, string TaskDesc, vector ListofKeys)
exchange_docs :content (null)
talkto_webdoc :content (string DocName)
talkto_asynchdoc :content (string DocName)
acquire :content (document|administration-thing|media, string ObjectName, int
value)
receive_msg :content (null)
```

3.2.4 An Example

Consider a mobile agent that must collect documents and media based on the human-given profile. The following steps occur after the agent migrates to a new AEE.

1. The agent initiates a *ask_for_auth* protocol with the facilitator to inform it of its authentication while at the same time assuring itself that the facilitator is not an impostor.
2. The agent then initiates a *help_with_document* (hwd) protocol, providing the human user's profile and task description using the ontological semantics - e.g. *agent.user.user-profile*, and *agent.agent-content.agent-task-type* - and waits for a response. Using the *AgentSys* ontology there is no ambiguity as to what components are in question.
3. The facilitator reads this data and proceeds with some processing while the agent blocks and waits for a returned message, either a *decline* or a *suggest*. The facilitator sends a *suggest* message containing the name of a document that it feels matches the user's profile and task, then blocks.
4. The agent wants another one so it responds with *inform(more)*, after which the facilitator sends another suggestion.
5. The agent sends an *inform(done)* message and examines in detail the documents that have been suggested (this may require retrieving *summary* information or the media themselves).
6. Having decided that these documents are sufficient (see Chapter 6), the agent informs the facilitator it wishes to return to its human user with the results. After two *acquire* messages (one for each document), the facilitator issues a *provide_connection(ready?)* message to the agent and waits for a response. The agent then does the appropriate ordering and packing of its contents and returns an *ACK* message at which point the facilitator sends the agent onto a socket to its home.

Note that the facilitator may be managing more than one conversation at a time so that when it "blocks" it does so only in the sense of that particular conversation. Several threads, each of which controls a conversation, are managed at once. Furthermore, the *receive_msg* messages are left out of this example but are implied at each "stage" in all agent-facilitator conversations. To achieve Facilitator-agent message-exchanges, a "blackboard" mechanism is used. In lieu of blocking, waiting, and sending a response directly to the calling agent, the Facilitator posts all responses to agent-originating messages onto a blackboard. When the calling agent wishes to have the response to its message, it sends a *receive_msg* message and the Facilitator retrieves the appropriate data from the blackboard. This enables asynchronous agent-controlled message passing. The blackboard mechanism is described further in Chapter 5.

In summary, agent communications can be thought of occurring within circles of nested control. From the outside (or highest level) to the inside the control is: (1) Agents, (2) Conversations, (3) Speech-acts and data, (4) Ontology terminology, (5) Mappings to physical resources. Agents engage in conversations that are composed of individual speech-acts. Different conversation policies may be implemented on the same set of speech-acts, or, the same conversation may take on a different meaning when a different set of speech-acts is used. The content of messages may refer to or be mapped to specific parts of an ontology. Local facilitators are responsible for retrieving media from physical resources to which mobile agents have no access.

3.3 An Inter-AEE Protocol - *AgentTransfer*

The HTTP protocol is currently in wide use as a methodology for transferring Web pages from servers to clients (e.g. *Navigator*). HTTP is a simple protocol that supports seven methods and two message types. Currently there are no such standards for agent transfer except the emerging IBM offering, ATP/0.1 [IBM96]. In [TAC95] and [WHI94] transfer functionality is embedded in the operating system but this is an inflexible, platform-dependent approach. ATP is an acceptable starting point but is lacking in several key areas. This section details how we extend it to support key future functionalities of mobile agents.

Protocol suites typically offer the following functions: connection, flow, and error control, ordered delivery, synchronization, addressing, segmentation, re-assembly, and encapsulation. An agent transfer protocol such as the one proposed here resides in the OSI *application layer* and as such, lower layers take care of the first six items listed above. An agent transfer protocol is mainly concerned with segmentation and re-assembly of agent-related data (e.g. cargo, state, etc.), encapsulation of agents, and addressing remote servers and agents at the application level. In order to define the transfer protocol for agents the following items must be specified: message types and fields, headers for messages, and the methods and their syntax that specifically accommodate the components of mobile agents.

The IBM ATP/0.1 (Agent Transfer Protocol) [IBM96] attempts to create a generic protocol for as many agent systems as possible by allowing machines to have single unique names independent of the agent platforms existing on them. It focuses on: agent naming, identifiers,

transportation and protection. Based on the request/response paradigm, ATP offers three standard patterns for agent service: (i) *Dispatch* - this method is used to send an agent from one system to another, (ii) *Retract* - this method is used to get an agent from a remote system, and (iii) *Fetch*, which is used to fetch executable code or classes from a remote system. ATP uses the Universal Resource Identifier (URI) naming scheme to name agent resources (except "atp:" and port 434 are used versus the Web's "http:" and port 80). The complete syntax for ATP can be found in [IBM96] but Figure 3.5 illustrates the three main requests and possible responses. Important points are as follows:

1. The message body is mono-typed as either text, postscript, image etc. This makes the transfer of compound documents difficult. The same applies to the encoding since an agent carrying a number of media objects may need to encode each object differently within its body.
2. The agent language is specified but not the particular ontology. This can be a problem since the language may have a different semantic within different ontologies and vice-versa
3. There are no headers to support the sending of remote messages and conversational speech-acts.

There are ten header fields in ATP/0.1 and their use depends upon on the request being used (see table 3.2). Headers precede the method in both the request and response modes. For example, the CONTENT-TYPE field is included in the DISPATCH method and in the RESPONSE but is not relevant in either the RETRACT or FETCH methods.

Examples of the ATP/0.1 Protocol

request A (dispatch agent) ----->

DATE: Sat, 12 Jan 1997 09:03:00 GMT
 SENDER-APPLICATION: AgentSys-facilitator
 FROM: ben@sol.genie.uottawa.ca
 AGENT-SYSTEM: AgentSys
 AGENT-LANGUAGE: AgenTransact
 CONTENT-TYPE: text/plain
 CONTENT-ENCODING:
 CONTENT-LENGTH: 8024
 DISPATCH atp://some.machine.com ATP/0.1
 agent data

 agent data ends.

request B (retract agent) ----->

DATE: Sat, 12 Jan 1997 09:03:00 GMT
 SENDER-APPLICATION: AgentSys-facilitator
 FROM: ben@sol.genie.uottawa.ca
 AGENT-SYSTEM: AgentSys

response A (ACK)

DATE: Sat, 12 Jan 1997 09:04:23 GMT
 RECIPIENT-APPLICATION: AgentSys-facilitator
 AGENT-SYSTEM: AgentSys
 AGENT-LANGUAGE: AgenTransact
 CONTENT-TYPE: text/plain
 CONTENT-ENCODING:
 CONTENT-LENGTH: 15
 ATP/0.1 100 OKAY

response B (provide agent)

DATE: Sat, 12 Jan 1997 09:05:12 GMT
 RECIPIENT-APPLICATION: Aglets-facilitator
 AGENT-SYSTEM: AgentSys
 AGENT-LANGUAGE: AgenTransact

AGENT-LANGUAGE: AgenTransact CONTENT-LENGTH: 66 COOKIE: 09893413 RETRACT atp://some.machine.com# ben.some.machine.com%231%323 ATP/0.1	CONTENT-TYPE: text/plain CONTENT-ENCODING: ATP/0.1 100 OKAY (or possibly, ATP/0.1 302 NOT FOUND) agent data agent data ends.
<u>request C (fetch a class)</u> -----> DATE: Sat, 12 Jan 1997 09:03:00 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: ben@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-LENGTH: 54 FETCH atp://some.machine.com/classes/Some.Class ATP/0.1	<u>response C (provide class)</u> DATE: Sat, 12 Jan 1997 09:05:12 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 8099 ATP/0.1 100 OKAY class data class data ends.

Figure 3.5 Examples of ATP/0.1 functionality as it exists before the proposed extensions

While *AgentTransfer*, the protocol introduced here, extends ATP/0.1, its overall goal remains the same. In order to make operations that are more complex available to media-collecting mobile agents the two protocols diverge in some respects. While ATP considers agent-communication to be outside the scope of the transfer protocol, *AgentTransfer* recognizes that the protocol must indirectly support fields and messages that help agents inter-communicate. For example, agents carry with them ontological data (e.g. data about data) and complex inter-agent relationships. It will be common for an agent to interrupt its conversation with a facilitator and ask to be sent to another one to resume this conversation. Moreover, the destination agent system may use a different semantic for data so the agent may also have to transfer its ontology and knowledge along with its executable self. In summary, an agent transfer protocol should support the following functionality in addition to the basic functionality provided by ATP/0.1 (e.g. dispatching, fetching, and retracting)

HEADER FIELD	REQUEST			RESPONSE
	DISPATCH	RETRACT	FETCH	
DATE	yes	yes	yes	yes
SENDER-APPLICATION	yes	yes	yes	no
RECIPIENT-APPLICATION	no	no	no	yes
FROM	yes	yes	yes	no
AGENT-SYSTEM	yes	yes	yes	yes
AGENT-LANGUAGE	yes	yes	yes	yes
CONTENT-TYPE	yes	no	no	yes
CONTENT-ENCODING	yes	no	no	yes
CONTENT-LENGTH	yes	yes	yes	yes
COOKIE	no	yes	no	no

Table 3.2 Existing header fields associated with requests and responses [IBM96]

- *Multimedia cargo* - cargo should be closely associated with the agent that owns it and must be transferred with appropriate agent-routing semantics.
- *Ontology packaging* - in a complex agent system more than one ontology will be supported. If a destination agent system requires an ontology it must be identified and transferred correctly.
- *Remote messaging* between agents - when agents communicate there may be cases where it is desirable to send messages remotely rather than transfer the agent so it resides physically with the other.
- *Agent querying* - a facilitator or human user may need to inquire of a remote agent's status (this issue is to be addressed by future ATP drafts) - the queries and results must be properly transferred.
- *Agent cloning* - semantics for both parent and child agents.

The elements listed above, combined with the original ATP functionality, can be considered a *threshold* model for agent transfer operation. Any agent transfer entity should be capable of such operations. *AgentTransfer* provides this by extending methods, header, and body semantics.

3.3.1 Extending ATP/0.1 Functionality

This section describes and illustrates how the existing ATP/0.1 protocol can be extended to support the important threshold features listed above. Extensions all preserve the request/response two-way communication mode between the parties, as per the original IBM proposal as well as the HTTP protocol.

Multimedia Cargo and Documents

Mobile agents very often carry with them some form of cargo that represents the results of a media search in the form of data. A few variations on cargo that should be supported but that depend upon the particular agent system:

- *Modes of collecting media* - the agent may carry media with it from node to node, or it may send individual media back to the user's static (local) agent as it finds them. Either way, the media must be transferred either on a per-hop basis or on a send-home basis.

- *Modes of carrying media* - an agent carrying media with it may encapsulate the media in its body (in which case the agent and media are transferred as a “bag”) or the media may be sent in separate streams to the destination node (this is arranged either by the agent or the cooperating facilitators).

A characteristic of multimedia data is that it ranges from being extremely storage intensive (e.g. Apple QuickTime video) to relatively lightweight (e.g. ASCII text). For this reason, a cargo consisting of different media-types may require several different encodings (for example postscript files may be g-zipped while HTML files will likely go unchanged). In both RETRACT and DISPATCH modes cargo may be associated with the action. In the former case an agent is retracted from a remote host with an option to also retract the media that it is (possibly) currently carrying. In the latter case an agent is dispatched from its current location to another node and its cargo is sent immediately thereafter. Since multiple media may be involved it is up to the local facilitators to use a session-type semantic to achieve the transfer. First the agent is sent and then media are sent one by one to the remote node, allowing each to be encoded in its own way (since the protocol remains stateless though, facilitators must track which media to send and when). To achieve this, the following extension is introduced to the message header fields:

NEW HEADER FIELDS	REQUEST			RESPONSE
	DISPATCH	RETRACT	FETCH	
AGENT-IDENTIFIER	yes	yes	no	yes
MEDIA	yes	yes	no	yes
MEDIA-COMING	yes	no	no	yes
MEDIA-RECIPIENT	yes	no	no	yes

Table 3.3 Extensions to the headers to support multimedia cargo

AGENT-IDENTIFIER provides a level of identity for the media transfer. This field identifies the agent locally with a unique identifier that is independent of the agent system. The transfer entity maps these identifiers to the agent systems. MEDIA has a number of roles. In a response, setting MEDIA to *true* indicates an acknowledgment of a piece of *cargo*, not an agent. In a DISPATCH, setting MEDIA to *true* indicates that *cargo* is being dispatched, and in a RETRACT, MEDIA set to *true* indicates that the requester wishes to receive the next media in the agent’s cargo, not the agent itself. MEDIA-COMING in a DISPATCH method indicates that

the agent is in the body of the message but that there is an associated cargo that is on the way next. In a RESPONSE, MEDIA-COMING set to *true* indicates that an agent or media is being sent in the body and that there is more cargo that is on the way next. MEDIA-RECIPIENT in both the RESPONSE and DISPATCH methods is used to ensure that, upon arrival, the cargo being sent is associated with the correct agent.

This new functionality allows agents to be transferred while maintaining a close association with their media. The following examples illustrate how the extensions to ATP/0.1 can be used to complete such a previously unsupported transferal. In Figure 3.6, the agent is dispatched from one node to another and is followed by its media consisting of an HTML page and the inlined-image. In Figure 3.7, an agent is retracted from a remote node and its media follow consisting of an image and a postscript file.

Dispatching an agent and its cargo

request (dispatch the agent) ----->

DATE: Sat, 12 Jan 1997 09:03:00 GMT
 SENDER-APPLICATION: AgentSys-facilitator
 FROM: ben@sol.genie.ottawa.ca
 AGENT-SYSTEM: AgentSys
 AGENT-LANGUAGE: AgenTransact
 CONTENT-TYPE: text/plain
 CONTENT-ENCODING:
 CONTENT-LENGTH: 8024
 AGENT-IDENTIFIER: 09092
 MEDIA:
 MEDIA-COMING: true
 MEDIA-RECIPIENT:
 DISPATCH atp://some.machine.com/ ATP/0.1

agent data

... ..

agent data ends.

request (send media - image) ----->

DATE: Sat, 12 Jan 1997 09:05:00 GMT
 SENDER-APPLICATION: AgentSys-facilitator
 FROM: ben@sol.genie.ottawa.ca
 AGENT-SYSTEM: AgentSys
 AGENT-LANGUAGE: AgenTransact
 CONTENT-TYPE: image/gif
 CONTENT-ENCODING: gzip
 CONTENT-LENGTH: 7833
 AGENT-IDENTIFIER:
 MEDIA: true
 MEDIA-COMING: true
 MEDIA-RECIPIENT: 09092
 DISPATCH atp://some.machine.com/ ATP/0.1

image data

... ..

image data ends.

request (send media - html) ----->

response (ACK)

DATE: Sat, 12 Jan 1997 09:04:23 GMT
 RECIPIENT-APPLICATION: AgentSys-facilitator
 AGENT-SYSTEM: AgentSys
 AGENT-LANGUAGE: AgenTransact
 CONTENT-TYPE: text/plain
 CONTENT-ENCODING:
 CONTENT-LENGTH: 15
 AGENT-IDENTIFIER: 09092
 MEDIA:
 MEDIA-COMING:
 MEDIA-RECIPIENT:
 ATP/0.1 100 OKAY

response (ACK image)

DATE: Sat, 12 Jan 1997 09:05:43 GMT
 RECIPIENT-APPLICATION: AgentSys-facilitator
 AGENT-SYSTEM: AgentSys
 AGENT-LANGUAGE: AgenTransact
 CONTENT-TYPE: text/plain
 CONTENT-ENCODING:
 CONTENT-LENGTH: 15
 AGENT-IDENTIFIER: 09092
 MEDIA: true
 MEDIA-COMING:
 MEDIA-RECIPIENT:
 ATP/0.1 100 OKAY

response (ACK html)

DATE: Sat, 12 Jan 1997 09:05:50 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: ben@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/html CONTENT-ENCODING: CONTENT-LENGTH: 1132 AGENT-IDENTIFIER: MEDIA: true MEDIA-COMING: false MEDIA-RECIPIENT: 09092 DISPATCH atp://some.machine.com/ ATP/0.1 text data text data ends.	DATE: Sat, 12 Jan 1997 09:05:55 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 15 AGENT-IDENTIFIER: 09092 MEDIA: true MEDIA-COMING: MEDIA-RECIPIENT: ATP/0.1 100 OKAY
--	--

Figure 3.6 A dispatch operation including the transfer of an agent followed by its cargo - an image and an html file.

Retracting an agent and its cargo	
<u>request (retract the agent) -----></u> DATE: Sat, 12 Jan 1997 09:03:00 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: ben@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-LENGTH: 58 COOKIE: 877 AGENT-IDENTIFIER: 76543 MEDIA: RETRACT atp://some.machine.com #ben.some.machine#900%800 ATP/0.1	<u>response (provide agent)</u> DATE: Sat, 12 Jan 1997 09:04:23 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 7032 AGENT-IDENTIFIER: 76543 MEDIA: MEDIA-COMING: true MEDIA-RECIPIENT: 76543 ATP/0.1 100 OKAY agent data agent data ends.
<u>request (request cargo) -----></u> DATE: Sat, 12 Jan 1997 09:05:00 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: ben@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-LENGTH: 8 COOKIE: 877 AGENT-IDENTIFIER: 76543 MEDIA: true RETRACT atp://some.machine.com #ben.some.machine#900%800 ATP/0.1	<u>response (send media - image)</u> DATE: Sat, 12 Jan 1997 09:05:05 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: ben@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: image/gif CONTENT-ENCODING: gzip CONTENT-LENGTH: 8833 AGENT-IDENTIFIER: 76543 MEDIA: true MEDIA-COMING: true MEDIA-RECIPIENT: 76543 ATP/0.1 100 OKAY image data image data ends.
<u>request (ACK image) -----></u> DATE: Sat, 12 Jan 1997 09:06:23 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain	<u>response (send media - postscript file)</u> DATE: Sat, 12 Jan 1997 09:06:40 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: ben@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact

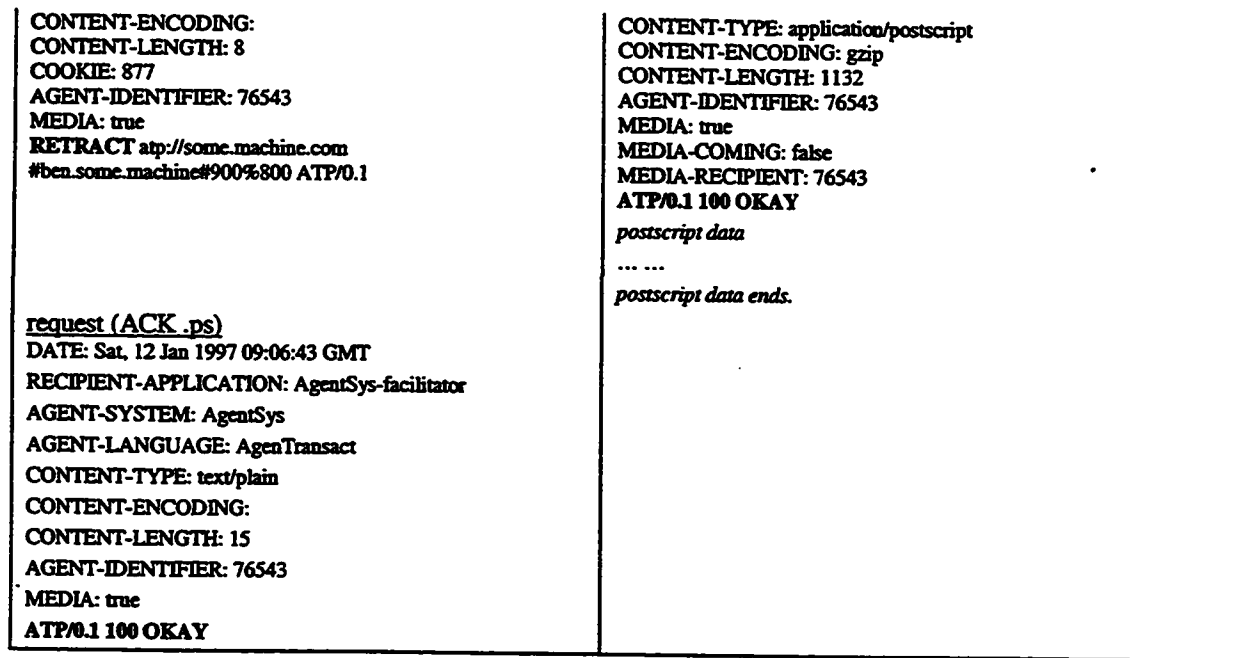


Figure 3.7 Retracting an agent and its cargo - an image and a postscript file

Figure 3.8 shows a timing diagram for the above dispatch example and Figure 3.9 illustrates the retract example. The implication of this methodology is that cargo is transferred separately while remaining semantically linked to the associated agent through the AGENT-IDENTIFIER field. The transfer protocol does not restrict the implementation of the cargo operations so that media may be sent one-after-another or on parallel connections.

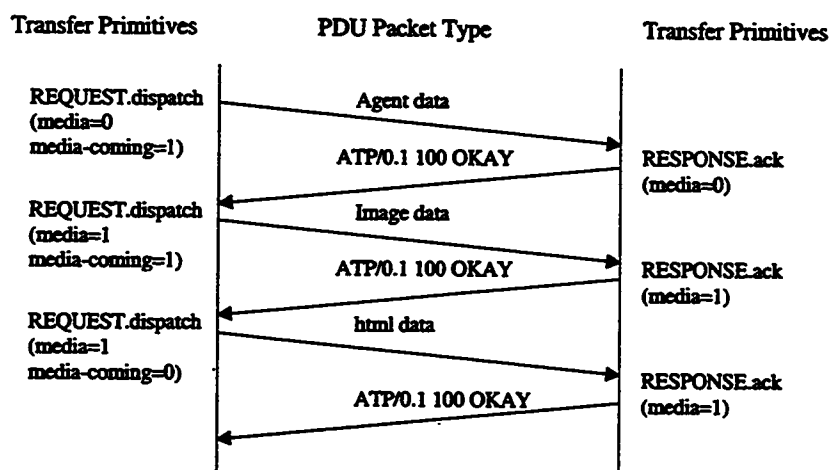


Figure 3.8 Timing diagram for an agent dispatch followed by two separate media representing cargo.

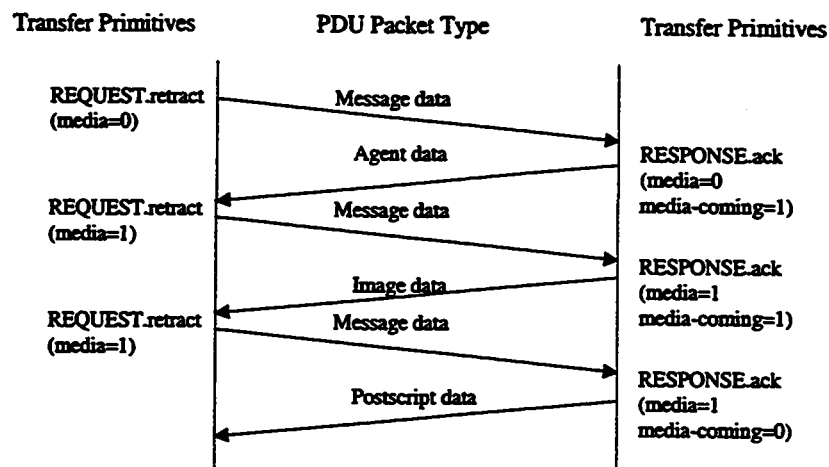


Figure 3.9 Timing diagram for the retraction of an agent followed by its cargo.

Ontology Semantics

For agents to communicate about data or other abstractions such as networks, servers, and services, they must share a common conceptualization of these abstractions. This conceptualization is called an *ontology*. There is no single universal ontology but rather domain-specific ones (e.g. medicine, libraries etc.) For the most part these are not internationally accepted but rather adopted only by communicating parties. Nonetheless, an ontology is a crucial part of an agent system. The current draft ATP/0.1 does not support these in any way.

With the following simple extension to ATP/0.1 agent systems can identify and fetch ontologies as an integral part of the transfer protocol. The simple modifications are fields called **ONTOLOGY** and **ONTOLOGY-LOCATION** in the header. The new fields are used in the **DISPATCH** and **FETCH** modes as shown in table 3.4.

NEW HEADER FIELDS	REQUEST			RESPONSE
	DISPATCH	RETRACT	FETCH	
ONTOLOGY	yes	no	yes	yes
ONTOLOGY-LOCATION	yes	no	yes	yes

Table 3.4 Adding a field to the header to support ontology

In DISPATCH mode, an agent and possibly its cargo is being sent to a remote node for execution. Yet to be able to negotiate with mobile agents the destination system must adhere to the same abstractions. The ONTOLOGY field names the ontology being used by the incoming agent and the ONTOLOGY-LOCATION field identifies from where the ontology can be fetched in case it is unknown at the destination system. These two fields are not used in a response to a DISPATCH. In FETCH mode, an agent system and a newly arrived mobile agent have need of an ontology in order to successfully communicate. The agent system issues a FETCH with the name and location of the ontology specified in the new fields. The ONTOLOGY-LOCATION must completely specify the location, including the file path. To allow flexibility, the ontology location is not supplied at the message level (e.g. as an argument in the method) since this would require conformance to the strict URI naming methodology - only the remote machine's URL must be supplied. In a response to the FETCH, the ontology is contained in the message body and the ONTOLOGY and ONTOLOGY-LOCATION fields must remain set to distinguish the message body from an agent-class, which may also be returned using a FETCH method.

Figures 3.10 and 3.11 illustrate the dispatching of an agent and the fetching of an ontology. Note that new header fields introduced in earlier sections are not shown unless they are explicitly needed. Figures 3.12 and 3.13 illustrate the timing diagrams for the above two examples.

<i>Dispatching an agent and identifying its ontology</i>	
<u>request (dispatch agent)</u> -----> DATE: Sat, 12 Jan 1997 09:03:00 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: ben@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 8024 ONTOLOGY: agentsys-documents ONTOLOGY-LOCATION: atp://.../ontologies/mm-doc.ont DISPATCH atp://some.machine.com ATP/0.1 agent data agent data ends.	<u>response (ACK)</u> DATE: Sat, 12 Jan 1997 09:04:23 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 15 ATP/0.1 100 OKAY

Figure 3.10 Dispatching an agent and identifying its ontology and location

<i>Fetching an ontology</i>	
<u>request (retract the ontology)</u> -----> DATE: Sat, 12 Jan 1997 09:03:00 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: sysadmin@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgentTransact CONTENT-LENGTH: 19 COOKIE: 877 ONTOLOGY: agentsys-documents ONTOLOGY-LOCATION: atp://.../ontologies/mm-doc.ont FETCH atp://remote.machine.com ATP/0.1	<u>response (provide ontology)</u> DATE: Sat, 12 Jan 1997 09:04:23 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgentTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 7032 ONTOLOGY: agentsys-documents ONTOLOGY-LOCATION: atp://.../ontologies/mm-doc.ont ATP/0.1 100 OKAY ontology data ontology data ends.

Figure 3.11 Fetching an ontology from a remote system

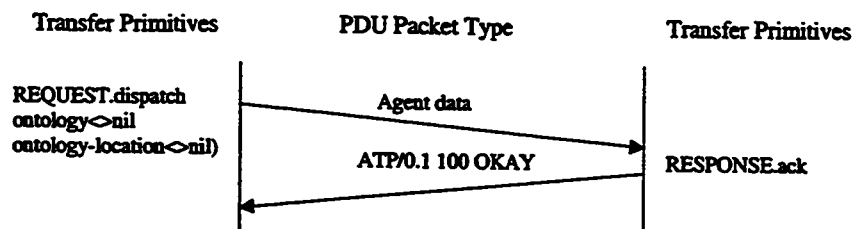


Figure 3.12 Timing diagram for agent dispatch with ontology specification

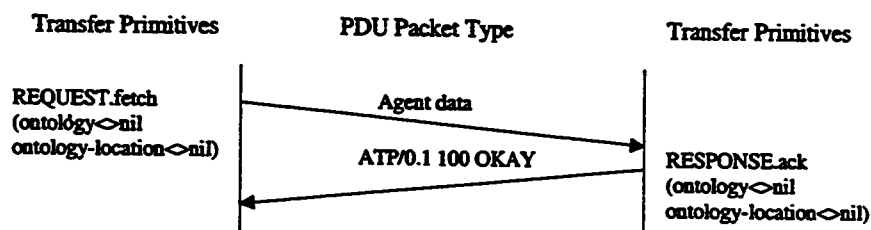


Figure 3.13 Timing diagram for an ontology fetch

Remote Messaging

Although agents typically co-locate at a common node and communicate via local messages, there may be situations where this mode is either not possible or not allowed. For example when:

- an agent is not allowed to run in a certain sub-net or domain and is thus restricted to messaging
- it is “cheaper” to run the agent locally and send messages than to send the entire agent.

To support the transfer of messages between agents it is necessary to create a new method for ATP/0.1. This method, called REMOTE_MSG, is different from DISPATCH that is used to send agents themselves. The transfer protocol is not concerned with how individual agent

systems implement support for conversation but rather supports the general procedure of sending messages between remote agents.

NEW HEADER FIELDS	REQUEST	RESPONSE
	REMOTE_MSG	
ONTOLOGY*	yes	yes
ONTOLOGY-LOCATION*	yes	yes
PROTOCOL	yes	yes
CONVERSATION-ID	yes	yes
SPEECH-ACT	yes	yes

Table 3.5 The REMOTE_MSG method. Headers marked with "*" have been previously defined. All other standard headers are required in both request and response modes in the REMOTE_MSG context.

In request mode, the ONTOLOGY and ONTOLOGY-LOCATION fields are used for the transfer of ontologies but are also required in the REMOTE_MSG method since speech-acts and conversational messages occur within an ontological context. The second of these fields can be used when one agent system needs to install an ontology from a remote site. The PROTOCOL field identifies the name of the protocol being used to achieve the conversation. The CONVERSATION-ID is maintained by the two communicating facilitators and uniquely identifies the conversation (this is important since there may be more than one remote messaging sessions occurring at a time). The SPEECH-ACT field identifies the speech act to be sent and the message body contains the data associated with the speech act. Other standard ATP/0.1 fields are important here such as AGENT-SYSTEM, AGENT-LANGUAGE, SENDER-APPLICATION, RECIPIENT-APPLICATION, FROM, etc.

In response mode, the receiving agent system acknowledges the delivery of the speech act and data and sends the response including reiteration of the header fields from the request. There is no message body in the response, only a response code. Figure 3.14 illustrates the use of the REMOTE_MSG method and the necessary fields - in this case an agent remotely initiates a negotiation conversation in order to negotiate for some data. The agent URL in the method identifies the agent to which the message is intended. Figure 3.15 illustrates the timing diagram for a remote message "session" between two agents.

Issuing a remote message to another agent	
<u>request (send a remote message)</u> -----> DATE: Sat, 12 Jan 1997 09:03:00 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: sysadmin@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-LENGTH: 298 ONTOLOGY: agentsys-documents ONTOLOGY-LOCATION: atp://.../ontologies/mm-doc.ont PROTOCOL: AgenTransact CONVERSATION-ID: 0800 SPEECH-ACT: negotiate_for REMOTE_MSG atp://some.machine.com# ben.some.machine.com%232%444 ATP/0.1 message data COST(mm-doc, 98789, \$0.1, ...) message data ends.	<u>Response (acknowledge message)</u> DATE: Sat, 12 Jan 1997 09:03:23 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 15 ONTOLOGY: agentsys-documents ONTOLOGY-LOCATION: atp://.../ontologies/mm-doc.ont PROTOCOL: AgenTransact CONVERSATION-ID: 0800 SPEECH-ACT: negotiate_for ATP/0.1 100 OKAY

Figure 3.14 Using REMOTE_MSG to send a speech act as part of a conversation to another agent

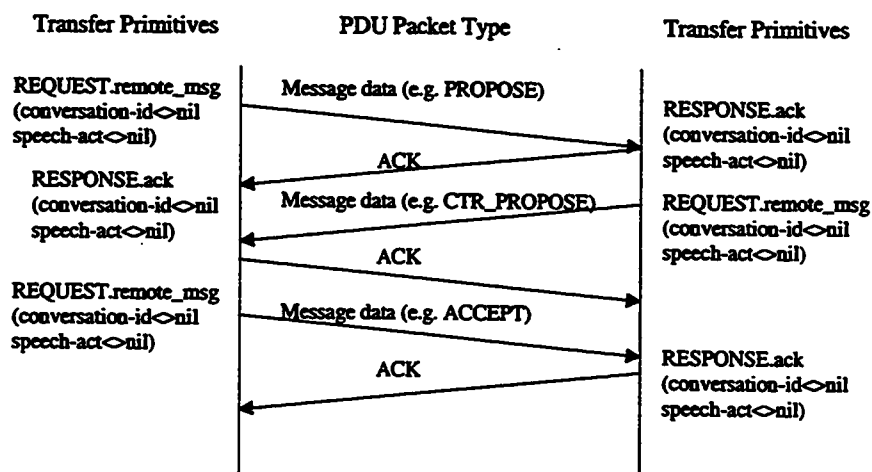


Figure 3.15 Timing diagram for remote messages

Agent Querying

Agents executing remotely, whether they are child agents or user-agents, will typically be able to be queried regarding their status. Status information might include the number of media collected and logs of conversations and transactions. This type of information is agent system dependent but the query action is one that should be supported at the transfer level and should include the ability to encapsulate simple queries and results inside the message. Queries differ subtly from remote messages delivered via the REMOTE_MSG method - the latter typically occur within conversations (agent interactions) whereas a query is a simple request for status

information from a remotely operating agent. Table 3.6 illustrates the QUERY method to allow this functionality – no new headers are required.

NEW HEADER FIELDS	REQUEST	RESPONSE
none	QUERY	n/a
	n/a	n/a

Table 3.6 The QUERY method does not require new header fields. All standard fields are used.

The QUERY method is followed by the URL describing the location and identification of the agent that is to be queried and the ATP version. When a remote system receives such a message the facilitator is aware that one of the agents (identified by the URL) in the execution environment has been asked to provide status information. If the agent is available the status information is returned in the response body along with the response code 100. Otherwise the particular agent cannot respond for some reason in which case an error response code is sent without a body. Figure 3.16 illustrates a simple query to a remote agent. Figure 3.17 illustrates the timing diagram for an agent query.

<i>Querying a remote agent</i>	
<u>request (query remote agent) -----></u> DATE: Sat, 12 Jan 1997 09:03:00 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: ben@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 61 QUERY atp://some.machine.com# remote.machine.com%231%323 ATP/0.1	<u>Response (send status info)</u> DATE: Sat, 12 Jan 1997 09:04:23 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 491 ATP/0.1 100 OKAY Status data Status data ends.

Figure 3.16 Querying a remote agent for status data using the QUERY method

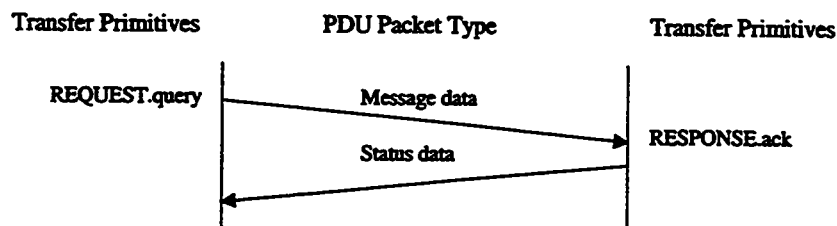


Figure 3.17 The timing diagram for an agent query

Agent Children

Since mobile agents may have the opportunity and power to create child agents (either “clones” of themselves or child agents that are different but subservient like slaves) the transfer mechanism must offer the related functionality. The important issues are:

- dispatching a child of an agent to execute in parallel with the parent which remains where it is
- ordering the termination (and possibly the subsequent return) of all child agents related to a particular parent

Agent cloning introduces many complicated problems such as coordinating children, lost parents, committing of transactions, etc. While these issues must be handled directly by facilitators and parent agents, the transfer mechanism should acknowledge and support their existence. The dispatch of a child agent occurs when an executing agent wishes to send a child to a remote node to perform a task for it on its behalf. The child may be an exact duplicate or not (this is similar in functionality to the DISPATCH method). Transferring a message to a remote facilitator to terminate all child agents occurs when an executing agent that has previously sent children to other nodes has finished successfully and does not require full results from its children. This is similar to the REMOTE_MSG method where one agent sends only a short message to another facilitator.

NEW HEADER FIELDS	REQUEST	RESPONSE
	SLAVE	
ONTOLOGY*	yes	yes
ONTOLOGY-LOCATION*	yes	yes
PARENT-ID	yes	yes
PARENT-LOCATION	yes	yes

Table 3.7 The SLAVE method and the headers that support it. Headers marked with “*” have been previously defined. All other standard headers are required.

In request mode when SLAVE is used by one agent to send a child to another node, the ONTOLOGY and ONTOLOGY-LOCATION fields are self-explanatory. The PARENT-ID field provides the local identifier of the parent (if available) and the PARENT-LOCATION provides the URL of the parent agent. The body of the message contains the child agent. When SLAVE is used to terminate child agents only the PARENT-ID and PARENT-LOCATION fields are required and there is no body to the message. Figures 3.18 and 3.19 illustrate.

<i>Dispatching a child of another agent</i>	
<u>request (dispatch child)</u> -----> DATE: Sat, 12 Jan 1997 09:03:00 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: sysadmin@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 4432 ONTOLOGY: agentsys-documents ONTOLOGY-LOCATION: atp://.../ontologies/mm-doc.ont PARENT-ID: 090943 PARENT-LOCATION: atp://some.machine.com# ben.some.machine.com%334%434 SLAVE atp://remote.machine.com ATP/0.1 agent data agent data ends.	<u>response (acknowledge message)</u> DATE: Sat, 12 Jan 1997 09:03:23 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 15 ONTOLOGY: agentsys-documents ONTOLOGY-LOCATION: atp://.../ontologies/mm-doc.ont PARENT-ID: 090943 PARENT-LOCATION: atp://some.machine.com# ben.some.machine.com%334%434 ATP/0.1 100 OKAY

Figure 3.18 Dispatching the child of an agent to a remote node using SLAVE

<i>Terminating the children of an agent</i>	
<u>request (terminate children)</u> -----> DATE: Sat, 12 Jan 1997 09:03:00 GMT SENDER-APPLICATION: AgentSys-facilitator FROM: sysadmin@sol.genie.uottawa.ca AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 87 PARENT-ID: 090943 PARENT-LOCATION: atp://some.machine.com# ben.some.machine.com%334%434 SLAVE atp://remote.machine.com ATP/0.1	<u>response (acknowledge message)</u> DATE: Sat, 12 Jan 1997 09:03:23 GMT RECIPIENT-APPLICATION: AgentSys-facilitator AGENT-SYSTEM: AgentSys AGENT-LANGUAGE: AgenTransact CONTENT-TYPE: text/plain CONTENT-ENCODING: CONTENT-LENGTH: 15 PARENT-ID: 090943 PARENT-LOCATION: atp://some.machine.com# ben.some.machine.com%334%434 ATP/0.1 100 OKAY

Figure 3.19 Terminating child agents using SLAVE

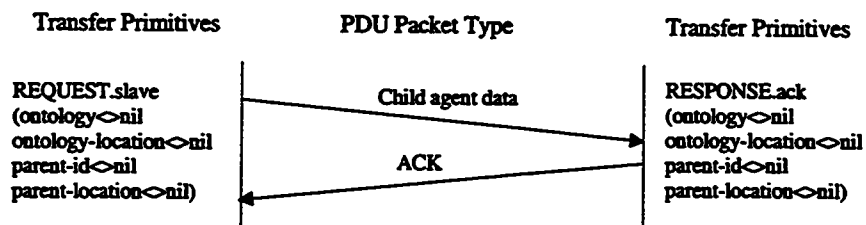


Figure 3.20 Timing diagram for dispatching a child agent

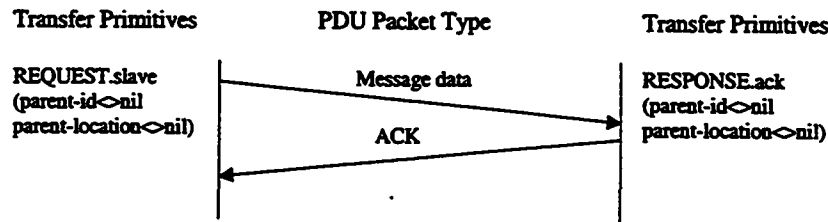


Figure 3.21 Timing diagram for terminating a remote child or SLAVE agent

Figures 3.20 and 3.21 use timing diagrams to illustrate the different semantics of the SLAVE method. The immediate effect of a SLAVE command to terminate children is that the remote facilitator (interpreter, compiler, etc.) will stop parsing/executing the child agents. Whether or not those child agents are subsequently transferred back to the parent with their cargo is dependent upon the children's termination conditions that are programmed at a higher level. If return is necessary and the agent has partial cargo it may be sent back to the parent using the extended DISPATCH method for multimedia cargo discussed earlier.

3.3.2 An Example

In this section, a brief example is illustrated graphically *from the point of view of the transfer protocol and the primitives*. In the example, a mobile agent runs on a user's machine. The agent's task is to find and deliver information about ACME Ltd.'s new line of computer monitors. In Figure 3.22 the mobile agent's behavior is explained from the transfer protocol's point of view. This example shows how the extensions to ATP/0.1 add power and functionality to agent systems. Features such as REMOTE_MSG and SLAVE add what should be considered "threshold" functionality in all agent transference systems.

3.4 Summary

In this chapter, we have proposed two agent-related protocols. *AgentTransact* is an inter-agent communication protocol that enables mobile agents to engage in meaningful conversations for the purposes of discovering, acquiring, exchanging, and manipulating data. This data, often called cargo, represents a solution to the user's task specification. *AgentTransact* is novel

because although it is based on KQML message format it adds new messages and combinations of messages that allow the above-mentioned types of interactions to occur. *AgentTransfer* is the proposed inter-AEE protocol that enables agents to be sent from node to node on a communications network. This protocol is a formal extension to the headers and methods of the IBM ATP/0.1 protocol. Unlike ATP, *AgentTransfer* supports multimedia cargo, ontology fetching, remote messaging, agent querying, and agent slaves.

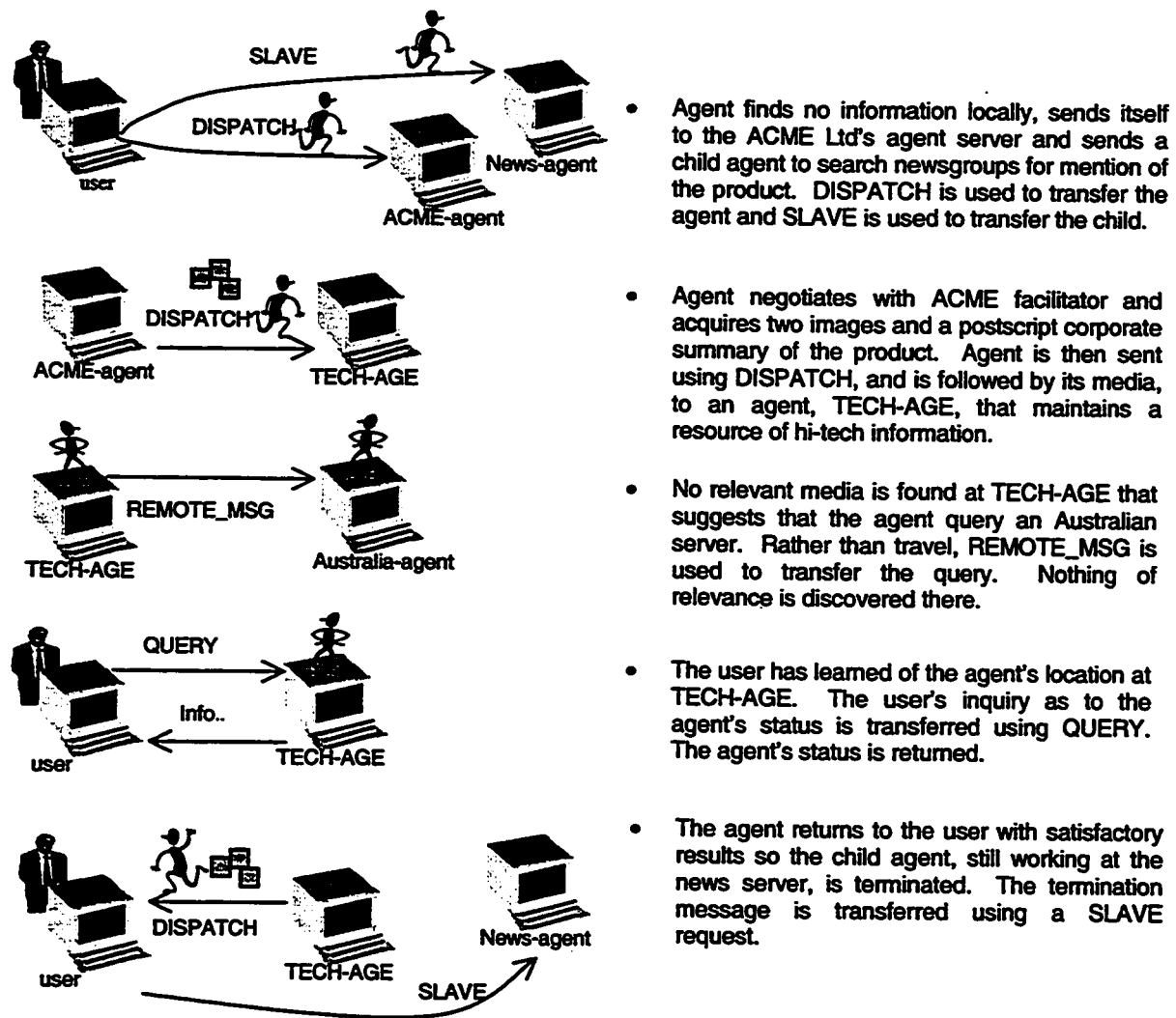


Figure 3.22 The transfer protocol methods utilized in a variety of ways during an agents life

Chapter 4

Agent Execution Environment and Platform

4.1 Introduction

This section describes the platform required for a mobile agent system in which agents are transferred from node to node gathering media for the end user. This platform can be visualized in both the physical and the functional senses and is hereafter often referred to simply as *AgentSys*. The *Agent Execution Environment* (AEE), in which mobile agents execute and are managed and manipulated, is described in detail. This chapter also proposes the *agent-stack*, the agent and facilitator algorithms, the class hierarchies, the “blackboard” communication method, and all of the required supporting structures. Prior to these topics, this chapter first provides a summary of ObjectSpace’s Voyager [VOY97] platform.

4.1.1 Summary of ObjectSpace’s Voyager Platform

Before detailing the proposed functionality of the *Agent Execution Environments* and the specifications of the *Agent-stack*, we will outline the main features and details of the Voyager 2.0.0 (beta) platform. We do this because Voyager formed the basis of our prototype platform and we used its powerful built-in Java classes to expedite the prototyping phase.

Voyager is an advanced, 100% Java, object request broker (ORB) designed from the ground up as a Java-centric distributed computing platform. Its object model is based on the Java language,

simplifying development and reducing time to market [VOY97]. The Voyager platform provides Java 1.1 API's that simplify the creation of distributed applications. A central part of these API's are the mobility and agent-related ones. Voyager implements a number of classes that allow Java objects (e.g. agents) to easily migrate from one virtual machine to another, or, from one object to another, regardless of their physical location. Voyager has recently been compared to its competitors (e.g. Concordia, Aglets, and Odyssey) in *IEEE Internet Computing* Vol.1 No.4 [KIN97] and fared equally well if not better in almost all categories of comparison (which included ease of use, documentation, sample code, and performance). Voyager is also more suitable for mobile agent development than JavaSoft's *Remote Method Invocation* (RMI). Although both are based completely on the Java language, Voyager is a superior desktop development package because it offers many higher level features and API's that RMI simply cannot. For example, remote-enabling a class is a one-step process in Voyager but a more complicated five-step one using RMI. Voyager is Java Beans compliant where RMI is not. Voyager supports object mobility and RMI does not. Voyager has several special purpose classes and API's to support the agent paradigm where RMI does not.

Many of Voyager's unique features will be useful to the *AgentSys* implementation, including the *Agent Inception System* and the *Iconic Modeling Tool*. Its database-independent persistence feature, for example, allows objects to be persisted for later use. Persistence is reduced to a single message, sent to the database. This API will be extremely useful in implementing the *Agent Inception System* which must store and retrieve Agents, LiveAgents, and Itineraries. With respect to mobile agents, Voyager takes care of the Java object serialization and reduces the migration of agents to a single programming line. However, complex transfer semantics could still be implemented on top of these features, and this is the intent with respect to the *Agent Execution Environment* and *Transfer Entity*. Voyager-based agents may also make use of life-spans. An agent can live forever, live until there are no references to it, live for a specified amount of time, or live until a particular point in time. These features will be extremely useful in implementing effective mobile agents.

ObjectSpace is also committed to the Voyager platform for the long term and promises frequent bug-fixes and version updates. Future versions will provide compliance with Common Object

Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM) objects and therefore extend the breadth of the platform with which we may experiment.

4.2 A Layered Agent-Stack

Figure 4.1 illustrates the interconnections that will be required for a mobile agent system and relates the *agent stack* to the OSI stack. Note that the agent protocols need not be positioned below the OSI *Application layer* and so are independent of the underlying format exchanges (e.g. Presentation layer), and transport mechanism (e.g. Session and Network layer à la TCP/IP). The motivation for layering the agent protocols into three distinct layers emerges from the same principles used in defining the OSI layers [ISO84]. Some of these principles are [STA94]:

- Create a boundary where it may later be useful to have the interface standardized
- Create separate layers to handle functions that are manifestly different in the process performed or the technology involved
- Do not create so many layers as to make the system engineering task more difficult than necessary
- Use localized functions within layers so that the layer could be redesigned or reimplemented to exploit new technologies without changing services expected by adjacent layers

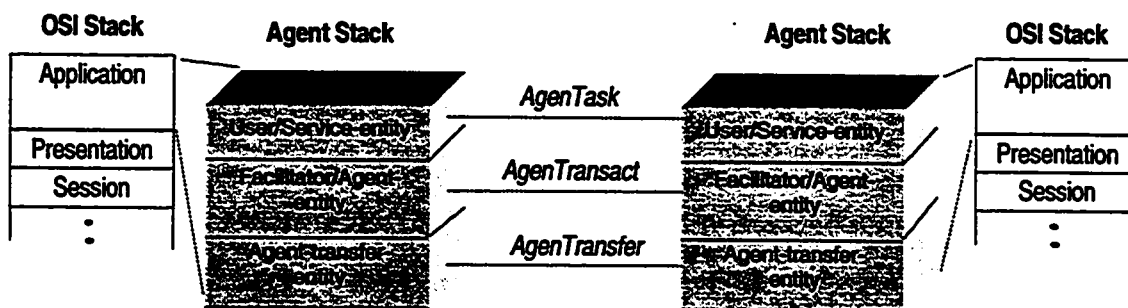


Figure 4.1 A layered view of the protocols necessary in an agent-based system

The *agent-stack's* layers are:

1. **User/service-entity layer:** This layer is concerned with the overall operation of the agent with respect to the application. It is at this level that the user indirectly interacts with the remote service agents and where an application-specific chain of command is created. For example,

physical migrations, cloning, and messaging options are specified. Agents' tasks are defined by the user (or possibly by another agent) using the *AgentTask* protocol. This protocol specifies the general goals of the agent, how it should behave, and what it is allowed to do. Developers may program with the protocol primitives directly while novice users are presented with a GUI that encapsulates the *AgentTask* protocol and offers pre-packaged semi-customizable agents.

2. **Facilitator/agent-entity layer:** This layer is concerned with the inter-agent message exchanges and consists of three conceptual sub-layers. The ontological layer is the lowest sub-layer. Single speech-acts are the next highest sub-layer and conversations, which govern the complex interactions and negotiations between agents, are the highest sub-layer. All communication between agents occurs through facilitators who maintain conversation integrity, log transactions, etc. This layer provides API's to support inter-agent communication to the *User/service-entity* layer. The communication protocol is called *AgentTransact*.
3. **Agent-transfer-entity layer:** This layer is concerned with the mechanisms and primitives that transfer agents, their messages, and cargo between systems. This includes headers, methods, formats, etc. It is invoked and used by upper layers to transparently route agents to remote nodes. It also provides error-free transmission of agents and packages their protocols and conversations to remote sites in such a way so as they may be continued or reused. It provides primitives to the *Facilitator/agent-entity* layer. The transfer protocol is called *AgentTransfer*.

In this thesis it is assumed that there is a network (Internet or LAN), human users with the freedom to program and insert agents onto the network, and service providers who offer various services to the human and agent community (see figure 4.2). Examples of services are WWW media stores, video or music-on-demand, and agent meeting places. Mobile agents, working on behalf of an authorized user, roam between *Agent Execution Environments* (AEE's) on the network in search of media that satisfy a user request. The user specifies a high-level task (e.g. itinerary) with the *AgentTask* protocol. Local transfer agents, using the *AgentTransfer* protocol over TCP/IP, transfer mobile agents between AEE's. Arriving agents are also accepted by transfer agents, unpacked and passed on to a facilitator agent that creates a "virtual machine" environment for the incoming agent. Facilitators manage local resources and serve as the only

gateways through which incoming agents may access the resources. Using *AgentTransact* messages, agents communicate with other mobile agents either locally or remotely and ask for services from local facilitators. Agents have no direct access to media resources.

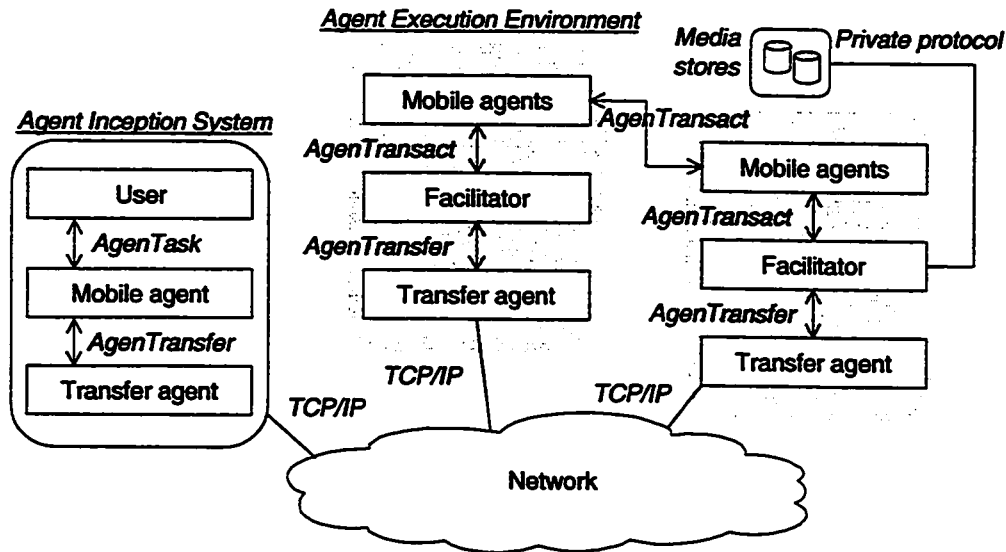


Figure 4.2 The physical architecture and protocols of AgentSys. Shaded areas are *Agent Execution Environments*.

4.3 The Agent Execution Environment (AEE)

The AEE is divided into several functionally distinct modules. These modules share responsibility within the AEE and communicate with each other in order to create a value-added environment in which agents not only execute but are managed, validated, and are rendered services. Almost all modules use local disk space to create caches, tables, or databases for the purpose of storing persistent data relating to their functional specification. Physically these data-stores are not considered a part of the AEE (and how they are implemented is not an important feature here – we assume they will use the Voyager interface). The facilitator is the coordinator of the system in the sense that it occasionally ensures that all other modules are operational, and in the sense that it serves as the gateway to all local resources and delivers messages to mobile agents that it hosts.

The AEE is a suite of Voyager [VOY97] agents that may communicate with other remote agents or even migrate should the need arise. The AEE and its components are implemented with the

Java [SUN94] programming language. Components of the AEE can be considered “agents” in the sense that they exhibit most of the defined requirements of agenthood from Chapter 1, including the facts that they:

- are autonomous and able to make their own decisions based on logic, inferred, or explicit information
- may delegate work to other agents
- co-operate to achieve the end-user’s goal
- both serve *and* request data – e.g. they may be both clients and servers of data
- message in a peer-to-peer fashion
- may migrate to other nodes and preserve state at instance-variable granularity (though this is not usual for AEE components)

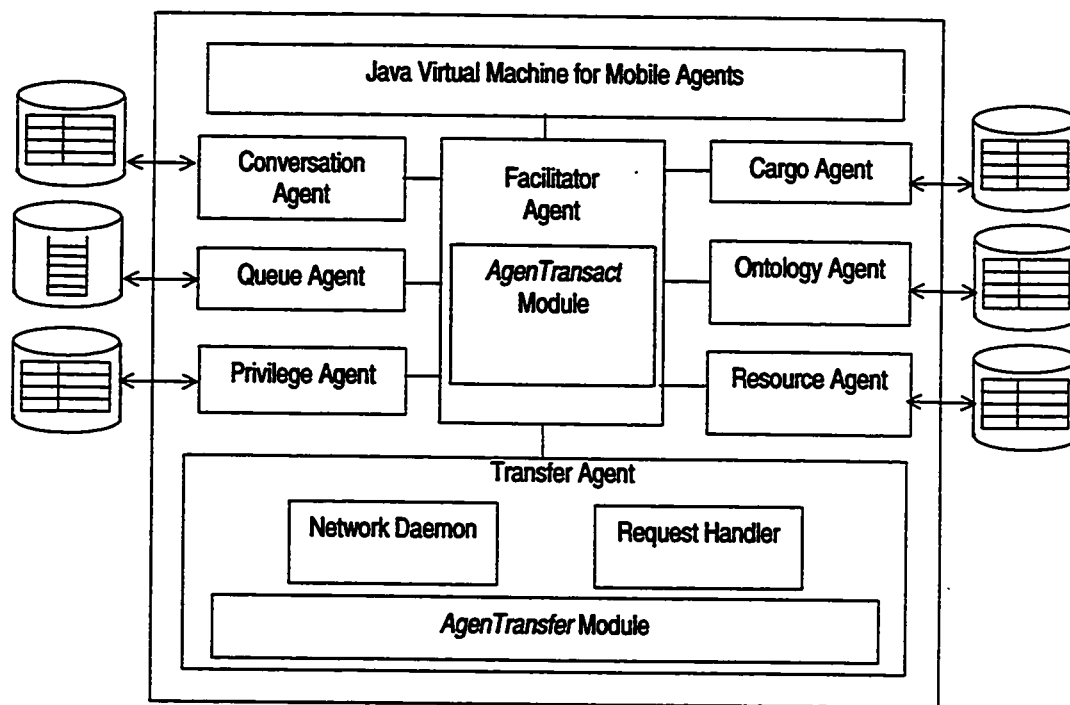


Figure 4.3 The modules of the Agent Execution Environment. Each module has distinct functionality and is essential to an operational environment for incoming agents.

Figure 4.3 illustrates the agent components that create the AEE environment. Note that some of the agent components manage databases of information. These databases are implementations of Voyager’s database `com.objectspace.voyager.db.DB` interface and allow AEE agents to easily add, remove, and query persistent objects.

The *Facilitator Agent* is the main entity in the AEE. Mobile agents request services and communicate with other agents through this *gateway*. The facilitator has a well-known port number through which it can be reached and a well-known Voyager handler – namely *facilitatorAgent* (this is the way it is most often reached). It is the role of the facilitator to occasionally ensure that the other modules are operational. The *AgentTransact* module implements the protocols necessary for inter-agent messaging and conversations and is based on KQML. The *Cargo Agent* organizes agent media. Incoming mobile agents may arrive with, or subsequently acquire, media. When an agent arrives and is followed by media it is given a handler to the media which are physically placed on tertiary storage such as a hard-drive. The actual location and schema of the stored media is transparent to the mobile agent. The salient point is that when the agent subsequently jumps to another node, the data manager maps the agent identifier to its media and then gives them to the transfer module. If the storage area that the data manager handles becomes too full, a message is sent to the transfer module. In this case, incoming agents with cargo exceeding a certain number of bytes may be rejected since the persistence of their cargo cannot be guaranteed. When mobile agents acquire media they are added into the storage area and associated thereafter with the agent. It is also the case that the acquisition of a media may be rejected if that media will overflow the agent's individual storage space restrictions. The overall sizes of the data area and individual quotas vary from AEE to AEE. The *Resource Agent* monitors local resources. As a set of processes, the AEE shares the local resources with other processes that may be active. The node hosting the AEE may be a stand-alone server or a workstation. In both cases other processes that may be active include user-sessions, print jobs, application programs, or other mobile agents. The local resource agent's role is to collect and monitor information on CPU load, disk-space, and other critical aspects. This module sends interrupts to the facilitator module when, for instance, high CPU usage by other more critical processes must pre-empt the interpretation of a mobile agent. In contrast to the agent data manager that creates a logical array of media "slots" for storage of cargo, this module monitors the physical resources. In this sense, the AEE monitors itself to ensure that it does not overload the resources on the node on which it exists. The *Ontology Agent* organizes and persists ontologies. Ontologies are used in order to communicate and to provide a shared understanding of terminology. Because of their hierarchic nature, they can be extended using specialization (as in the object-oriented paradigm) in which case new versions

are created. Individual ontology specifications and their versions are stored on disk and must be managed by the ontology agent. A remotely operating agent may require an ontology (specified by *name* and *version*) be sent to it – in this case the request is received by the transfer module which asks the ontology agent for the data. The data is sent to the transfer module and then sent on its way over the network to the remote transfer module. Ontologies may be stored using any representational syntax as long as the facilitator can interpret that representation; for example, *Ontolingua* [GRUB93] is used in *AgentSys*. The *Conversation Agent* parses inter-agent messages. During the course of conversations, agents send messages to each other through the facilitator module. These messages are passed to the conversation agent whose role it is to validate the messages in the context of the conversation, and then log them. The conversation agent has a small knowledge base that describes the conversation protocols. Messages are checked for integrity using these rules. Logs are made in the form of relations, describing for each agent the conversations it engaged in and with whom, the individual messages that composed them, and timestamps. When the agent's user requests a detailed record of agent activity the facilitator requests the appropriate logs. This module also stores extensions to the *AgentTransact* protocol when necessary. The *Queue Agent* manages a queue of agents. When incoming mobile agents cannot be executed immediately they are placed on a waiting list. This occurs when an agent happens to arrive for execution at a time when the Resource Agent deems that the CPU is too heavily loaded (if there is insufficient storage for the agent's cargo then the agent is rejected regardless of CPU load). The queue agent reports the current queue contents to the facilitator. This data includes the agent identifier (for local purposes), the agent's task type, human-user identification, and a handler into the Cargo Agent's resources for that agent. This allows the facilitator to answer queries regarding the agents in the queue as well as active ones (e.g. a mobile agent might ask the facilitator, "Are there any other agents, active or queued, with a similar task-type?") The *Privilege Agent* manages account numbers. Incoming mobile agents must be authenticated. The local AEE maintains a list of human users and their authentication. Furthermore, agents acquire privileges to use certain resources. For instance, a mobile agent may have privilege to use access web documents but not asynchronous documents. The privilege agent stores records that indicate *what* the agent may access and stores securely the *cookies* (passwords). These passwords must match those that are encrypted within the agent's structure. *Virtual Machines* are required for execution – regardless how agents are implemented

(e.g. declaratively or procedurally), they require a virtual machine in which to run. This is typically an interpreter (e.g. LISP, Java etc.), forked by the facilitator, that processes the agent's executable code line by line. Depending on the language, agents may be able to create local variables, open files for reading, etc. Typically however, the virtual machine represents a "safe" (from the local machine's point of view) place in the sense that agents are forbidden from performing operations that may lead to malicious or mischievous activity such as opening files, writing to storage, or using pointers. The owning agent has access rights to its own cargo, but that is the only case in which an agent has direct access to a local file's content. All other access is through facilitators via authenticated conversations. The *Transfer Agent* sends agents and data between AEE's. The transfer agent is a set of sub-modules that handle the transfer and reception of mobile agents between nodes. Each node has a Network Daemon that uses a well-known port number upon which it listens for activity. When it detects activity it reads the message and sends it to the *AgentTransfer* module that implements the transfer protocol and its syntax. If the message is not well formed or has been corrupted the *AgentTransfer* module sends the appropriate error code. Otherwise, the message is given to the Request Handler which examines the primitive and the parameters (e.g. DISPATCH, CLONE, etc.) and then does one of many things. If the incoming agent has cargo to follow, a local identification is made for that agent and given to the Cargo Agent. When each piece of cargo subsequently arrives, it is given to the Cargo Agent who stores it appropriately. The Request Handler thus implements a sort of "session" in which cargo arrives piece by piece. The Request Handler passes validated incoming agents to the Facilitator for interpretation.

Figures 4.4a and 4.4b illustrate the class hierarchy of the AEE agents and show the "usage" relationships between the AEE agents and the AEE databases as well as the patterns between agents themselves. A line from class A to B with "uses" close to class A means that A "uses the services" of class B. For the databases, this relationship tends to be 1-to-1 but in the case of inter-agent usage we see that the Facilitator indeed "runs the show" since it must use the services of many of the other agent managers.

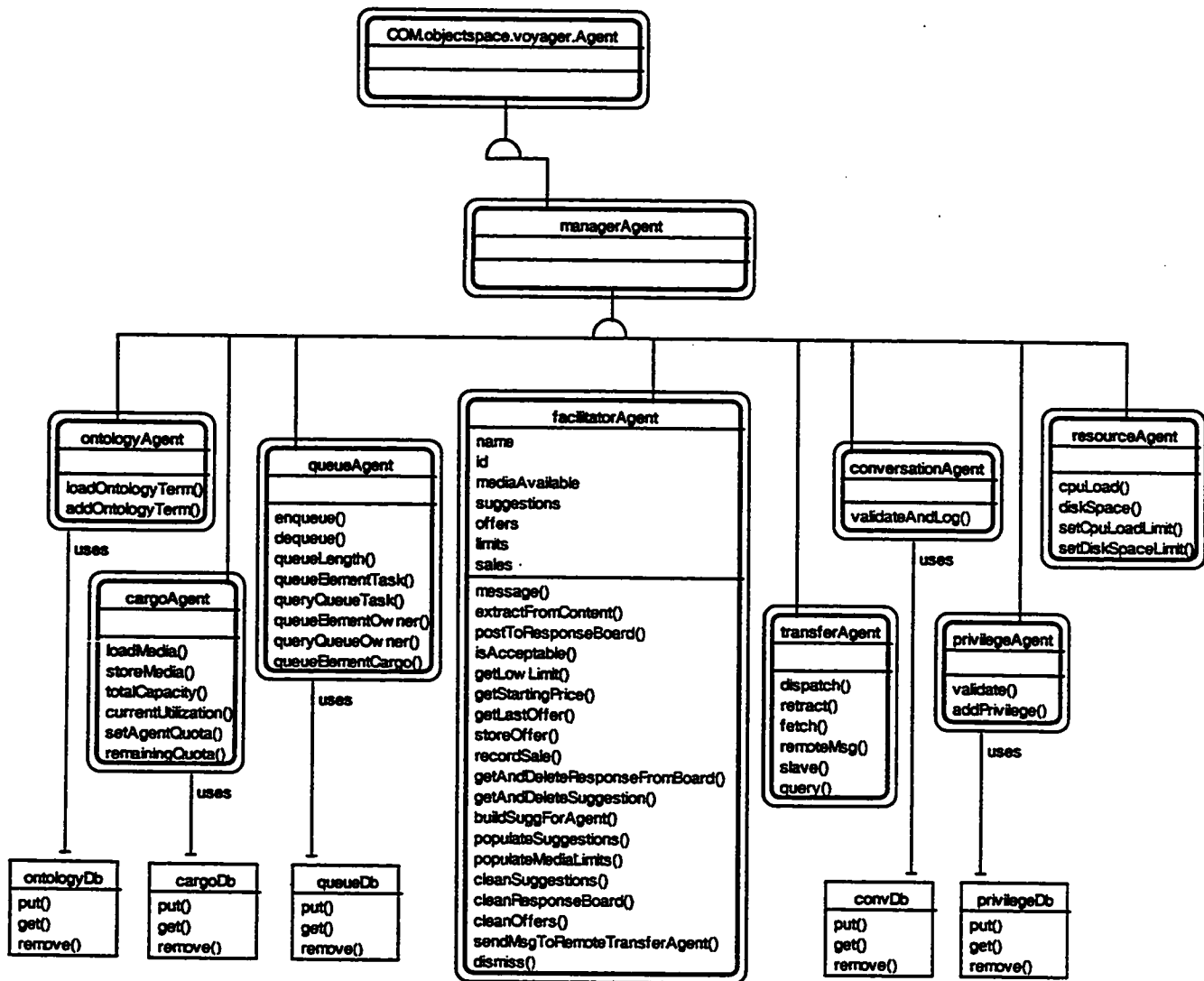


Figure 4.4a) The organization of classes and their methods that implement the AEE. Usage pattern for AEE databases is also illustrated.

The Facilitator provides dynamic services and information to incoming mobile agents. The important methods that each AEE agent implements are listed. The Resource Agent has methods to check or set CPU load and disk space limits. The Ontology Agent may verify that a term is in the ontology database and then return the term, or add a term to such a database. The Cargo Agent can both load and store media from and to the cargo database. It can also provide status on the cargo area including the total capacity, current utilization, and remaining quota for a given agent. It may also be used to set a default disk quota for a particular agent. The Conversation Agent validates and logs messages sent in the *AgentTransact* protocol.

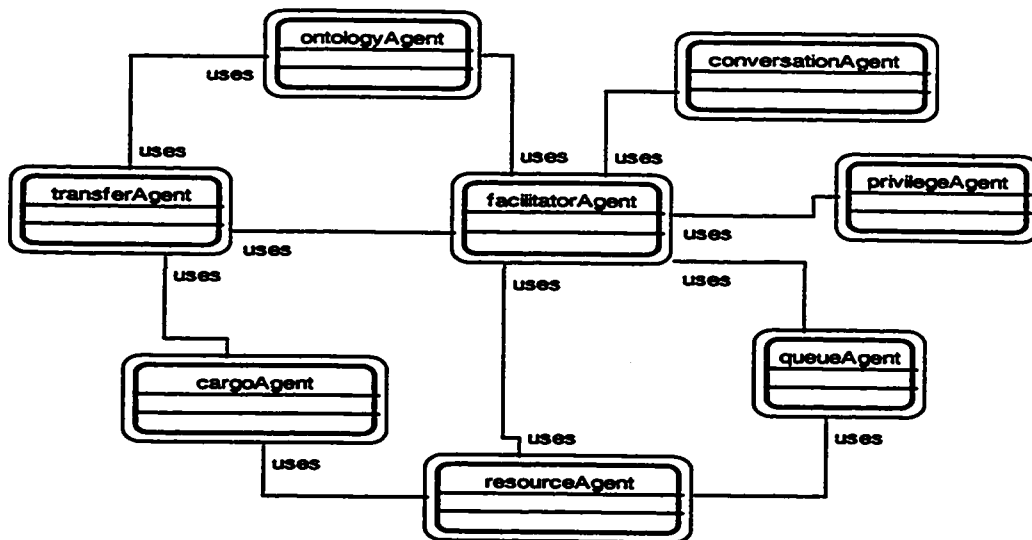


Figure 4.4b) Usage diagram for the AEE entities

The Queue Agent may enqueue or dequeue an agent from the queue database, or indicate the current length of the queue. It may also provide task or owner information about any queue entry (e.g. *QueueElementTask(2)* returns task data for the 2nd element in the queue) or query the cargo of any queue element.

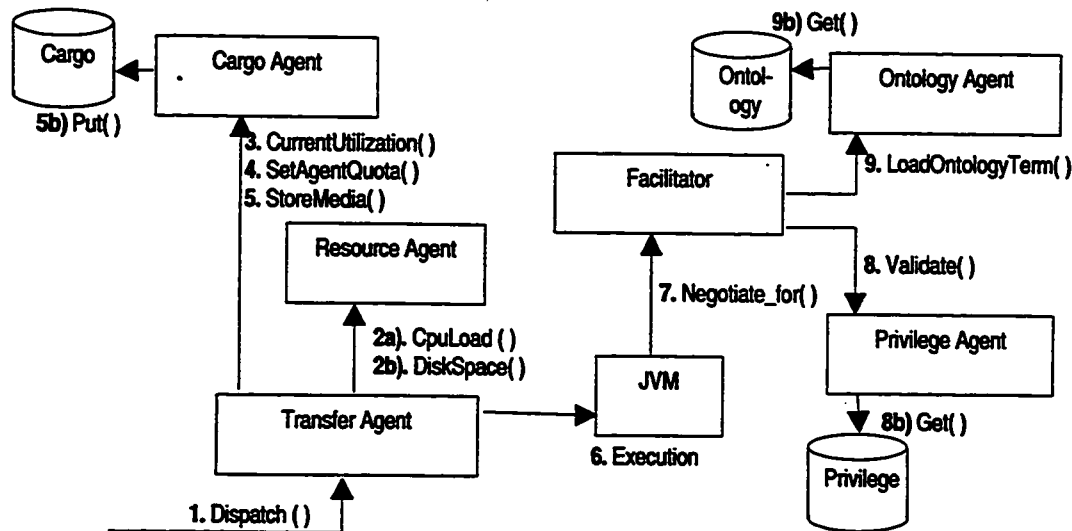


Figure 4.5. An arriving mobile agent and the behavior of the AEE

The Transfer Agent implements the *AgentTransfer* methods and provides an Service Access Point (SAP) for Facilitator agents into the *AgentTransfer* layer. The Privilege Agent can validate a

user to verify permissions or may add permissions into the user's file. It maintains a database of privileges.

4.3.1 Example Scenarios

In this section some examples of how the agents within the AEE work together to complete tasks is provided. Figure 4.5 illustrates how the AEE behaves when a mobile arrives for execution. The Transfer Agent (the remote system uses a Dispatch to send the agent to this AEE) first receives the agent. The steps that then might occur are as follows:

1. The Transfer Agent (TA) receives the incoming mobile agent from the byte stream, as well as an indication of whether or not cargo will subsequently be arriving.
2. The TA checks with the Resource Agent (RA) to determine if there is currently enough space or time for the incoming mobile agent. It does so using the RA's *CpuLoad* and *DiskSpace* methods.
3. The TA then checks for the current utilization of the cargo disk space using the Cargo Agent's (CA) *CurrentUtilization* method.
4. The Transfer Agent sets a default quota for this agent with the CA's *SetAgentQuota* method.
5. The incoming mobile agent's cargo are now received and stored in the Cargo database.
6. The mobile agent begins to execute in the Java Virtual Machine (JVM).
7. During the course of execution, the agent wishes to negotiate for some entity (e.g. media). It sends the appropriate *AgentTransact* message (*negotiate_for*) to the Facilitator.
8. The agent is validated using by the Facilitator via the Privilege Agent's *Validate* method. The Privilege Agent looks up the appropriate information from its database.
9. A term is used in the message to the Facilitator that it does not understand. It then attempts to find this term by using the Ontology Agent's (OA) *LoadOntologyTerm* method. The OA then tries to acquire this term from its database.

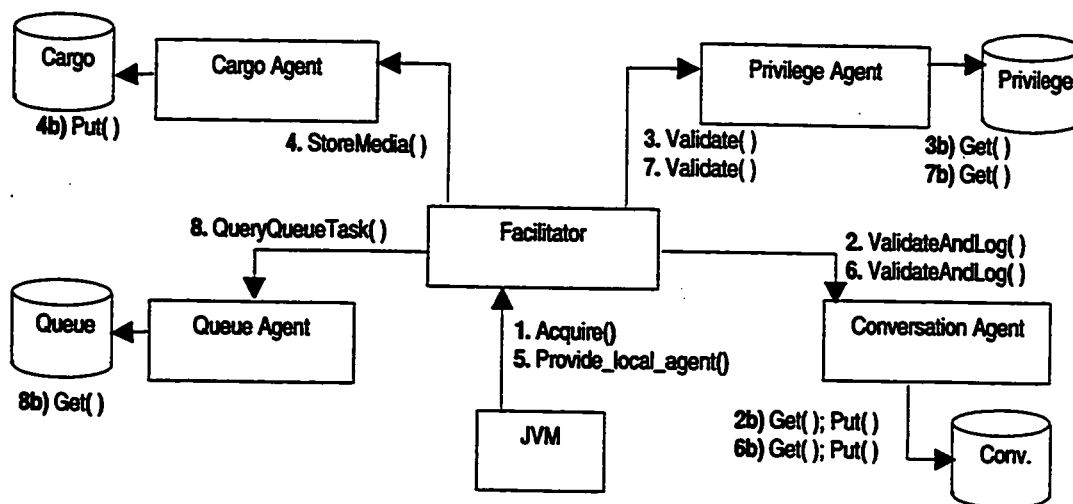


Figure 4.6 An executing mobile agent and the operation of the Facilitator and other AEE agents.

In figure 4.6, some important steps that occur during agent execution are illustrated and explained. In this example, the mobile agent has arrived and is executing within the JVM. During execution it messages with the Facilitator, which in turn requests services of the other supporting agents of the AEE.

1. The executing mobile agent requests the acquisition of a named media using the *Acquire* method.
2. The Conversation Agent at the request of the Facilitator (using the *ValidateAndLog* method) analyzes the context of the *Acquire* message within the current conversation. The Conversation Agent uses its database to verify the syntax and then stores a log entry.
3. The Facilitator checks with the Privilege Agent to make sure that the named media are legally accessible by the current agent. The Facilitator uses the Privilege Agent's *Validate* method to do this and the Privilege Agent then uses its own database lookup.
4. If the mobile agent made a correct request and was authenticated then the Facilitator stores the acquired media in the mobile agent's cargo bay. It does this by using the Cargo Agent's (CA) *StoreMedia* method. The cargo is stored by the CA using the *Put* method.
5. In a subsequent conversation, the mobile agent asks for a handler to another locally executing agent that it may want to message with. It uses the Facilitator's *Provide_local_agent* method to do so.
6. The message sent to the Facilitator is validated and logged by the CA.
7. The Privilege Agent uses *Validate* to verify the appropriate privileges.

8. The Facilitator goes ahead and queries the Queue Agent (QA) for any other agent's in its queue that match the current one with respect to task information. The QA checks its database of Agents using *Get* and responds either positively or negatively.

4.4 The Mobile Agent, Facilitator and Blackboard Mechanism

We have proposed a layered agent-stack consisting of three levels of protocols. In order for mobile agents to achieve their goals they must interact with Facilitators. Such interaction utilizes the *AgentTransact* protocol and is achieved transparently via a “blackboard” mechanism implemented by the Facilitator. This section outlines the operation of mobile agents (that gather information based on user parameters), Facilitators, and describes the blackboard mechanism. The full pseudo-code for all agents and Facilitators can be found in the Appendix A.

The Mobile Agent

The *MobileAgent* class is a child of Voyager's *Agent* class (see figure 4.7). This therefore gives it the ability to migrate, via the *moveTo()* method, and other capabilities such as forwarding and termination control. Mobile agents follow their pre-specified itineraries to the full extent possible. In the case of asking a Facilitator for server suggestions the itinerary is dynamically updated – in any case, however, the agent always tries to interpret the itinerary correctly. Mobile agents are instantiated with three variables: *name*, *home_IP*, and *Itinerary*. The *name* variable is derived from user input (e.g. “call this agent instance ‘Freddy’”), the *home_IP* variable is inferred automatically by the Voyager system if not explicitly specified, and the *Itinerary* variable names and points to a saved itinerary. The agent also has a number of instance variables that it always tries to set correctly. These include *name*, *homeIP*, *itinerary*, *addressBook*, *userName*, *email*, *city*, *interests*, *topic*, *author*, *date*, *LAST_CONV*, *LAST_MEDIA*. Most of these are self-explanatory except: *addressBook* which is a Vector containing the places to visit, ordered in sequence; *interests* which is a list of textual keys describing the users general interests; *LAST_CONV* which is the last conversation engaged in by this agent; and *LAST_MEDIA* which is the last media acquired by this agent.

Methods important to the mobile agent's behavior are: *launch()*, *next()*, *atAEE()*, *parseItinerary()*, *helpWithDocConversation()*, *negotiateFor()*, *askForConversation()*, *dismiss()*.

The *launch()* method is invoked by the AIS to begin the agents lifetime. *Next()* loops through the addressBook and if addresses remain to which to migrate it invokes the native Voyager *moveTo()* method to achieve this. Whenever the agent arrives at a new AEE it invokes its own *atAEE()* method. This method sends a message back to the home AIS (to update the GUI – see StatusMgr agent) and then invokes the *parseItinerary()* method to parse the sub-itinerary intended for this AEE. In the process of parsing the Itinerary the agent invokes other methods depending on the nature of the itinerary.

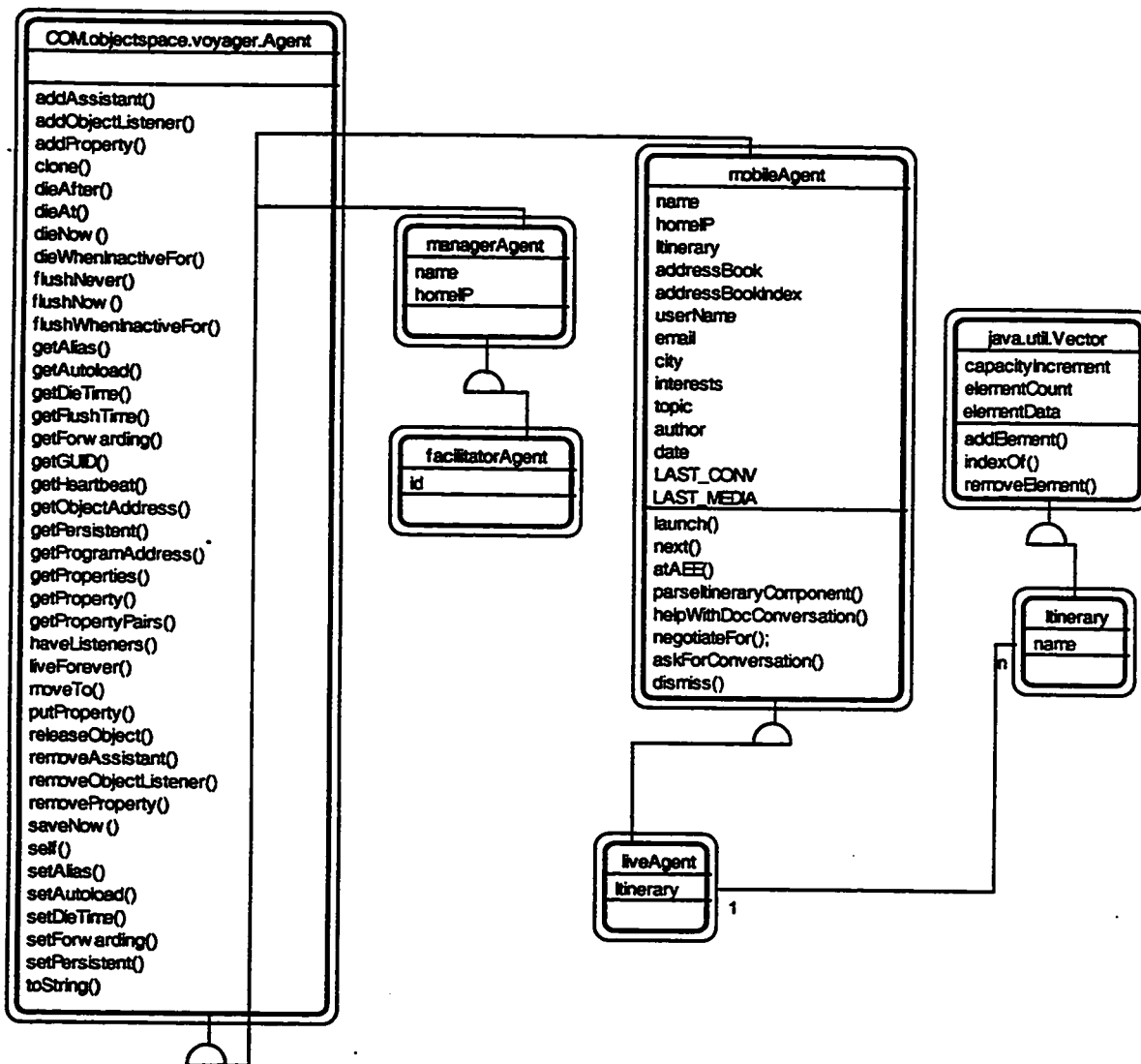


Figure 4.7. The Agent class hierarchy, relationships, and the Itinerary.

For example, if an “ACQUIRE=document” is caught the *helpWithDocConversation()* is triggered. This method implements the *AgentTransact help_with_doc* conversation protocol from the agent’s point-of-view. When an agreement is made between an agent and Facilitator to accept a media suggestion the mobile agent then triggers the *negotiateFor()* method (see section 3.2.2). This method implements the *AgentTransact negotiate_for* protocol in which a final price for an entity is either decided upon, or one of the parties declines. The *dismiss()* method can be invoked by the end use (e.g. kill agent immediately) and effectively terminates the agents thread.

The Facilitator

The Facilitator (see figure 4.7) is usually more complex than typical incoming mobile agents (though not as a rule). Since it must also manage and coordinate other agents, track and validate information, and pass media and data to mobile agents, the Facilitator has more methods and instance variables. As specified in the ontology, the Facilitator has an *id* variable as well as: *name*, *mediaAvailable*, *suggestions*, *offers*, *limits*, *sales*. The name variable can be used to give the facilitator a human-readable name, though in our scheme this is not necessary since each Facilitator is started with a standard Voyager “handler” – e.g. *FacilitatorAgent*. Thus any agent need only know the IP address of an AEE since the Facilitators name is well-known. *MediaAvailable* is a boolean variable that the Facilitator uses to quickly answer media queries. If media are not available it is set to false, and visa versa. *Suggestions*, *offers*, *limits*, and *sales*, are internal tables that track those important commitments and data. *Suggestions* records any suggestion made by a Facilitator to an agent, *offers* maintains a table of offers pending to agents (e.g. an offer was made to sell media *m* to agent *a* for *x* units, at time *t*), *limits* maintains a table of price limits for specific media (e.g. lowest acceptable price, starting price for negotiations, etc.), and *sales* records final sales for media acquisitions (e.g. media *m* was sold to agent *a* for *x* units).

The facilitator is not instantiated with any variables. Upon startup however, a Facilitator populates the media databases with whatever information is necessary. For example it may read a file of pending pricing changes to be made and it will go ahead and make the changes to ensure that it uses an up-to-date *limits* database. The facilitator is mostly passive, only using other agents services from time to time. The main method is the *message()* method which is a well-

known method-name through which mobile agents send *AgentTransact* messages to the Facilitator. Such messages are extracted using the *extractFromContent()* which strips away extraneous data and leaves only the KQML-like message to be interpreted. Depending on the performative used and the intent of the message any of a number of things happen. If it is a *receive_msg* request then the appropriate message is extracted from the blackboard using *getAndDeleteResponseFromBoard()* and returned. Otherwise the performative is usually a part of a conversation protocol and the action taken depends upon the protocol. This thesis implements the *help_with_doc*, *negotiate_for*, and *ask_for* conversations. The specification and operation of these protocols are discussed in Chapter 3 and the remaining Facilitator pseudo-code can be found in Appendix A.

Other important methods within the Facilitator class are: *buildSuggForAgent()*, *isAcceptable()*, *getLowLimit()*, *getStartingPrice()*. The first of these, *buildSuggForAgent()* is invoked by the Facilitator when an agent asks for help with a media task. Upon first receiving such a request the Facilitator builds a list of suggestions based on the query (e.g. suggest images of Clinton if the task keyword was "American politics"). During the course of the ensuing conversation the mobile agent asks for one or more suggestions – to accommodate such requests the Facilitator reads from the suggestions list until the list is exhausted. The *isAcceptable()* method is used by the Facilitator to determine if an offer for media is acceptable or not – e.g. whether or not to return a counter-proposal (e.g. a higher price), a decline message, or an acknowledgement message. In the current implementation, this method is not given much intelligence and will simply counter-propose until a pre-set price is reached for the particular media. The *existence* of this method, however, allows future agent child classes to override the simple mechanism with more complicated models (e.g. supply-demand, etc.). The *getLowLimit()* method is used to determine the lowest allowable give-away price for a particular media object, and the *getStartingPrice()* method is used to determine the starting bidding price for a particular media object. Other miscellaneous methods are listed in the class hierarchy figure.

The Blackboard Mechanism

To implement the asynchronous message-passing paradigm for inter-agent communications in *AgentSys*, a blackboard conceptualization was used. This means the following: Whenever the

Facilitator F responds to a message from agent A , the response R is posted onto a blackboard. The response R is *not* sent directly to A by invoking a method on A . This frees agent A from having to implement a named method and semantic. When agent A wishes to receive the response to its message to F it uses the well known “message” method on F and sends the “receive_msg” performative as data. F receives the message and tries to find the response from the current postings on the blackboard. If the response R is available it is sent back to the agent, otherwise a null message is returned (see figure 4.8). This effectively creates a non-blocking, asynchronous, messaging environment and is desirable because:

- The agent does not block while the Facilitator computes the response and so it is free to perform other computations that may be relevant to the response. Furthermore, some queries on Facilitators may be time intensive (e.g. help with this task by suggesting media) or even indefinite (e.g. monitor for the presence of a condition), making this aspect more crucial.
- It puts the agent “in charge” of the exchange of messages. This coincides with the desire to have agents that exhibit a degree of autonomy (see chapter 7). It is the *agent* that decides when to receive its next message – *not* the system.

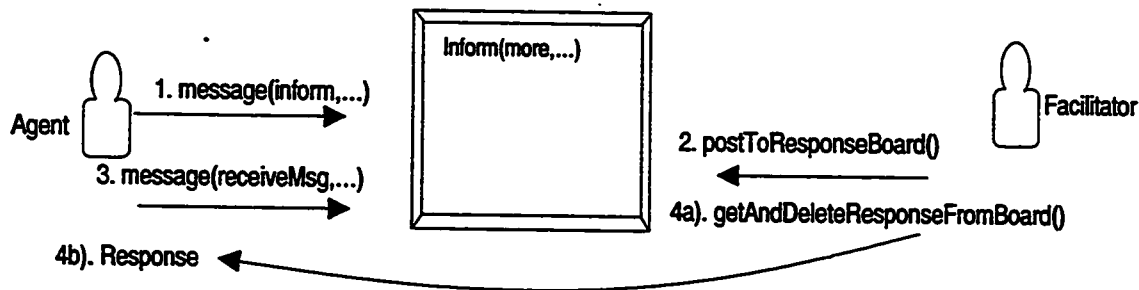


Figure 4.8 The blackboard mechanism for agent communication. To send a message to the Facilitator an agent invokes that Facilitator’s (well-known) *message* method and passes an *agentMessage* instance. The *agentMessage* structure corresponds directly to the *AgentTransact* (e.g. KQML) message structure.

4.4.1 Supporting Structures

During the process of message exchange the AEE and mobile agents use several key data structures (implemented as classes). Figure 4.9 illustrates. The structure used by both Facilitator and agent, and the most important structure, is the *agentMessage* class. This class has no

methods except `toString()` which simply returns a `String` representation of the instance variable values. The instance variables of *agentMessage* are *performative*, *ontology*, *sender*, *receiver*, *in_reply_to*, *reply_with*, *content*, and *policy*. Note that this structure holds the KQML-like message [FIN97] – the variables are the KQML parameters of the *AgentTransact* message and correspond to a subset of standard KQML parameters. Whenever an *AgentTransact* message is sent it is encapsulated in an instance of *agentMessage*. Either the agent or the Facilitator, depending on who is sending, sets the value of these parameters. The parameter names explain the functionality of them parameters, or see Chapter 3 for further documentation.

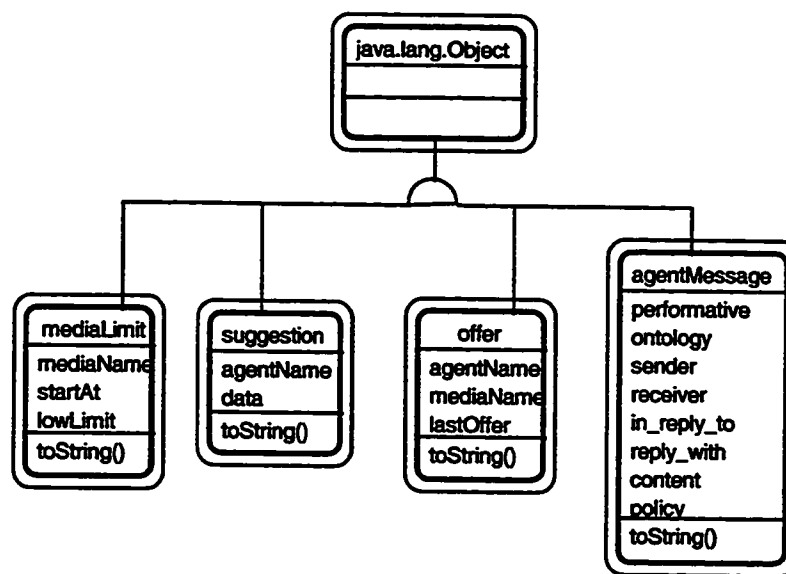


Figure 4.9 Class hierarchy of supporting structures used by Agents and Facilitators in *AgentSys*

There are three other supporting classes that play a critical role in AEE and Facilitator operation. The *mediaLimit* class is used by the Facilitator to track and associate media (named in *mediaName*) to their starting prices and lowest price limits (*startAt* and *lowLimit* respectively). Before making its first proposal for a media price, the Facilitator refers to the *mediaLimit* instance corresponding to that media. The *suggestion* class is used by the Facilitator to store suggestions that it has or will make to mobile agents. When an agent asks for help expecting suggestions, the Facilitator creates and fills several instances of this class. The class stores the name of the agent in *agentName* and the data that is being suggested in *data* (e.g. a named media, a named server, etc.). When suggestions to agent *A* are no longer needed, those corresponding to *A* are garbage-collected by the AEE. The *offer* class is used analogously but

for offers. During the process of negotiating (bargaining) to settle a media price many offers are made back and forth. Because of the asynchronous nature of messaging in *AgentSys* the Facilitator must remember the last offer made by a particular agent for a named media (because it may do other processing of other agents before the response is required – e.g. multi-threading). This is the role of the *offer* class. *AgentName*, *mediaName*, and *lastOffer* are the instance variables and are self-explanatory.

4.5 An Example AEE Trace

This section provides a part of a textual trace of an AEE and Facilitator execution. Figure 4.10 illustrates the trace. In this trace a mobile agent (denoted “MA”) asks for media suggestions, negotiates for the prices of media with the Facilitator (denoted “FA”), and then acquires the media. By examining the figure each of the speech-acts of the conversation protocols can be seen (the “MA:” or “FA:” denotes the sending party).

4.6 Summary

We have proposed a three-layered agent-stack to enable a mobile agent platform, and indeed *AgentSys*. The *User/Service-entity* Layer of protocol allows mobile agent tasks to be created visually and textually. The *Facilitator/Agent-entity* Layer enables agents to inter-communicate using a shared understanding of messages and terminology. The *Transfer Layer-entity* enables mobile agents to be transferred between *Agent Execution Environments* (AEE’s) unambiguously. We also proposed the structure of such AEE’s and the functionality that should and must be supported. AEE’s consist of several cooperating agents working together to manage and support all aspects of mobile agent execution and management. AEE agents are implemented as Java objects. Mobile agents that arrive at any given AEE should expect to execute within such an environment but need not be aware of the implementation of the AEE agents (e.g. Conversation Agent, etc.) since they message only with the Facilitator agent. As such however, mobile agents must implement the *AgentTransact* messages that allow meaningful conversations between agents. Since all Facilitators are given the same handler name, *facilitatorAgent*, the Voyager system lets mobile agents easily get this handler at any AEE. Furthermore, by passing an *agentMessage* instance to a Facilitator using its *message* method a mobile agent easily engages

in conversations consisting of the KQML-like *AgentTransact* message-format. From the Facilitators point-of-view all responses to agent messages are “posted” to a blackboard of messages. This blackboard can be accessed only by this Facilitator and represents a pool of answered queries and messages. To receive a response to a message a mobile agent asks for it using the well-known *receive_msg* performative. The correct response (if available) is then received and deleted from the blackboard by the Facilitator. To achieve full operation of the AEE the Facilitator uses several local classes including *offer*, *mediaLimit*, and *suggestion*.

```
voyager(tm) 2.0 beta 1, copyright objectspace 1997
address = 137.122.107.154:8000
```

AgentSys Execution Environment Trace

```
FA: Started.
MA: caught a ADDRESS cmd
MA: caught a STATUS cmd
MA: caught a ACQUIRE cmd
MA: help_with_doc(Lewinsky, Clinton Scandal)
FA: suggest(index.html)
MA: inform(more)
FA: suggest(Clinton.gif)
MA: inform(more)
FA: suggest(cnn.html)
MA: inform(done)
MA: out of help_with_doc conv.

MA: negotiate_for(media, index.html, 10)
FA: ctr_propose(media, index.html, 70)
MA: ctr_propose(media, index.html, 20)
FA: ctr_propose(media, index.html, 60)
MA: ctr_propose(media, index.html, 30)
FA: ctr_propose(media, index.html, 50)
MA: ctr_propose(media, index.html, 40)
FA: recorded an agreement to sell index.html to Freddy for 40 units.
FA: accept(media, index.html, 40)
MA: done negotiating.

MA: negotiate_for(media, Clinton.gif, 10)
FA: ctr_propose(media, Clinton.gif, 80)
MA: ctr_propose(media, Clinton.gif, 20)
FA: ctr_propose(media, Clinton.gif, 70)
MA: ctr_propose(media, Clinton.gif, 30)
FA: ctr_propose(media, Clinton.gif, 60)
MA: ctr_propose(media, Clinton.gif, 40)
FA: ctr_propose(media, Clinton.gif, 50)
MA: accept(media, Clinton.gif, 50)
FA: recorded an agreement to sell Clinton.gif to Freddy for 50 units.
FA: done negotiating. Cleaning.
MA: done negotiating.

MA: negotiate_for(media, cnn.html, 10)
FA: ctr_propose(media, cnn.html, 40)
MA: ctr_propose(media, cnn.html, 20)
FA: ctr_propose(media, cnn.html, 30)
MA: accept(media, cnn.html, 30)
FA: recorded an agreement to sell cnn.html to Freddy for 30 units.
FA: done negotiating. Cleaning.
MA: done negotiating.

MA: acquire(media, index.html, -1)
FA: ACK()
FA: sending index.html to home of Freddy for 40 units.
MA: acquired index.html successfully!

MA: acquire(media, Clinton.gif, -1)
FA: ACK()
FA: sending Clinton.gif to home of Freddy for 50 units.
MA: acquired Clinton.gif successfully!

MA: acquire(media, cnn.html, -1)
FA: ACK()
FA: sending cnn.html to home of Freddy for 30 units.
MA: acquired cnn.html successfully!

MA: caught a ASK_FOR cmd

MA: suggest_server()
FA: suggest(localhost:10000)
MA: got suggest(localhost:10000)
MA: address_index=1, aBsize=3
MA: mig. to localhost:9000 after sugg

MA: caught a MIGRATION cmd
MA: caught a ADDRESS cmd
MA: about to move to localhost:10000
```

Figure 4.10 AEE trace showing a mobile agent in negotiation with a Facilitator

Chapter 5

The Agent Inception System

5.1 Introduction

The *Agent Inception System* (AIS) is a suite of software agents that run on a user's machine to simplify the programming, management, and manipulation of agents, their cargo, and their messages. It is also critical to the presentation of media and messages, which represent the final steps in the completion of a user-based task. Figure 5.1 illustrates the relationships between the AIS, the user, the network and other *Agent Execution Environments* (AEE) (see Chapter 4). Terms used frequently in this chapter are defined as follows:

Itinerary – An object that encapsulates the specific behavior of a mobile agent during its lifetime. An Itinerary is created or loaded by the end-user using the AIS interfaces. While an agent may be an instance of a specific class of problem-solving agents, without an Itinerary it has no task parameterization.

LiveAgent – An agent that contains a non-null Itinerary. LiveAgents are created, managed, and launched using the AIS. The following “formula” illustrates the relationship:

$$(\text{Agent Instance}) + (\text{non-null Itinerary}) = (\text{LiveAgent})$$

Voyager – A powerful platform from *ObjectSpace Inc.* for creating distributed Java applications and mobile agents. Object persistence API's and messenger and agent classes make this a suitable platform [VOY97].

Mobile Agent Platform – A networked environment consisting of *Agent Servers* to which mobile agents migrate, execute within, and acquire resources from, and end-user “home-bases” for agents called *Agent Inception Systems* (AIS) (the topic of this chapter).

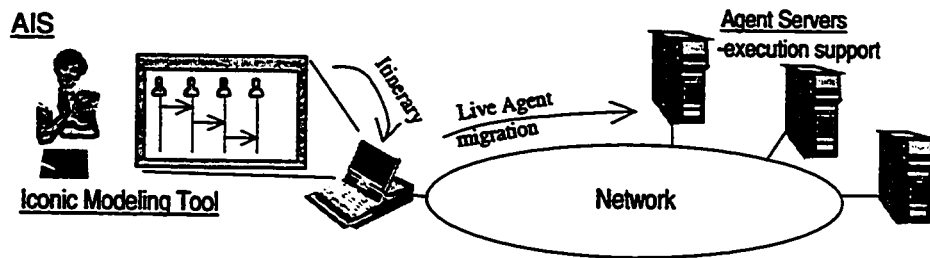


Figure 5.1 General architecture of the Mobile Agent Platform and position of the AIS

The AIS is a suite of Voyager agents that may communicate with other remote agents or even migrate should the need arise. The AIS and its components are based on the Java [SUN94] programming language. Components of the AIS can be considered agents in the sense that they:

- are autonomous and able to make their own decisions based on logic, inferred, or explicit information
- may delegate work to other agents
- co-operate to achieve the end-user's goal
- both serve *and* request data – e.g. they may be both clients and servers of data
- message in a peer-to-peer fashion
- may migrate to other nodes and preserve state at instance-variable granularity

The Voyager platform is the one that we have adopted for the development of the *Agent Inception System*. Its general programmatic features include the remote enabling of Java classes, easy object persistence, and object locating. Creating mobile agents is relatively easy and a *moveTo()* method makes agent migration very simple from a programmer's point of view. Other features that motivated our choice were: the availability of several messaging modes between agents, transparent agent messaging, the ability to remotely create agents, flexible agent life-span facilities, publish/subscribe options, and small system requirements.

The important Iconic Modeling Tool (IMT) component of the AIS is comparable to other such tools. [RON97] introduces a timing-like interaction diagram on which the messaging aspects of agents can be visualized. The graphical notation is based on six fragments that may be combined to create ones that are more complex. [TAK96] tackles the problem of visualizing workflow using an icon set - an Agent Prototype tool can be used to create agents that, for example, set up meetings between their human users. First an outline or state diagram is created, then a screen layout is generated, and finally actions are associated with each state in the state diagram to create a working agent. Anthropomorphic characters are used in the interface. The IMT is superior to both [RON97] and [TAK96] since it allows the visualization of agent mobility, cargo mobility, messaging, monitors, purchases, and other complex aspects of mobile agent behavior. AgentSheets [AGE97] is an end-user programming environment usable by novice users. It uses tactile programming where program components have both visual and interactive interfaces. AgentSheets can be used to create SimCity-like worlds, games, and cellular automata. Agents created using this tool may speak, play sounds, or open URL's, but are *not* mobile.

Overall, the AIS is a response to still-open questions regarding the creation and management of mobile agents and their itineraries, and fills an enormous need for a simple and intuitive end-user tool for itinerary modeling for the mobile agent paradigm.

5.1.1 Platform Independent Itinerary Format

This chapter proposes a platform-independent itinerary format. Such a format, consisting of parameter-value pairs (e.g. EMAIL:john@ucla.edu), is crucial to agent operation. Not only does the Itinerary describe the distributed Itinerary (e.g. task) to the agent, it does so in an agent-platform-independent fashion and so can be exploited by other agent platforms. All that is required is to create new Application-Program Interfaces (API's) to the format. In *AgentSys*, an itinerary-agent interface exists at the agent-level, implemented in Java. Supporting the process of Itinerary creation above the Itinerary level are the GUI's of the *Agent Inception System*. Below the Itinerary GUI's is the Itinerary script, followed by the Java agent class that interprets the script. Other agent construction kits could replace the AIS suite of GUI's so long as they also write the same Itinerary format. Similarly, other mobile agent platforms could exploit the power

of the Itinerary format by creating their own API's to this format (see figure 5.2). The Itinerary format and functionality is described elsewhere in this chapter.

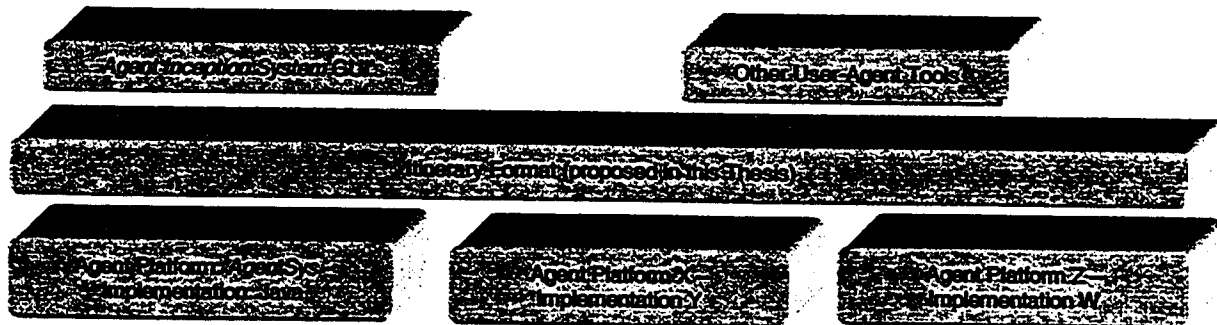


Figure 5.2. Platform independent itinerary format and interfacing. The AIS and Itinerary format combined with *AgentSys* protocols are a top-to-bottom agent platform implementation.

5.1.2 Dividing and Consolidating the Itinerary

The main purpose of the AIS GUI's is to enable the quick, simple creation of mobile agent itineraries. From the point of view of the user, once the itinerary is bound to an instance of the agent class, the task is in the agent's hands. At the AIS implementation level, however, the Itinerary is first scanned and divided into two distinct parts. Effectively, Web-based operations are separated from migratory Agent-based ones. In other words, operations that can be achieved statically from the AIS are separated from those requiring migration. Static operations are passed to the *Web Access Manager* while migratory ones are left alone – the mobile agent implementation and the Voyager platform manage these inherently. Results are consolidated and summarized dynamically using HTML and persisted or retrieved via the Media Manager. Figure 5.3 illustrates the process. The following sections illustrate the roles of each of the AIS managers.

5.2 Component Functionality & Description of AIS Managers

The AIS consists of several co-operating Voyager agents and is both the *home base* and *launching pad* for end-user mobile agents operating on a generic *Mobile Agent Platform*. The AIS is implemented on top of both Voyager2.0 and the SUN JDK1.1.5 in turn. The components

of the AIS are intended to be small, limited-purpose agents that, together, create an efficient and effective environment for the end user (with respect to mobile agents).

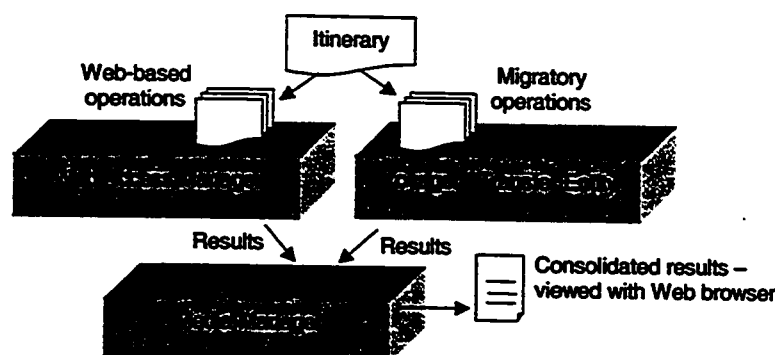


Figure 5.3 Dividing and consolidating the itinerary and results

Specification Collection Tool (SCT)

The SCT is both a form-based and visual toolkit for the collection of Itinerary data and for local agent management. The tool provides a suite of GUI's; agent behavior is specified using three classes of input tools: social, media, and physical GUI's. Once an Itinerary is created it can be "joined" with an Agent to create a LiveAgent or made persistent by the Itinerary Manager for later use. A LiveAgent may also be made persistent using the Agent Persistence Manager or may be injected onto the network by sending it through the Transfer Entity. Session options are persisted by making use of the User/Session Manager's facilities. The Media Assembler's services are utilized when users wish to assemble and view media returned by agents. Figures 5.4a and 5.4b illustrate the GUI representation of the SCT. The *AEE* and *Facilitator* pull-down menus enable the creation of Voyager servers and Facilitator agents (respectively).

Transfer Entity (TE)

The TE is a complex agent that handles the transfer and reception of mobile agents and data between nodes. Each node has a Network Daemon that listens for and reads data from the network - if the message is not well formed or corrupted, the TE sends the appropriate error code. The TE is essentially the port through which arriving messages or media reach the users Agent Inception System. A Transfer Entity instance implements the following important

methods: *dispatch()*, *retract()*, *fetch()*, *remoteMsg()*, *slave()*, and *query()*. The TE implements *AgentTransfer*, the transfer protocol, whose functionality and syntax are found in chapter 3 of this thesis.

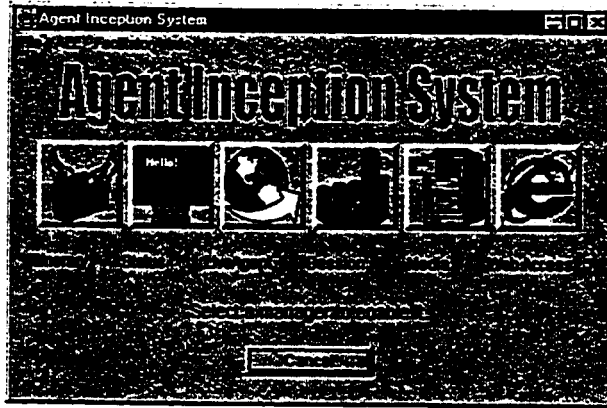


Figure 5.4a). The SCT interface.

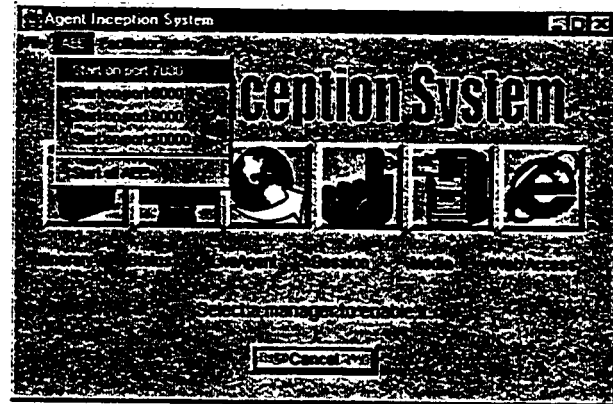


Figure 5.4b). Starting remote AEE's

Itinerary Manager (IM)

The IM manages all aspects of Itinerary administration. It receives raw Itinerary information and from the Specification Collection Tool and creates Itinerary objects. It also translates the visual representation into Itinerary data and interacts with the Itinerary Database to save or load itineraries at the request of the Specification Collection Tool, Agent Persistence Manager, or Media Assembler. The IM manages a number of GUI's for the benefit of the user and invokes those GUI's at the user's requests. The most important of these is the *Iconic Modeling Tool* that lets users literally "draw" distributed tasks (e.g. itineraries) upon a canvas. To support this operation the IM uses a number of supporting Java classes. *imageLabel* is a simple class that allows image components to be dragged and dropped onto other components. *Icon*, extending *imageLabel*, lets images be used as icons and implements mouse clicks upon them. *MyDrawingPanel* is a specialization of Java's *Panel* class that allows *Icon* instances to be added. *MyEntry* is popup menu used during the creation of itineraries. This popup contains both a textual entry fields as well as a pull-down menu of most-often-used choices (e.g. often-used Web page URL's such as *www.excite.com*). The important methods implemented by the IM are: *createItinerary()*, *saveItinerary()*, and *loadItineraryFromDb()*. The first method creates an Itinerary instance from the information specified textually and graphically by the user. The second method "serializes" an Itinerary into the platform-independent itinerary format and saves

it for later use. The third method creates a new Itinerary and fills it with the specifications from a previously saved Itinerary.

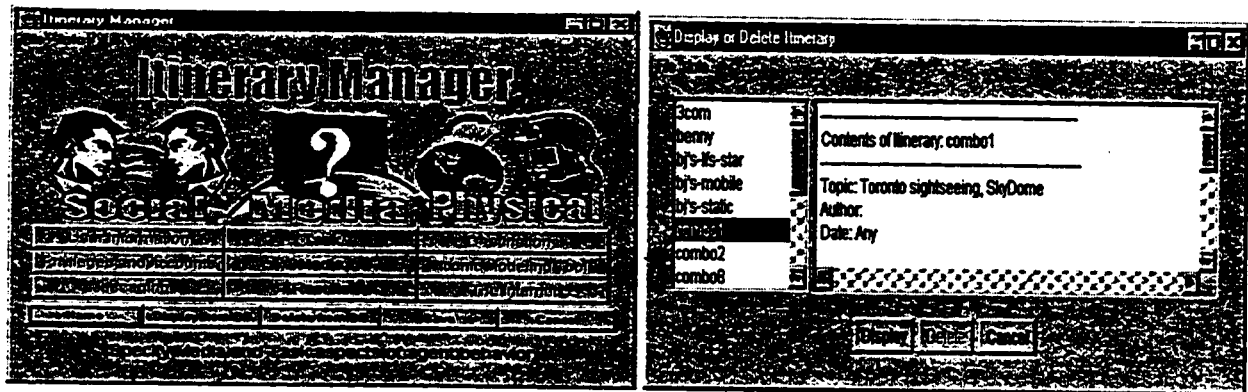


Figure 5.5 The Itinerary Manager (left) and the Itinerary display tool (right). The IM is the entry point for a number of behavior specification-GUI's, including the IMT (e.g. under "physical")

The *ModelingTool* class extends Java's *Frame* class. This class implements the Iconic Modeling Tool and all its functionality. It is instantiated with two parameters; *parent*, which points to the instance that invoked it, and *ItineraryName*, the name of the active itinerary. During construction of the instance, the visual components of the tool are created and data is initialized. During operation of the instance, the construction of the itinerary by the user proceeds in three stages: (1) drawing the "places" on the canvas, (2) adding information to the places, and (3) adding icons to the places. These stages are implemented as methods *stage1()*, *stage2()*, and *stage3()*. The *MyEntry* popup is used whenever the bubble button is selected by the user. Either place info is added or an icon parameter is specified (depending on the stage). Full pseudo-code for the Itinerary Manager classes can be found in Appendix A and the IMT is described in full detail later in this chapter.

The Media Persistence Manager (MPM)

The MPM manages all aspects of storing media on the local file-system in the form of files or as objects in a persistent Media Database. It is used primarily to deal with cargo (i.e. media objects) returned home after and during a successfully completed agent task. Cargo passes first through the Transfer Entity and then to the MPM. The MPM maintains an index of media and documents in the database, which are indexed by name for easy access by the user via a simple

list component. The MPM is used by the Web Access Manager (WAM) after the WAM retrieves media from remote Web servers – such incoming Web-based data is either:

- stored as is in the database, or
- indexed as a pointer (e.g. URL) and stored

The MPM implements the following self-explanatory methods to achieve these tasks: *storeMedia()*, *removeMedia()*, and *loadMedia()*. The interface to stored media is the Media Manager GUI, illustrated in figure 5.6, that offers basic file features (e.g. delete, view, etc.).

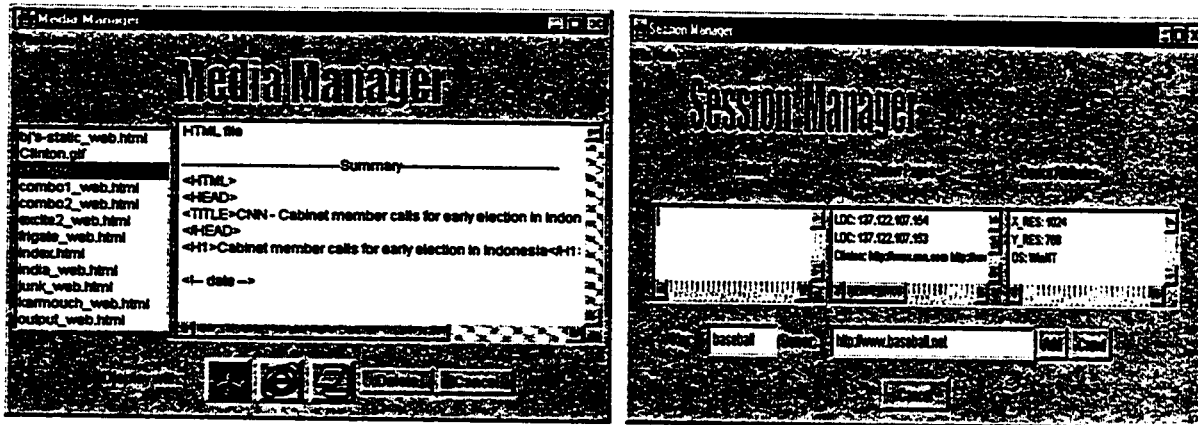


Figure 5.6 The Media Manager for manipulating and viewing agent-acquired media (left) and the Session Manager to track user preferences and device information (right).

Agent Persistence Manager (APM) – Launching Live Agents

The APM creates, saves, and loads two types of agents; standard agents are those with no associated Itinerary and “live” agents are those that contain non-null Itinerary objects. LiveAgent and Agent Databases are used for persisting these agents. The APM instantiates agent objects during the creation phase and initiates them with user parameters. It is used by the Specification Collection Tool to save or load agents at the users request, and uses the services of the Itinerary Manager to acquire itineraries from the Itinerary Database. The APM’s methods include: *createAgent()*, *saveAgentToDb()*, *loadAgentFromDb()*, *createLiveAgent()*, *saveLiveAgentToDb()*, and *loadLiveAgentFromDb()*. These method names are self-explanatory. They generally take an agent object and save it, or retrieve a pointer to a previously saved agent object. The APM is manifested via the Live Agent Manager GUI (figure 5.7). This GUI uses an intuitive GUI that enables users to associate the agent class, Itinerary name, and agent name together to create a Live Agent.

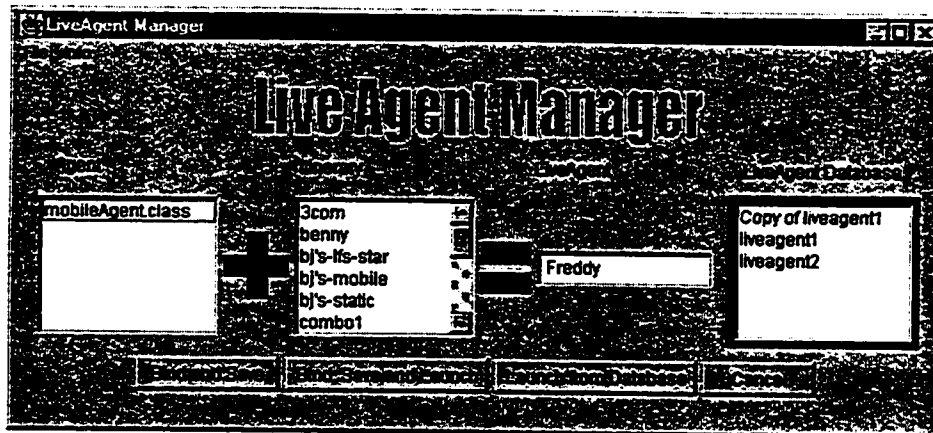


Figure 5.7. The Live Agent Manager exposes the basic equation $\text{agent} + \text{itinerary} = \text{liveAgent}$ to the end user via an intuitive interface.

User/Session Manager (USM)

The USM maintains a History Database of tasks and agent usage. It updates a Yellow Pages Database that associates common tasks to both agent and Web servers. The database consists of keyword and server pairs. Periodically coordinates with the Web Access Manager to add to the database using simple Web-crawling techniques on the Web. The USM maintains a Device Database to keep track of device capabilities. YellowPages Database services are used by the Web Access Manager while task histories and server lists are used by the Specification Collection Tool. Device and History data are stored in the User Database. USM methods include *showTerminalAttributes()* and *showDeviceData()* which are used to display the current device and terminal settings. The USM is illustrated in figure 5.6 (right)

Web Access Manager (WAM)

The WAM provides HTTP access to Web servers from the local host when it is called for by the agent Itinerary. When specific Web servers are identified as possible document resources, the server is indexed in the Yellow Pages Database. When no specific server is identified, the WAM uses the yellow pages followed by Web search tools such as catalogs and search engines (e.g. YahooTM and ExciteTM) to find one. The Yellow Pages Database maintains Vectors of keywords indexed to Agent and Web servers. The WAM is periodically used by the User/Session Manager to update the yellow pages. When Web documents or media are retrieved by the WAM they are

sent to the Media Persistence Manager who stores them in the Media Database. Figure 5.8 illustrates the WAM GUI.

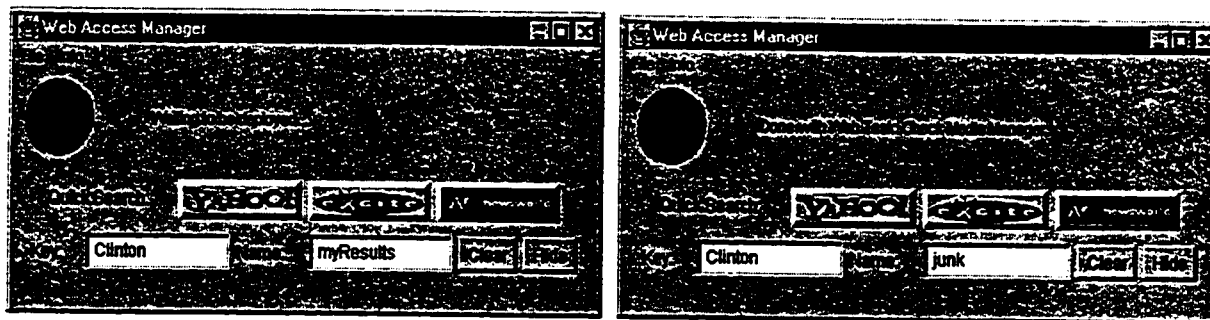


Figure 5.8 The WAM interface (left and right). The globe flashes as the search proceeds. Quick searches can be done using the Excite™ or Yahoo™ wrappers. Results are passed to the MPM.

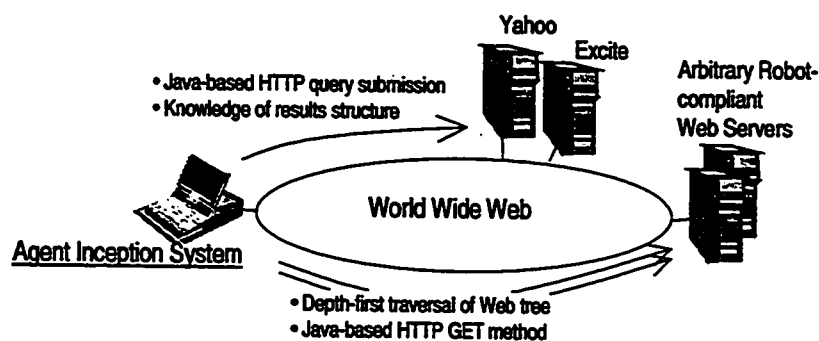


Figure 5.9 The Web is accessible and integrated into the AIS via the WAM. Yahoo™ and Excite™ Web sites are “wrapped” with Java methods while arbitrary Web sites are “spidered”.

The WAM class extends Java’s *Frame* class and its constructor builds the GUI. No parameters need be passed into the class upon construction. The WAM’s important methods include *updateYp()*, *sendTask()*, *setStatus()*, *setIcon()*, *startFlashingIcon()*, *stopFlashingIcon()*, *killWebGrep()*, and *cleanUp()*. *UpdateYp()* is used when either the user requests the YP database be updated or when the WAM determines it needs updating. *SendTask()* is the method visible to other classes. When object A wishes to perform a web-based task, it invokes this method. *SendTask()* takes two parameters; the itinerary name and the agent’s name. Upon being invoked, the *sendTask()* method opens the named itinerary from the database, and parses it for the following information; (i) the Web-task keywords and (ii) the Web URL’s to visit. With this information, it enters the following loop:

```
For(each Web site to visit){  
    For(each keyword in the task description){  
        Instantiate a WebGrep object for this sub-task;  
    }//for  
}//for
```

The WebGrep object implements the Web-spider paradigm – each such instance is started in a separate thread of control to maximize concurrency and reliability. The remaining WAM methods are used only by the WAM itself for programmatic and administration purposes. Associated strongly with the WAM are the classes *WebGrep*, *WebGrepThread*, *Spider*, and *TimeKiller*. These classes are used almost exclusively by the WAM (e.g. not by other classes) and are explained in the following paragraphs. Figure 5.9 illustrates the Web integration via the WAM and AIS.

WebGrep and associated Classes

The WebGrep class implements a spider-like search of a tree of Web-accessible files. It is based on the algorithms implemented in software found at <http://www.acme.com/java>. The tree of reachable files beneath the given root is searched for the occurrence of particular keywords. To achieve this functionality flexibly, WebGrep uses a *to-do-queue* and a *done-queue* to keep track of pages to visit and those to be visited (resp.). As pages are searched, they are added to the *done-queue*. If a hyperlink is followed to a page that is not on the *done-queue* it is added to the *to-do-queue*. The tree is searched in a depth-first fashion. A WebGrep instance is constructed with the following parameters: *pattern*, *location*, *name*, *MAXHITS*, *privateNumber*, *agentName*, and *parent*. *Pattern* is the string to find within the given page Web page. *Location* is the URL of the root of the tree to search. *Name* is the name of the itinerary. *MAXHITS* is the maximum number of matching pages desired for this pattern. *PrivateNumber* is used internally. *AgentName* is the name of the calling agent and *parent* is a handler to the owner of this WebGrep object. The main loop of this class first checks the location of the search and reacts accordingly. If *location* indicates that either the Yahoo™ (www.yahoo.com) or Excite™ (www.Excite.com) Web sites is to be explored then the *searchYahoo()* or *searchExcite()* method is invoked (resp.) These two methods implement HTML-tag-based wrappers to the Excite™ and Yahoo™ search engines. Such wrappers require knowledge of:

1. the location (e.g URL) and format (e.g parameters) of the HTML query input

2. tag-level details regarding the presentation of the results of such a query

The WAM tracks and stores such knowledge. The *searchYahoo()* method uses the following basic algorithm to extract query results directly from the dynamically generated page:

```
URLstring="http://av.Yahoo.com/bin/query?p="+pattern+"&hc=0&hs=0";
OpenConnection to URLstring;
While(true){
    Readline of webpage;
    If line is null break;
    If line starts with "<li><a href>" and we have MAXHITS yet
        Add this line to the YAHOO_OUTPUT;
        Increment hit count; if past MAXHITS break;
} //while
writeHTMLfile();
```

The *searchExcite()* method uses a similar technique to extract query results from the Excite-generated page:

```
replace spaces in pattern with "+" characters;
OpenConnection to URLstring;
//read past junk at top of output
Read lines until line contains the string "% </SMALL>";
While(true){
    Readline of webpage;
    If line contains "<A HREF=" and line doesn't have "<br><a HREF="
        Add this line to EXCITE_OUTPUT;
        Increment hit count; If past MAXHITS then break;
    Else if line contains "<form" break;
} //while
writeHTMLfile();
```

In both of these cases the method *writeHTMLfile()* is used to create or update the HTML file that will later be accessed via the *Media Manager* and browsed with a Web browser.

If the search location is neither Yahoo™ nor Excite™ then first a new *Spider* instance is created. A *Spider* is an instance of a class that generates the index of links on a single HTML page. The reference to the *Spider* is then passed to the WebGrep's *findIt()* method along with the *pattern*, and other boolean options. *FindIt()* creates an array of threads that will implement the depth-first search of the Web tree. While threads remain free and there is more information to *Spider* a new *Spider* is created.

```
create and nullify new thread array threads[nThreads];
for(;;){
    nAlive=0; freeSlot=-1;
    for(j=1 to nThreads){
```

```
        if thread[j]==null freeslot=j;
        else if thread[j] notalive then freeslot=j; thread[j]=null;
        else ++nAlive;
    }//for
    if(no threads still alive and no more elements to spider) break;
    if(free slot avail. And more to spider to){
        threads[freeslot] = new WebGrepThread(pattern,spider,this);
    }//for
```

Finally, two important classes are used to manage the operation of the search procedure. *WebGrepThread* is used to create a separate thread of control for the Spider. *TimeKiller* is a class that watches any thread and kills it after a particular time-out period.

It should be noted that the Spider implemented in *AgentSys* as part of the Agent Inception System is a well-behaved one. That is, it obeys the *Robot Exclusion Protocol* (REP) [ROB96]. REP allows Web site managers to allow or disallow various types of Web-based spiders and robots. If the REP is enforced at a particular Web site, it is found in a file called *ROBOTS.txt* directly under the document root. This file spells out which agents or class of agents should or should not use the Web server. If no classes of agents are desired, the *AgentSys* spider obeys.

Media Assembler (MA)

The MA assembles and (if necessary) automatically edits HTML files to create a “presentation” for the user based on the specifications (e.g. may create links, etc.). It also creates other types of presentations from raw media, depending on task types, including multimedia documents and tele-learning courseware (multimedia documents and digital courseware are other components of a larger project). The MA uses the Media Persistence Manager to save assembled documents, and spawns document viewers based on the capabilities of the current terminal – for example, PDA’s cannot use viewers like *Navigator*. It communicates with the User/Session Manager to acquire terminal characteristics and occasionally the Itinerary Manager to consider Itinerary information. The Specification Collection Tool uses the MA to create and view documents on demand.

Status Monitor (SM)

The SM creates and updates a “live” monitor that tracks agent behavior and displays it visually for the user. It accepts and displays status messages from remote user agents intended for their

users, and allows users to send messages to remote agents or to facilitators perform various operations on agents. The SM uses the Transfer Entity to send messages to remote LiveAgents, and is used by the Transfer Entity to receive messages. The SM uses a simple but effective visualization play for illustrating the current agent's mode. At the bottom of the SM GUI is a small horizontal area in which an iconic representation of the mobile agent "moves". Icon movements correspond to the physical migration of the agent during its task. When the agent is "at home" it appears at the far left. When the agent is "migrating" it appears on top of the network cloud. When the agent is at a remote AEE it appears beside the AEE icon. Furthermore, the user has a pull-down menu of queries that he may send to an executing agent (figure 5.10). These include:

- What is the last media you acquired?
- What is the last conversation that you engaged in?

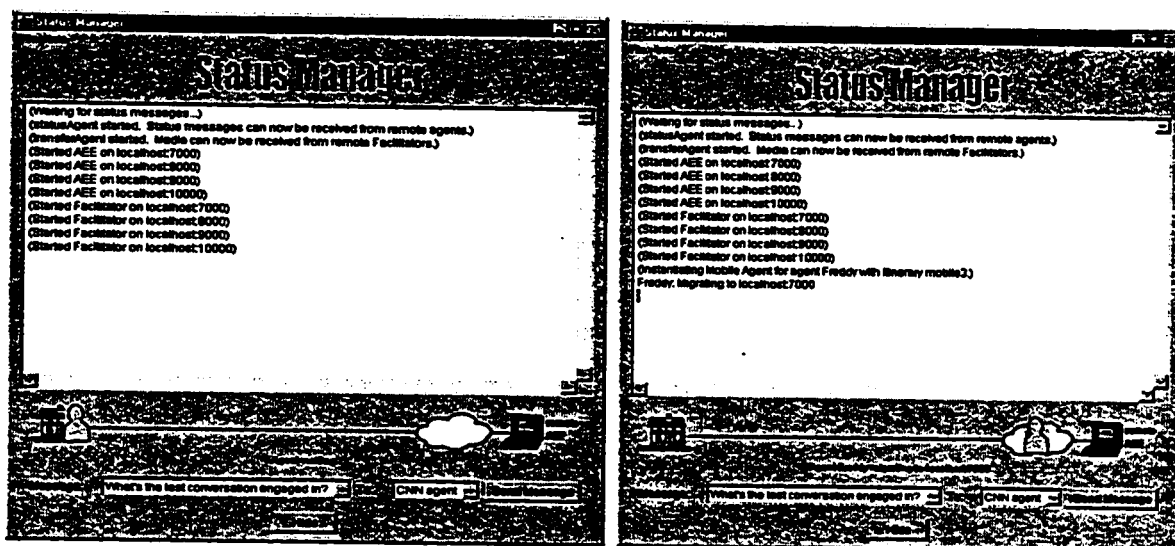


Figure 5.10 The SM GUI (left and right) is used to receive messages from remote agents (e.g. the blue text). It can also send messages to remote agents (e.g. via "send message"). The positioning of the agent icon represents the current state of the currently active mobile agent.

5.2.1 Class Hierarchy of AIS Components

Components of the AIS fall into three general categories: agents, databases, and GUI's. Manager agents inherit from Voyager's Agent class and in turn, each of the AIS managers inherits from Manager. Agents and LiveAgents also inherit from this class – LiveAgents "contain" Itinerary objects that inherit from Java's Vector class. Figure 5.11a illustrates the AIS

managers and the class hierarchy to which they belong. Figure 5.11b illustrates the usage pattern amongst AIS managers – usage is denoted by a connecting line with the word “uses” closest to the class that uses the services of the other. Figure 5.11c illustrates the relationship between managers, agents, and ObjectSpace’s Agent class. It also spells out all of the important methods and instance variables contained in the classes. Database objects implement Voyager’s VoyagerDb interface which defines Get and Put methods to move objects easily to and from a persistent store. Databases are managed by Manager objects. The GUI objects implement the human-agent interfaces.

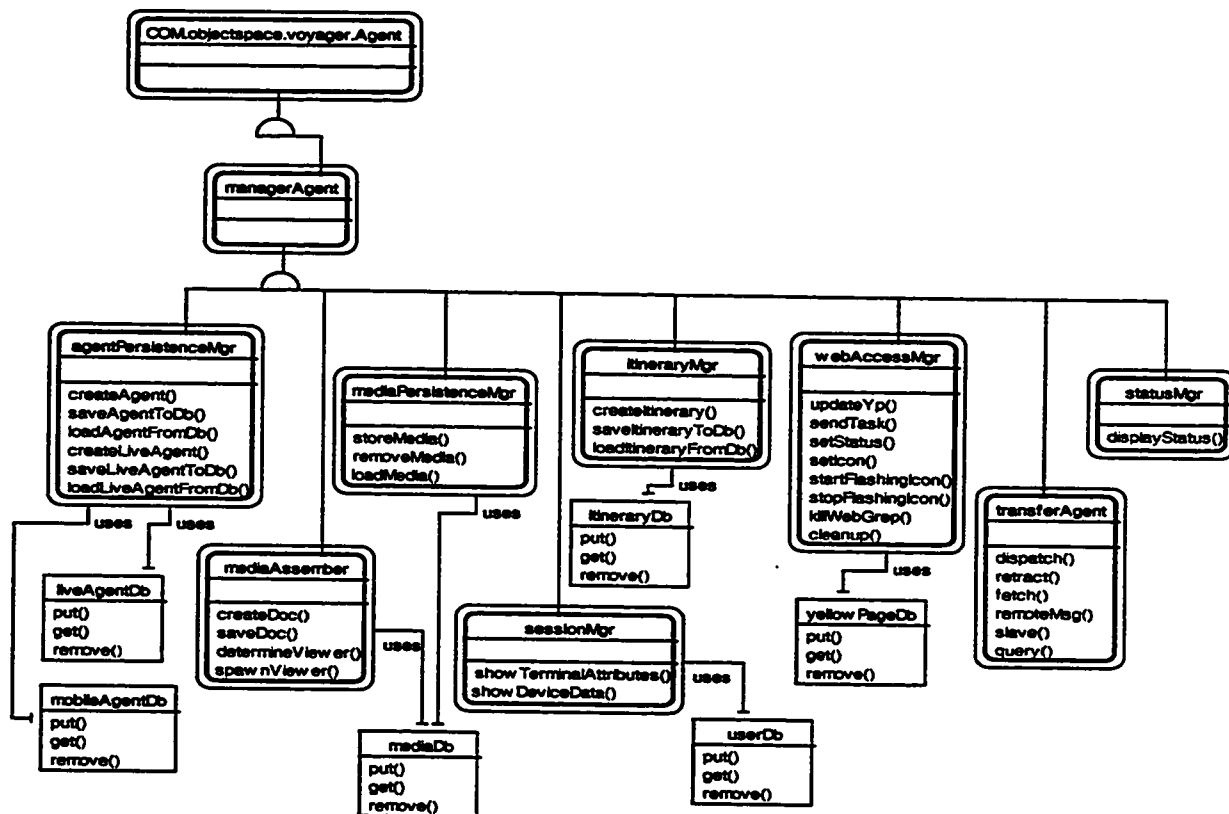


Figure 5.11a). Class hierarchy and important methods of the AIS managers.

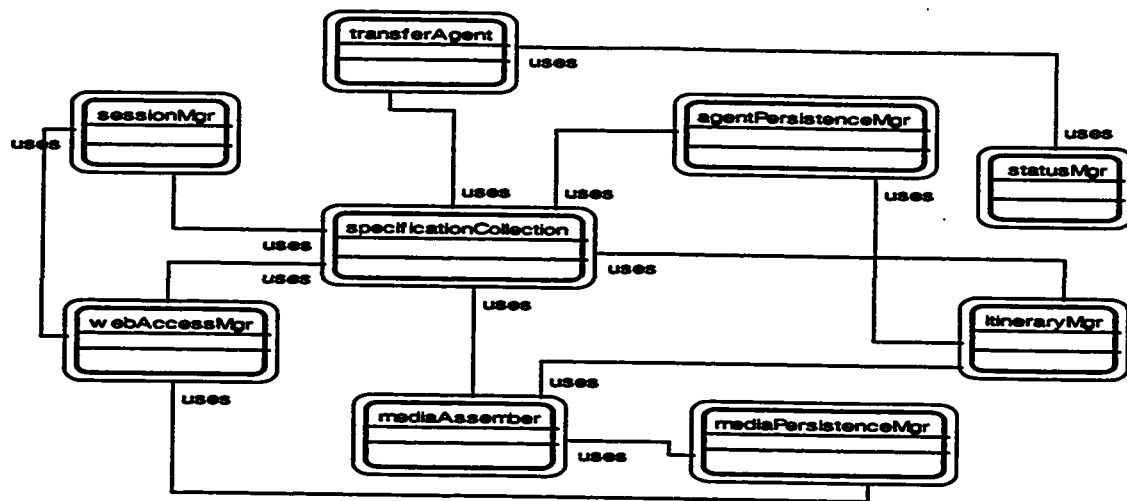


Figure 5.11b) Usage pattern amongst the AIS managers.

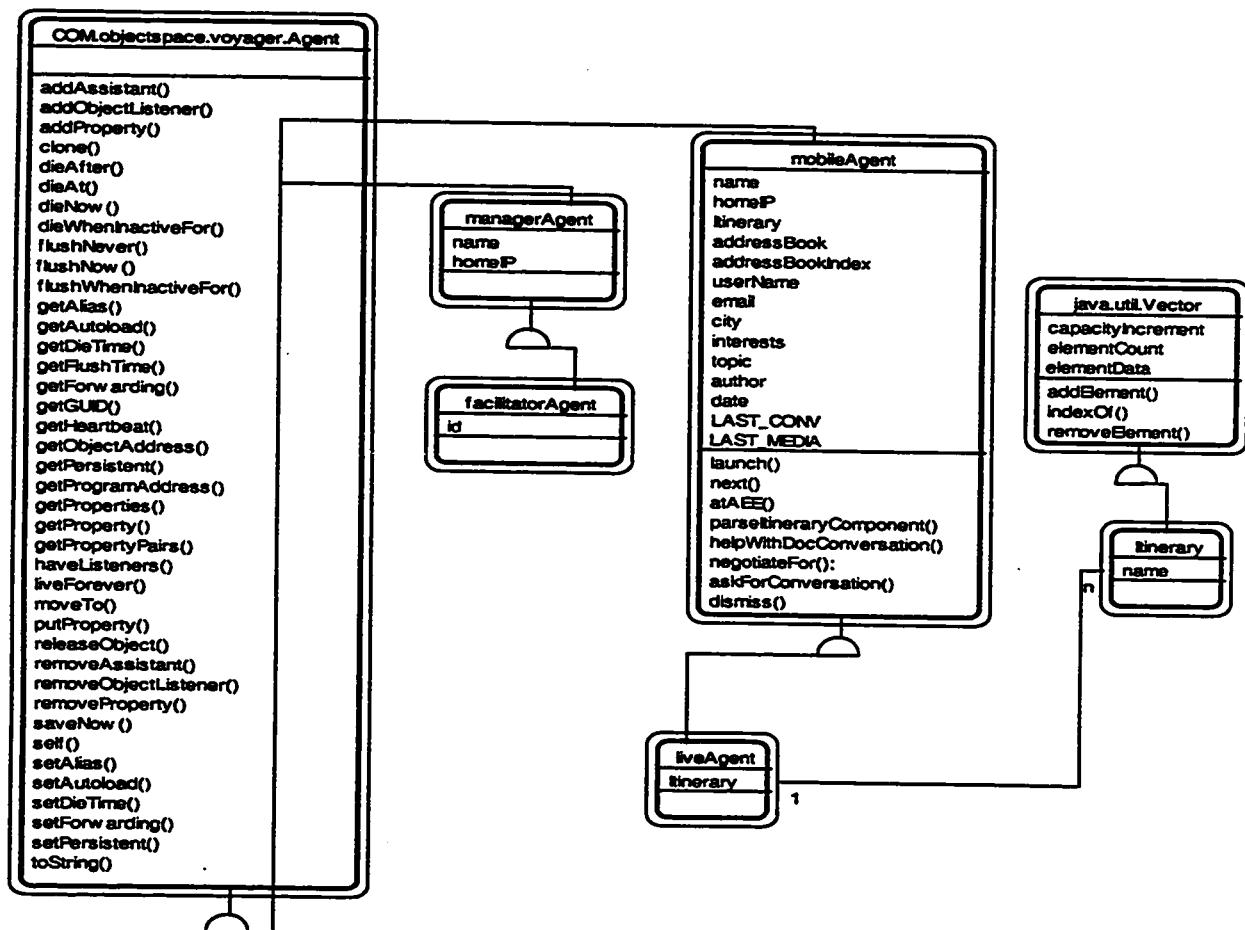


Figure 5.11c). The AIS Managers all inherit from Voyager's Agent class – other types of agents are LiveAgents and generic Agents. The LiveAgent “contains” an instance of an Itinerary object that in turn inherits from Java's Vector class.

5.2.2 Example Scenarios

Figure 5.12 illustrates the important steps involved in creating a persistent agent and launching it upon the network. Starting with the initialization of the specification collection tool which supplies the user options and GUT's, the steps are as follows:

1. The graphical end-user Specification Collection Tool (SCT) is initialized and started with the *Init* method. Through the facilities of the SCT, an Itinerary is created in human-readable form. The Itinerary Manager (IM) is called to instantiate an Itinerary object from the supplied data using the *CreateItinerary* method. Session options and Itinerary creation histories are persisted using the User/Session Manager's (USM) *SaveSessionData*. The USM adds elements to the User Database using *Put*.
2. The SCT calls upon the IM to persist the Itinerary object using the *SaveItineraryToDb* method.
3. IM interacts directly with the Itinerary Database and uses the *Put* method to add the Itinerary.
4. Through the SCT the user selects the "type" of agent that is desired; the Agent Persistence Manager (APM) instantiates the appropriate agent object.
5. The user requests the agent to be named and persisted for later use – the SCT uses the APM's *SaveAgentToDb* method to achieve this.
6. The SCT request to save the named agent goes through the APM which uses the Agent Database's *Put* method.
7. A LiveAgent object is requested by the SCT and is created by the APM's *CreateLiveAgent* method using the current Agent and Itinerary objects.
8. The LiveAgent is to be persisted - the SCT uses the APM's *SaveAgentToDb* method, passing the named LiveAgent as data.
9. The APM saves the LiveAgent in the LiveAgent Database using the *Put* method.
10. The SCT launches the current LiveAgent onto the Network by using the Transfer Entity's *Dispatch* method.
11. If the LiveAgent's task calls for Web page acquisition the Web Access Manager (WAM), which implements the HTTP protocol, acquires them. If the WAM finds documents, it uses the Media Persistence Manager's (MPM) *StoreMedia* method to store them in the Media Database, which in turn uses the *Put* method native to all databases. The WAM may also update the Yellow Pages Database with a new entry describing the Web Server from which it acquired relevant media.

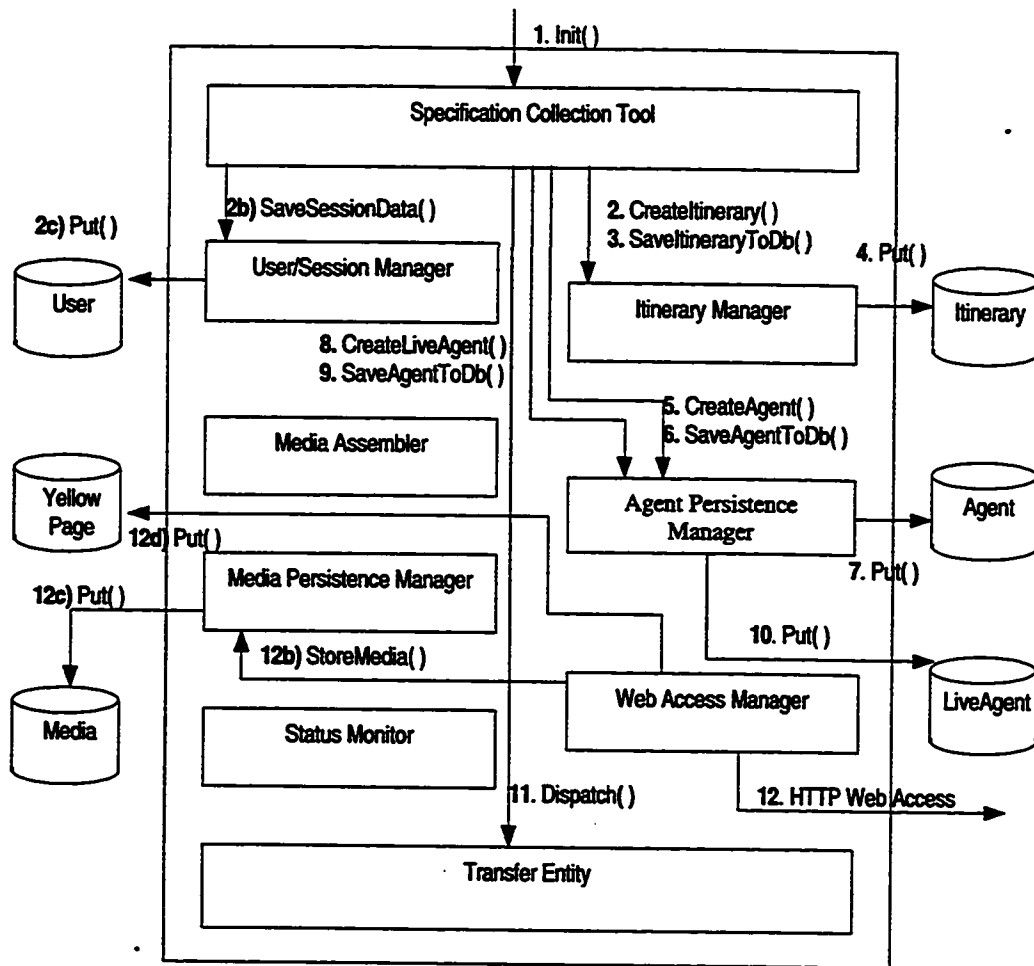


Figure 5.12 The process of agent creation, persistence, launching, and execution

In figure 5.13 the main steps involved in receiving media and status messages, assembling and viewing documents, and refining the task are shown. Starting with the reception of a dispatched status message from a remote agent, they are as follows:

1. The Transfer Entity (TE) receives a dispatched media (e.g. acquired cargo such as HTML files) from a remote agent (from a local point of view, the TE uses its *Listen* method to receive network messages – the remote agent's media, however, are sent using the TE's *Dispatch* method).
2. The Media Persistence Manager (MPM) is sent the *StoreMedia* message along with the media.
3. The MPM communicates with the Media Database using the *Put* method to persist the media.
4. When remote status messages arrive from remote user-agents, they are sent using the *Remote_msg* method (the local TE uses its *Listen* method to receive them).

5. The remote message body is passed to the Status Monitor using the *DisplayStatus* method. The Status Monitor updates its state and prints the message to its GUI – in this way the human user is immediately informed of the incoming message and its content.
6. Using the SCT, the user may wish to assemble all retrieved media from a particular task into a document. The SCT invokes the Media Assembler's (MA) *CreateDoc* method to instantiate such a document. The MA, in turn, uses the Media Persistence Manager's *LoadMedia* method to acquire media from the Media Database.
7. The appropriate viewer is determined when the SCT sends the MA the *DetermineViewer* method. A Web browser is typically the viewer but in some cases (e.g. a mobile user) device data, terminal attributes, and media attributes may need to be examined first. To do this the MA sends the *DeviceData* and *TerminalAttributes* methods to the User/Session Manager which communicates with the User Database.

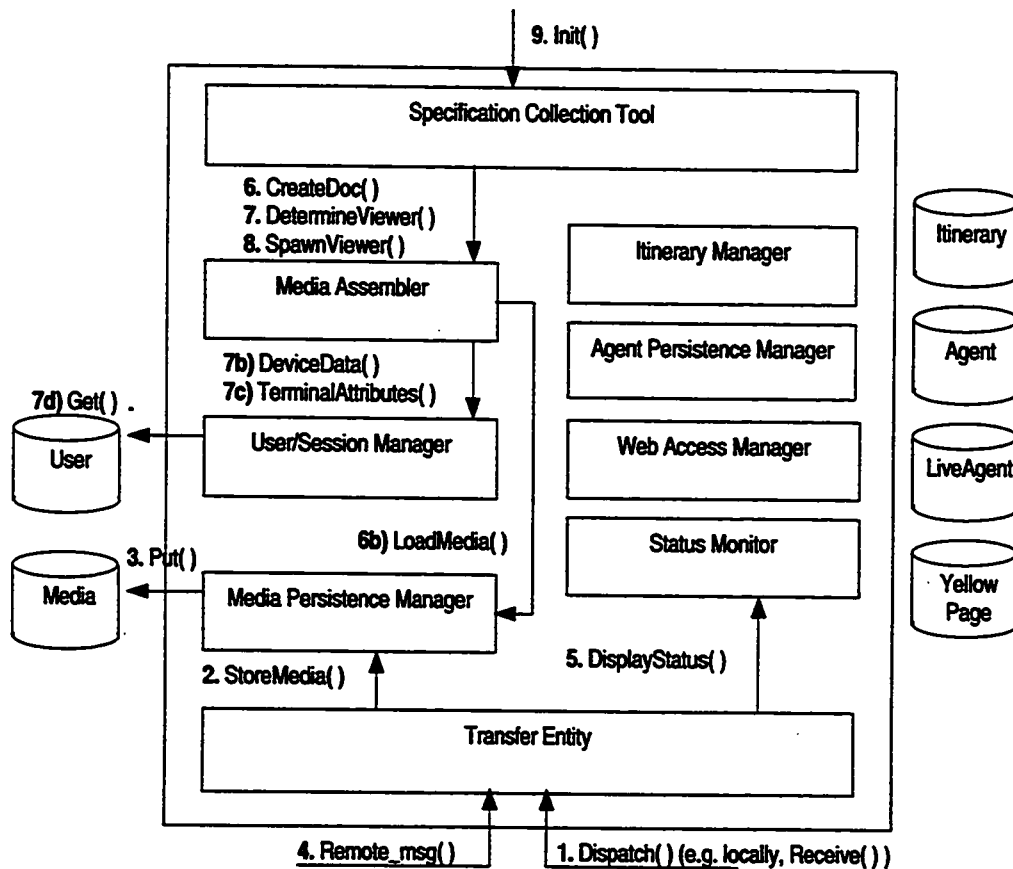


Figure 5.13 Receiving media and status, assembling and viewing documents, and refining the task

5.3 Creating LiveAgents and Itineraries Visually

Visual languages for program control have existed for quite some time and remain an active research area. The Iconic Modeling Tool (IMT) component of the AIS differs in that it directly addresses the challenges of task specification and control in the mobile agent domain. The IMT is the result of several design goals:

- Use a simple, easy to understand and *small* set of icons
- Allow drag-and-drop manipulations on the GUI when possible and form-based GUI's when necessary
- Address primarily the difficult issues of physical agent behavior (e.g. cargo, messaging, etc.) in an information-retrieval context. Information-retrieval, in some sense, is the driving scenario behind much of this work
- Based on the classical (e.g. easily-understood – even for novices) timing diagrams used to describe network communications and protocols
- Create an agent-platform independent Itinerary structure, exploitable by other communities

5.3.1 Minimal Specification Requirements for Mobile Agents

We describe agent behavior in three dimensions: social, physical, and task. This section describes the requirements of each of the dimensions of agent behavior specification. While the mobile agent moves around a network collecting and providing information to other agents, it must do so according to the desires and guidelines of its human user creator. Furthermore, it should adhere to a **socially** acceptable fashion of interaction with other agents (figure 5.14). For example, when a mobile agent negotiates with a facilitator for the price of a data element, it should make continually *increasing* offers for the data (whereas the service agent will make *decreasing* offers). Fortunately, the nature of the required information is such that it lends itself to a form-based representation. User information and sharing describes what information represents the user on the network and what can be shared with other agents. Negotiation rules describe what type of offers to make for particular media types of different sizes. If any account numbers, ticket numbers, etc. are required at certain sites, these must also be provided. Interaction information describes what type of other agents may be helped or asked for help. Messaging information defines messages that are intended for other specific users or agents, and describes status information and human-intervention rules.

The physical behavior (figure 5.16) of mobile agents is the most interesting and most difficult for a user to pre-specify. Furthermore, there is a tradeoff between pre-specification of physical agent behavior and agent-adaptivity since clearly an increase in one decreases the other. The mobility of agents on networks creates a number of issues that must be resolved. The cargo issue is critically important. It is assumed that the mobile agent will acquire media to be returned to the user. When the mobile agent “carries” these media to intermediate nodes it is called cargo. However, there are several other options: The user may opt to have cargo sent home immediately after it is acquired, or leave it at the node where it was acquired and where it may be fetched at a later time. The physical location of name-servers, resources, other users, etc. is also important and must be defined. Networking information describes the type of messaging to use, off-limit sub-networks, when to use open channels, QoS, etc. Cloning information describes how many clones (i.e. child agents) to spawn and how to divide the work. Since this is a rather abstract and complicated scenario the default (and most often used) action is to repeat the previous sub-itinerary with cloned children. Other specifications include protocol naming and advertising options.

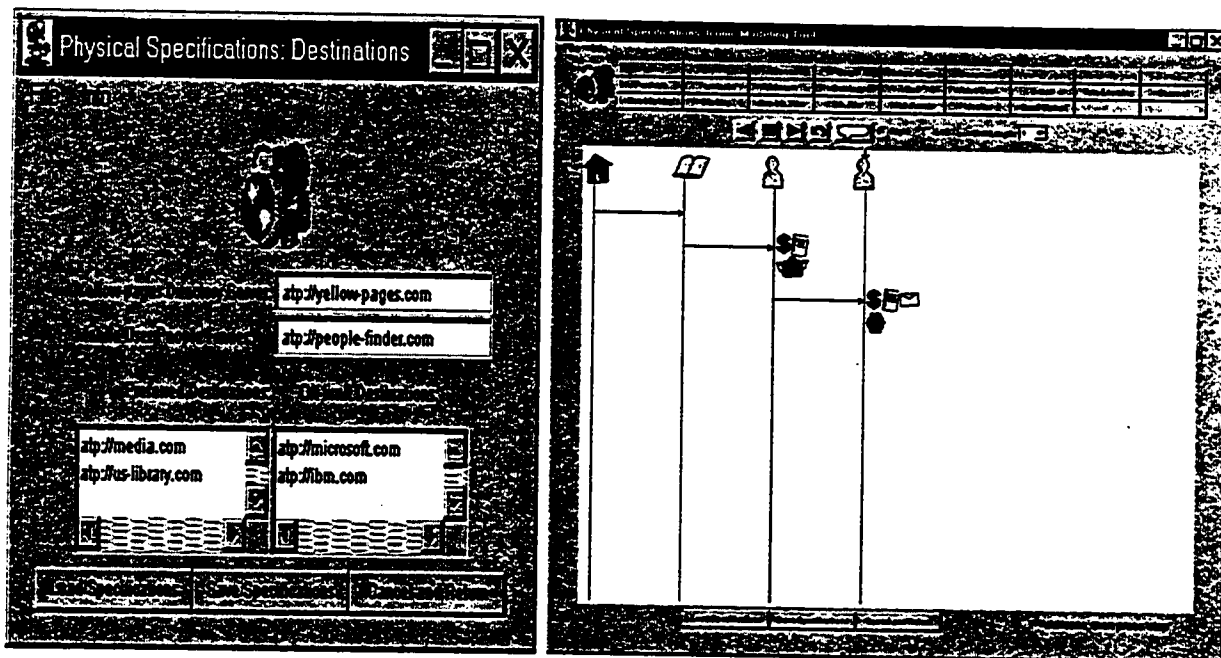


Figure 5.16 Interface for human entry of physical agent characteristics (left) and the Iconic Modeling Tool (right - see section 5.4) for itinerary creation.

5.3.2 The Icon Set

The IMT icon set is used to create, edit, and save mobile agent behavior. By associating icons spatially and semantically, and providing textual information when required, the IMT can be used to model useful mobile agents. The meaning of icons is context and situational dependent. Compared to learning and using an agent scripting language such as TCL or Java, the IMT is sufficiently easy to use and has a quick learning curve. The user must still learn how to associate icons together to form meaning (e.g. the dollar sign next to a document or media imply the acquisition or purchase of that document class), but after only a few trials it becomes natural. Mobile agent behavior is established through the form-based GUT's and the visual drag-and-drop-style IMT. Graphically, the IMT is based on timing diagrams that are commonly used to describe network communications and protocols. In this paradigm, network entities such as the user's machine, other agent execution environments, and web servers are abstracted to vertical lines with appropriate descriptive icons at the tops of the lines. Time flows downward along the y-axis. A description of the icon set and its functionality follows (see figure 5.17).

The *Agent* icon represents facilitator (e.g. a service or suite of services) and typically has an attached DNS or IP address. The *Unknown-Agent* icon is used to represent mobile agents and services that are not explicitly known at modeling time. The *Home* icon denotes the computer to which results should return, and helps the user visualize the system with respect to his/her own computer. The *Clone* icon is to indicate that an agent should send a child agent to a remote node instead of itself. The *Store-Cargo* and *Return-Cargo* icons are used to indicate that collected media should either remain at the machine where it was acquired (to be sent home at a later time), or return immediately and directly to the home computer (respectively). The default action encapsulates cargo with the agent; thus the cargo migrates from node to node *along with* the agent. Options such as these distinguish the IMT from other such tools. The *Yellow-Pages* icon is used to represent a type of directory services that might exist on the network – the IP of this service may not need to be explicitly specified. The *Acquire* icon is used to indicate that negotiation for media should occur at a particular resource – the *Document* icon is typically used adjacent to the *Acquire* icon. The *Terminate* icon indicates that the mobile agent should be terminated and not returned home. This can also be used in conjunction with clones (e.g. child agents). The *World-Wide-Web* icon represents a Web server and is used whenever the agent is to

acquire media from remote Web servers using HTTP. The *E-mail*, *Initiate-Video-Conference*, and *Initiate-Tele-learning* icons indicate how the results should be returned to the user – video conferencing and tele-learning are two other related agent-based projects in our lab.



Figure 5.17 The icon set of the Iconic Modeling Tool.

Some of the IMT icons represent network and communications operations. The *Open-Channel* icon is used to create an open channel between agent and server for communication. An example of this is the creation of a virtual circuit. Both agents and remote messages may be sent in blocking or non-blocking fashion simply by choosing the appropriate icon type. HTTP messages may be sent directly to Web servers using the *Web-Access* icon. The *Send-Status-Info* is used to insist that the mobile agent sends status information. For example, the user may want such information after each media is acquired.

5.3.3 Technique for Icon Combinations and Itinerary Generation

The basic steps to create an agent itinerary using the IMT are:

1. add “places” to the workspace
2. add information (e.g. IP addresses) to the places if necessary
3. add migration specification, messaging, and cloning operations to the workspace
4. add additional icons to complete the specification.

At the implementation level, the IMT uses grouping to infer meaning from icons dropped onto the IMT canvas. By enforcing a modestly strict “style” to visual Itineraries the IMT is easily able to associate by proximity, for example, the dollar sign and document icons from figure 5.10 (right). Then by using a look-up table the IMT adds the appropriate information to the Itinerary object. Dynamic aspects of Agent behavior, such as branching or decision-making, are hard to program visually without complicating the overall visual representation beyond the point of what a novice user can grasp. To get around this we implement the more complex (but general) decision-making logic in the Agent classes and leave only specific parameterization (e.g. application-oriented data) to the IMT and Itinerary class. More complex visual behavior specification such as branching is possible future work. Three aspects are important in itinerary generation: (1) icon combinations, (2) textual parameter entry, and (3) itinerary objects.

Icon Combinations

The following icon combinations are valid and meaningful. Acquisitions can be implied by combining the Acquire icon and one of: Media, Ontology, Authentication, or Protocol/Data. In all cases, the intent is to acquire an entity as follows (all correspond to the *Acquire* protocol):

 Acquire	 Document	Acquire a media or document. If the document is explicitly named then that one is acquired. Otherwise, media are collected using the parameters of the task specification.
	 Ontology	Acquire an ontology or an ontological term. The name of the ontology and/or term must be specified.
	 Authentication	Acquire some sort of authentication or privilege. The authentication must be named.
	 Protocol/Data	Acquire a protocol. The protocol must be named.

Table 5.1 Icon combinations for acquisitions.

Exchanges of information are an important part of the *AgentSys* scenario. To enable a useful and social system the IMT must support the exchange of documents authentication and protocols as follows:

 Exchange	 Document	Exchange a media or document. If the document is explicitly named then that one is exchanged. The exchange process is defined in the <i>AgentTransact</i> protocol. The initiating agent offers a media and the recipient then chooses whether or not to participate. Corresponds to the <i>Exchange_docs</i> protocol.
	 Authentication	Exchange some sort of authentication or privilege. The authentication must be named. This corresponds to the <i>Ask_for_auth</i> protocol.
	 Protocol/Data	Exchange a protocol. Corresponds to the <i>Ask_for_protocol</i> conversation.

Table 5.2 Icon combinations for exchanges.

Agents also ask facilitators and other agents for services related to media acquisition. These services are defined in the *AgentTransact* section of this thesis. The IMT offers the following combinations that correspond to asking for various types of services:

 Ask for	 Document	Asks for help with the current document. The facilitator responds (if possible) with media that may fit into the current document parameterization. Corresponds to <i>Help_with_doc</i> .
	 User	Ask for user information. This combination is used to query another agent's user information. Corresponds to <i>Query_user_info</i> .
	 Yellow-pages	Ask for index of media. Returns a list of media by name.
	 Agent	Ask for local agent. When an agent desires communication with another similar agent, this combination is used to gain a "handle" onto such an agent (if possible). Corresponds to <i>Provide_local_agent</i> .
	 Server	Ask for server. When an agent wishes to migrate and the next migration node is not specified, it will ask the local server to suggest (if possible) another server. Corresponds to <i>Provide_server</i> .

Table 5.3 Icon combinations for queries.

Textual Parameters

The following textual keywords are offered to the end user through the IMT. These keywords are used to add semantics to agent behavior that would otherwise be difficult to specify visually. The entry point for these keywords is always the "bubble" icons on the IMT tool panel and the user does not have to remember these words – instead they are selected from a pull-down of terms. The parameters available to the user at any time are as follows:

Parameter	Description	Example
MESSAGE	When an agent is to deliver a message to another agent or user, that message can be entered by the user programmer upon inception.	MESSAGE="Hello, Tim! Cancel our Wednesday meeting, please."
MAX_SERVERS	After a visit to a Yellow Pages server the agent may have been provided with many suggested servers. This parameter limits the number of these suggestions that should actually be used.	MAX_SERVERS="4"
COURSEWARE	When establishing a courseware session, it is necessary to supply the remote agent with the name of the courseware.	COURSEWARE="BIOLOGY101"
MAX_PRICE	Negotiation options can be set on a per-site basis with this parameter. Limits the maximum amount offered for	MAX_PRICE="1.5"

	any one media.	
ADDRESS	A multi-purpose parameter used whenever an DNS or IP address should be supplied.	ADDRESS="123.12.23.3" or, ADDRESS="http://www.cnn.com/"
NAME	A multi-purpose parameter used whenever a name is required on a per entity basis – e.g. an ontology name, document name, user name, etc.	NAME="John Doe" or, NAME="doc123.html" etc.

Table 5.4 Keyword parameters that are available through a pull-down menu on the IMT canvas. Users of the IMT must be aware of the meanings and usage of these simple terms.

Itinerary Structure

The itinerary structure of *AgentSys* mobile agents is designed to support the dynamic addition and deletion of itinerary items. Unlike other systems, including Voyager [VOY97], which offer simple one-dimensional vector-like itineraries, *AgentSys* uses a *two*-dimensional vector – e.g. a vector of vectors. Figure 5.18 illustrates this concept. The first sub-vector in the itinerary is a vector of keyword and attribute combinations. These attributes capture the form-based selections made with the Media, Social, and Physical GUI's and includes information such as the users E-mail address and the type of presentation that is desired. Table 5.5 illustrates the keywords and possible attributes.

<i>Itinerary Keyword</i>	<i>Example</i>
PARENT	PARENT: ag23412
CHILD	CHILD: ag2132
ACCOUNT	ACCOUNT: 120231, 123302, 5099A12
TRANSACTION_LOG	TRANSACTION_LOG: YES
USER_AUTH	USER_AUTH: 1231333
USER_NAME	USER_NAME: Paul Smith
USER_EMAIL	USER_EMAIL: paul@aol.com
USER_CITY	USER_CITY: Chicago
USER_COUNTRY	USER_COUNTRY: USA
USER_INTEREST	USER_INTEREST: baseball, Clinton, politics
SHARE_USER_INFO_WITH	SHARE_USER_INFO_WITH: John Smith, Bill Paxton
SHARE_CARGO_WITH	SHARE_CARGO_WITH: John Smith
ASK_FOR_HELP	ASK_FOR_HELP: YES
ASK_FOR_SERVERS	ASK_FOR_SERVERS: YES
ASK_FOR_AGENT	ASK_FOR_AGENT: NO
PROVIDE_HELP	PROVIDE_HELP: NO
QUERY_AGENT_USERS	QUERY_AGENT_USERS: NO
PROVIDE_HELP	PROVIDE_HELP: YES
MESSAGE_FOR	MESSAGE_FOR: Ken Jones <crf> Hi, Ken! How are you?
SEND_STATUS	SEND_STATUS: EACH
USER_INTERVENTION	USER_INTERVENTION: NO
VIDEO_TOP	VIDEO_TOP: 20
VIDEO_BOTTOM	VIDEO_BOTTOM: 1
TEXT_TOP	TEXT_TOP: 10
TEXT_BOTTOM	TEXT_BOTTOM: 1

AUDIO_TOP	AUDIO_TOP: 10
AUDIO_BOTTOM	AUDIO_BOTTOM: 1
HTML_TOP	HTML_TOP: 10
HTML_BOTTOM	HTML_BOTTOM: 1
ASCII_TOP	ASCII_TOP: 5
ASCII_BOTTOM	ASCII_BOTTOM: 1
IMAGE_TOP	IMAGE_TOP: 15
IMAGE_BOTTOM	IMAGE_BOTTOM: 1
TOPIC	TOPIC: Chelsea Clinton
AUTHOR	AUTHOR: Williams
DATE	DATE: 01-03-98
MAX_SIZE	MAX_SIZE: 10000
MAX_DURATION	MAX_DURATION: 10
PREFERRED_DOC	PREFERRED_DOC: HTML
PREFERRED_MEDIA	PREFERRED_MEDIA: TEXT
APPLICATION_TYPE	APPLICATION_TYPE: NONE
PRESENTATION_TYPE	PRESENTATION_TYPE: NETSCAPE
MAX_PAGES	MAX_PAGES: 10
QOS	QOS: NONE
TASK_TYPE	TASK_TYPE: INFO
TERMINATION	TERMINATION: MEDIA 5
MAX_FINANCE	MAX_FINANCE: 1000
MAX_DOCS_FROM_SITE	MAX_DOCS_FROM_SITE: 2
YELLOW_PAGES	YELLOW_PAGES: 120.44.124.12
PEOPLE_FINDER	PEOPLE_FINDER: 115.23.55.1
PREFERRED_DESTINATIONS	PREFERRED_DESTINATIONS: www.cnn.com www.nbc.com

Table 5.5 Keyword instance variables within Itinerary objects encapsulate the personalized task preferences captured using graphical form-based text entry

This approach can more easily accommodate the multiple operations that may or may not have to occur at each node. The sub-itinerary (vector) contained in the first slot of the main itinerary corresponds to the operations at node 1, and so on - the 0th slot is the personalization vector. Structuring the itinerary in this fashion allows for the insertion and deletion of sub-itinerary objects. This is important because itineraries often have a dynamic nature. For example, an agent travels to a meta-index (yellow-page server) to find servers that may respond to its task. The user sets the upper limit of servers to accept (e.g. MAX_SERVERS=5) but the actual number used could range from zero to five. If two are chosen, two sub-itinerary objects are inserted in the Itinerary after the current slot.

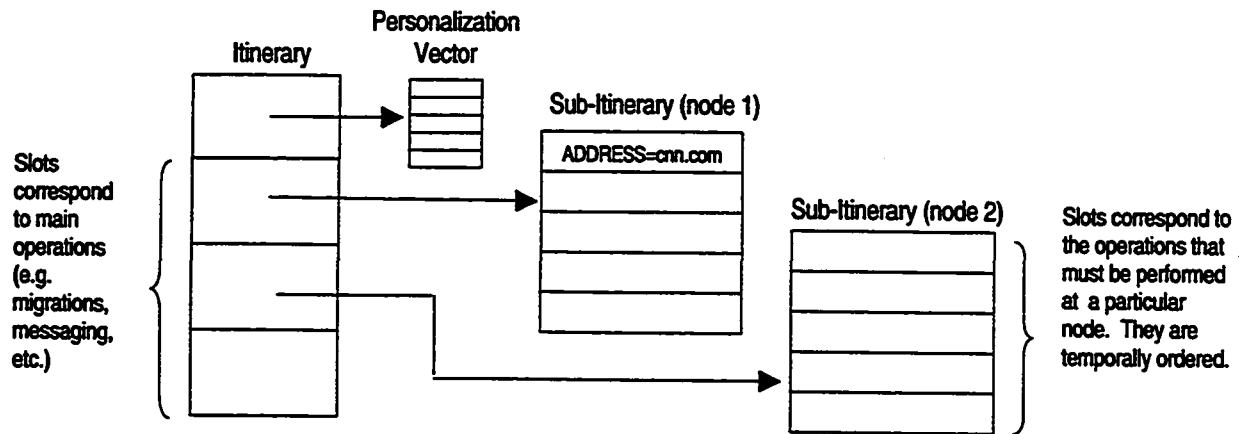


Figure 5.18 Logical structure of an agent itinerary object. The itinerary “slots” (top to bottom in the figure) order the servers (agent and Web) to which agents will migrate or send messages.

The itinerary object is a vector of vectors. The entries in the sub-vectors are application-dependent terms that convey per-site operations to the mobile agent. The scope and meaning of such terms is illustrated in table 5.5. This “internal” set of terms is an intermediate representation of the task. At the highest level, the task is represented using icons and a small set of parameter-value settings. Intermediately the task is represented in the itinerary object using these terms (table 5.6). The agent algorithm that implements the task must read from the itinerary object and interpret the terms that appear within it. To some extent, the terms represent a textual format of the features of the IMT.

5.3.4 Examples

Figure 5.19a illustrates how simply agent Itineraries can be created. It shows the creation of an Itinerary to simply access two Web servers and retrieve documents according to media and topic specifications (inputted elsewhere). The itinerary of this agent would have only one sub vector as follows: WEB=www.cnn.com; WEB=www.abcnews.com; TERMINATE;. Figure 5.19b shows a multiple hop Itinerary. The Agent first travels to a YellowPages Agent server where it determines names of other servers that are likely to satisfy the given task parameterization. It then chooses to travel to the two most likely of those servers in turn. At the first one it is to acquire the required document (if possible) and send a status message (the thicker left-pointing arrow) to the Status Monitor component of the AIS (which in turn displays this message to the user). The Agent is then to migrate to the second server, E-mail any results back to the users mailbox, send another status message, and terminate. Figure 5.21 illustrates a more complicated

itinerary. A multi-hop agent establishes a monitor with E-mail notification, migrates to two servers to acquire documents (sending media home immediately), uses human-provided data to attempt to contact another user to request a video-conference session at another node, and finally migrates to a final node to acquire media and terminate.

<i>Term</i>	<i>Description</i>	<i>Example</i>
ADDRESS	Identifies the IP or DNS name of a server.	ADDRESS="123.12.13.3"
EMAIL	Results should be sent via Email	EMAIL, or EMAIL="ben@elg.uottawa.ca"
VIDEO	A video conference negotiation is required	VIDEO
TELE_LEARN	A tele-learning course-ware is to be setup	TELE_LEARN
STORE_CARGO	Stores all acquired media at current node	STORE_CARGO
RETURN_CARGO	Returns all acquired media to the home node or to another specified node	RETURN_CARGO or, RETURN_CARGO="121.12.1.1"
YELLOW_PAGES	Force a migration to the meta-information server or to a specifically named server	YELLOW_PAGES or, YELLOW_PAGES="121.12.1.3"
WEB	Get media from a remote Web server using HTTP	WEB="www.cnn.com"
TERMINATE	Terminate the agent	TERMINATE
ACQUIRE	Acquires data from an agent or other entity type	ACQUIRE="document" or, ACQUIRE="ontology" etc.
ASK_FOR	Requests data of some form (dep. on content)	ASK_FOR="document" or, ASK_FOR="agent" etc.
EXCHANGE	Exchanges data with another entity	EXCHANGE="document" or, EXCHANGE="authentication" etc.
STATUS	Sends a status message back to the user	STATUS="I'm at cnn.com"
REMOTE_MSG	Sends a message to a remote entity (non-blocking)	REMOTE_MSG="123.12.12.32"+ "John Doe"
BLK_REMOTE_MSG	Sends a message to a remote entity (blocking)	BLK_REMOTE_MSG="123.12.12. 32"+"John Doe"
MIGRATION	Defines the migration type; non-blocking or blocking	MIGRATION="nb" or, MIGRATION="bl"
MONITOR	Creates a monitor at a node. The monitor waits for an event to occur and notifies the user when it does.	MONITOR="John Doe"
CHANNEL	Attempt to open a circuit between two nodes	CHANNEL="123.23.22.1"+"123.11. .14.3"
CLONE	Create a clone of the current agent and itinerary. The clone agent's itinerary contains specified elements from the original one (e.g. the clone performs only elements 5 and 7 from the itinerary)	CLONE="5"+"7"

Table 5.6 The grammar and examples of itinerary terminology.

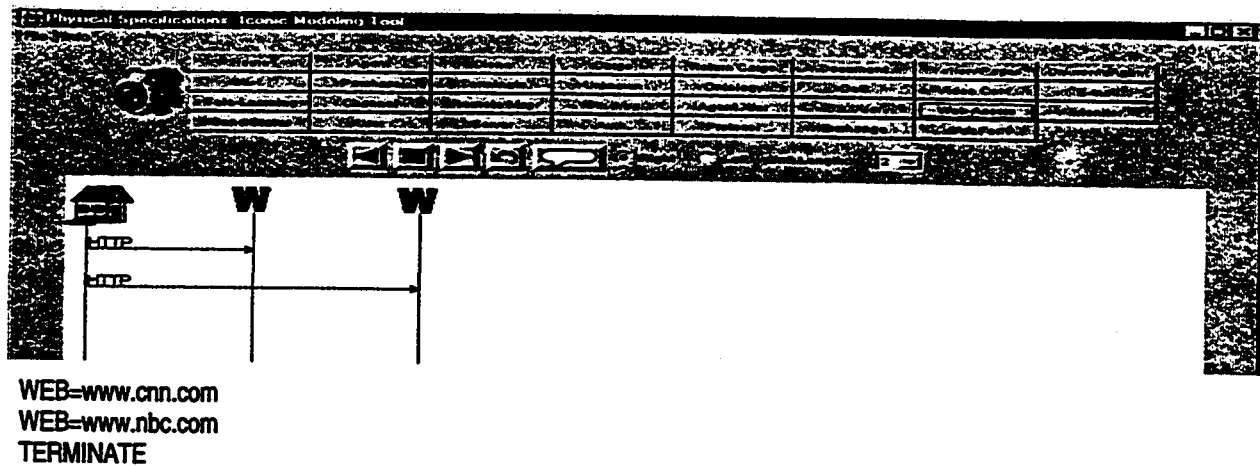


Figure 5.19a) Itinerary for accessing remote Web servers for documents. Itinerary contents are illustrated.

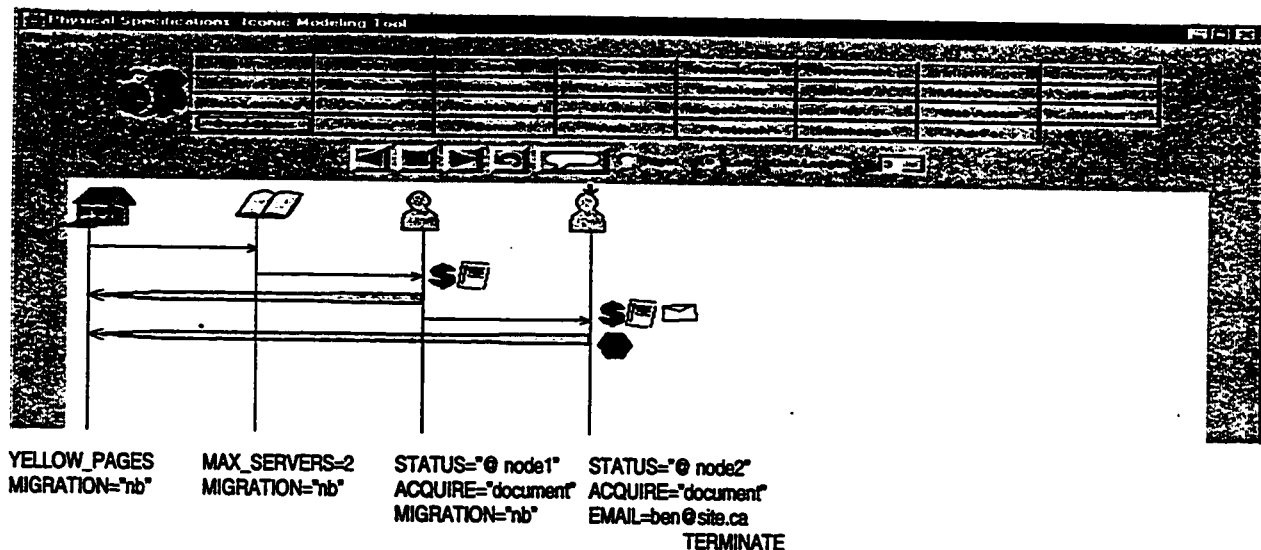


Figure 5.19b) Itinerary for multiple hops and status messaging. Itinerary contents are shown for each place.

The IMT is also ideal for creating itineraries that replace user-based tasks – particularly those types of tasks that are tedious and repetitive or both. Figure 5.20 illustrates an itinerary that can be both easily and quickly created yet serves as a “daily chore” mechanism. This itinerary is executed every day and does the following: First it spiders a particular web-site for new documents matching the document specifications. Then it migrates to a specified agent server to do the same. If documents are found, there they are sent home immediately to the user. It then

asks the Facilitator to recommend another agent server and proceeds to migrate to that server to search for similar documents. It then terminates. Such an itinerary, though small, is useful because:

- Daily execution of the task very likely finds new information at either the Web site or the agent server.
- Dynamic server-suggestion by the Facilitator allows the agent to “discover” resources
- The user is spared having to perform this task him/herself

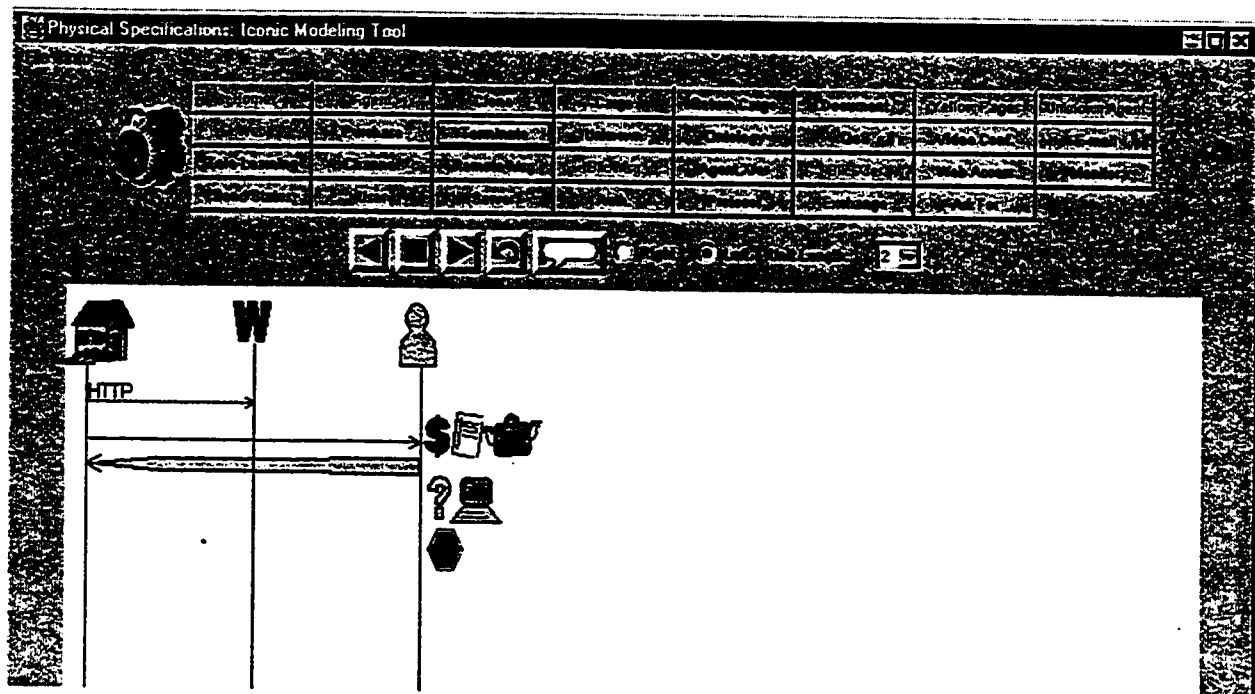


Figure 5.20 A simple but useful “daily chore” itinerary.

5.4 Summary

Generating agent behavior from forms and the graphical IMT is a multi-step process. Visually, iconic combinations and easy to use parameter names have a meaning to users. These icons and parameters are translated into a textual command language that becomes embedded in the agent’s itinerary (see figure 5.22). At each step in Figure 5.22 a translation must occur from the provided representation to an understandable one. The visual placement of icons (e.g. the *acquire* or *migrate* icons) and parameter entries (e.g. `MAX_SERVERS=2`) are translated into an

itinerary object. The itinerary contains textual commands that specify behavior. This object is “read” and interpreted dynamically when the agent executes.

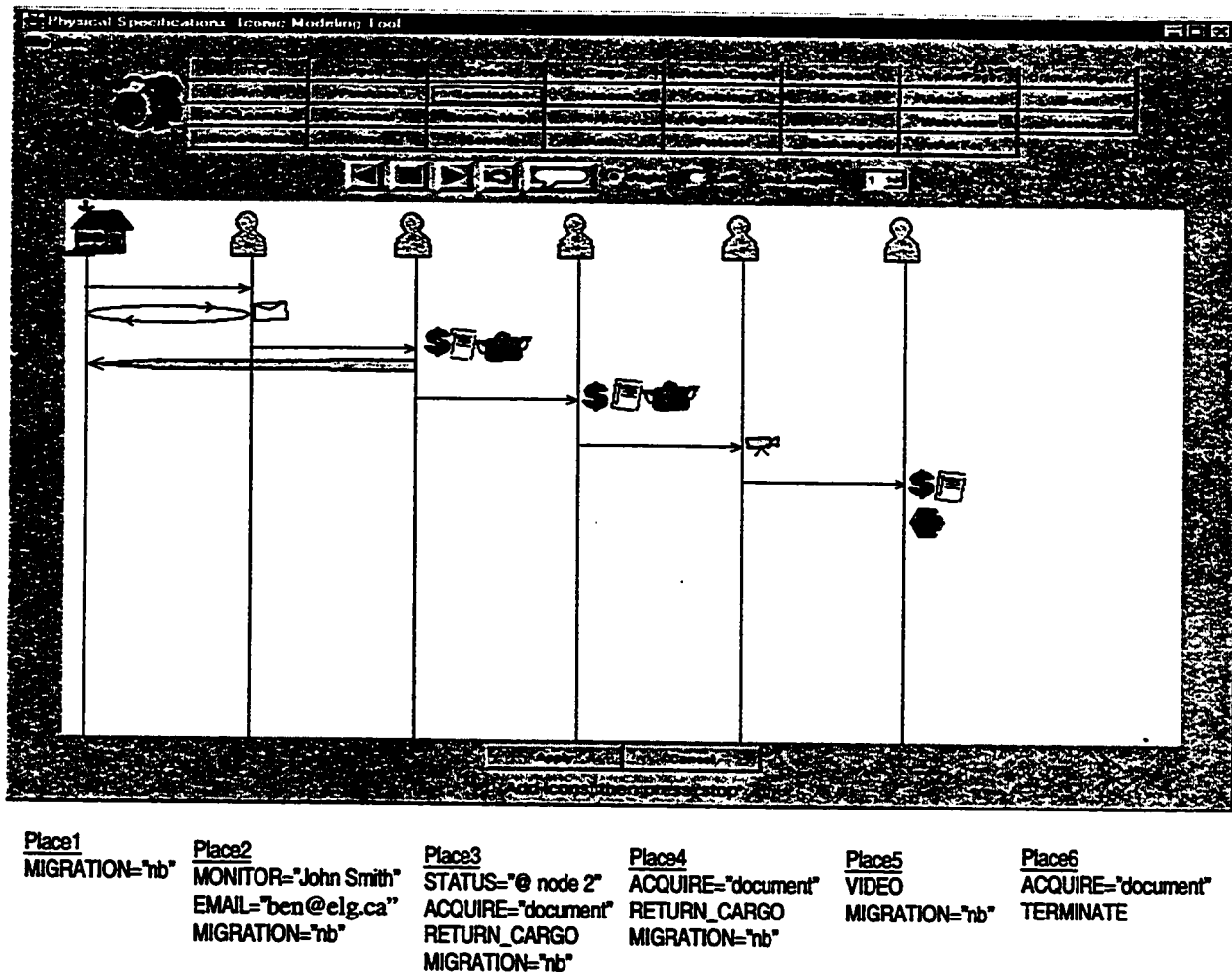


Figure 5.21 Creating a more complex itinerary. Actual itinerary Vector contents for each entry are shown below each node

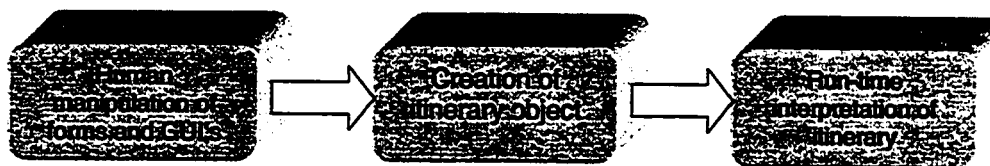


Figure 5.22 The process of agent behavior specification

A typical AIS desktop session begins with the main control GUI (figure 5.23 top left), followed by the form-based inputs (bottom and middle left), and finally the physical specification via the IMT (behind right). The *Agent Inception System* (AIS) and its *Iconic Modeling Tool* (IMT) are novel because:

- The logical and physical separation of the Agent and the Itinerary into objects makes the AIS very powerful. Both LiveAgents and Itineraries may be persisted, manipulated and re-used by the end-user.
- They provide control of Agent Itinerary information in a simple but modestly flexible fashion. *Informal* user feedback and a small number of case studies indicate that the IMT is not too difficult for novices to use.
- The IMT is more suitable for Live mobile Agent modeling than [RON97, TAK96] and [AGE97]. It is also easier and more intuitive than complex modeling tools such as SDL and Use-Case Maps (though it should be noted that SDL and Use-case maps need to be robust enough to model complex system behaviour whereas the IMT models only an agent itinerary)
- Although LiveAgents can migrate only to compliant execution environments, the integrated Web tools of the AIS mean that LiveAgents (non-migratory) can exist locally and retrieve documents even when there are no other such environments.
- The AIS can be used to create agent-based applications of diverse classes depending on the capability of the platform on which it resides— such applications could include information retrieval, courseware acquisition, videoconference setup, network management, and electronic commerce.

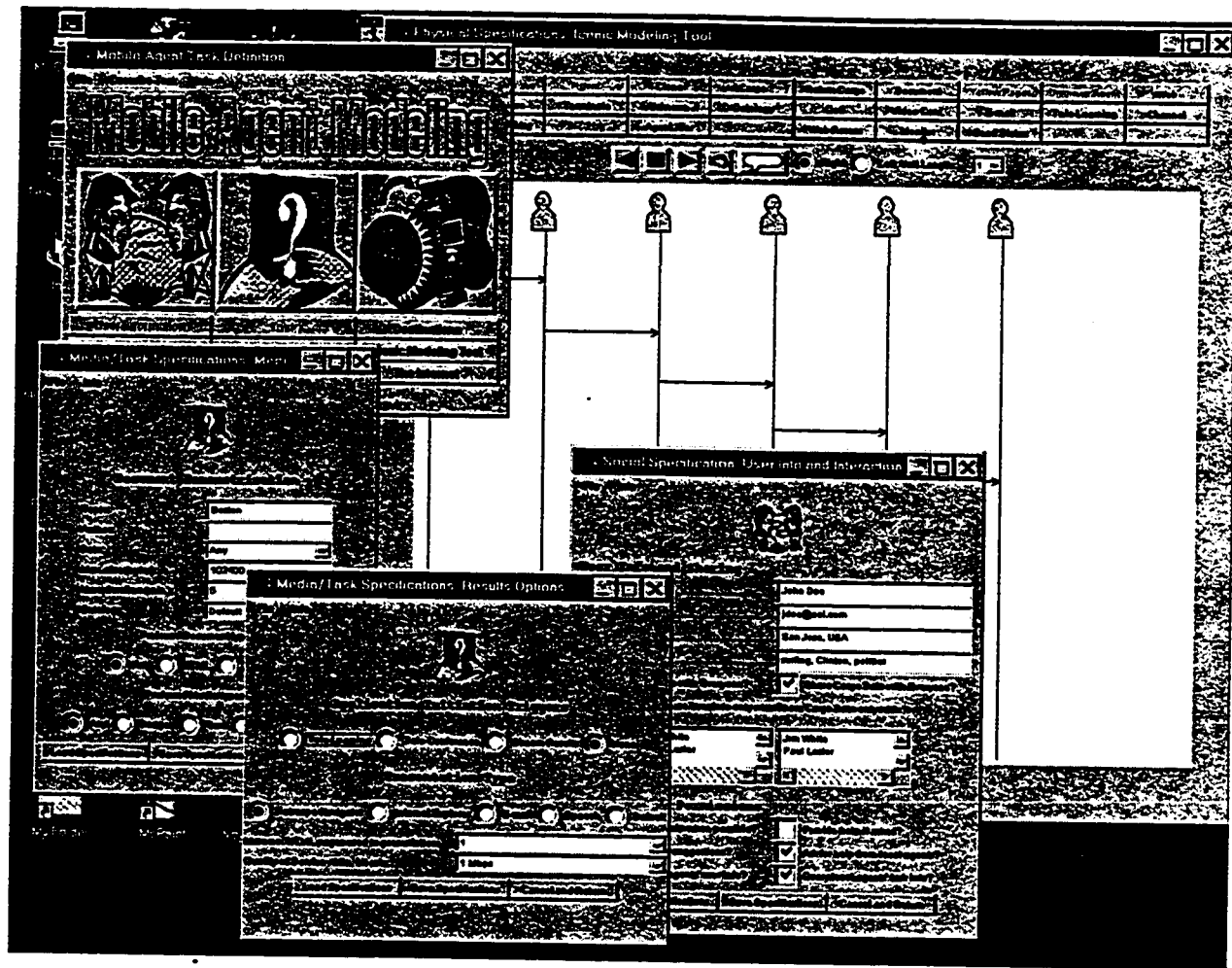


Figure 5.23 A typical AIS session involves form-based GUI's as well as the IMT.

Developing supporting tools for mobile agent activity is not simple. The currently dynamic nature of agent-related standards, emerging proprietary solutions, and workstation technology all have an influence on the design of such tools. The AIS is acceptably flexible, functional, and novel to support mobile Agent, LiveAgent, and Itinerary creation and management.

Chapter 6

Conclusions

This thesis has proposed an agent platform to enable mobile software agent-based data access, retrieval, and interaction. The problem was divided into three independent layers and the need for the design of both an *Agent Execution Environment* to support and execute mobile agents and an *Agent Inception System* to easily create and manage user agents was recognized. A unique and powerful iconic agent programming language was also proposed to reduce the task of agent programming to dragging-and-dropping icons onto an interactive canvas. An implementation on the Java JDK1.1.5 was also provided to prove the concepts and provide quantitative data.

The key to the success of inter-agent communication is in the power of the messaging system that the agent community adopts. The *AgentTransact* protocol specifies a number of speech acts (e.g. messages) and parameters based on the KQML syntax. Furthermore, *AgentTransact* extends KQML to define state-full conversations between agents using messaging and parameters. These conversations enable meaningful exchanges of information such as negotiations, media-exchanges, suggestions, and queries. While these extensions make *AgentTransact* incompatible with standard KQML, many core KQML messages are implemented so from a system point-of-view it remains a close sibling to KQML. The use of a shared ontology is also critical for communications between agents. This thesis proposed a small but useful set of hierarchically

organized terms and variables that are used by agents during messaging. This application ontology provides all agents in the community with a common representation of entities in the “real-world” such as *documents* and *media* and is proposed as a standard “agent-ontology” for any agent system. In order to transfer mobile agents, their media, and messages between agent environments, we proposed a logical and syntactic extension to the *IBM Agent Transfer Protocol/0.1* called *AgentTransfer*. These extensions added support for agent querying, cargo options, ontologies, and remote messaging and made the transfer protocol more closely mirror the semantic and needs of mobile software agents. To enable human users to interact with AgentSys we proposed *AgentTask* which is a visual protocol for specifying mobile agent behavior – the main part of this protocols implementation is the unique *Iconic Modeling Tool(IMT)*, a drag-and-drop canvas for “drawing” mobile agent itineraries. The IMT allows users to simply and quickly define distributed tasks for information agents. Supporting tools let users save, load, and modify existing itineraries for re-use. This suite of protocols composes a top-to-bottom agent platform. Although this thesis has focussed on agent-based information retrieval, AgentSys as proposed can theoretically enable a wide range of mobile agent-based applications ranging from information retrieval to electronic commerce and network management.

6.1 Limitations

Designing and implementing a platform and protocols for mobile agents is difficult due to the ever-changing nature of both agent and Internet technologies. Furthermore, this thesis and its implementation have limitations due to both time and resource constraints. They are summarized as follows – some relate specifically to aspects of this work while others relate more generally to the mobile agent paradigm:

- The IMT cannot effectively model dynamic asynchronous events or instances where the agent must decide to take one of many actions based on the current state of its environment.
- The IMT and the platform itself have a limited sense of “time”. Scheduling agent behavior to occur at specific times is not currently supported, though could be in future versions.
- Although lightweight, a full blown *AgentSys* implementation would require the agent middle-ware, ObjectSpace’s *Voyager* in this case, to be installed on all participating nodes. Nodes without a *Voyager* installation cannot host *AgentSys* agents (though they *can* serve data).

- The Facilitator uses a primitive mechanism for “suggesting” media to agents that ask for suggestions based on keywords. In fact, in this prototype, the Facilitator has access to only a very small database of media.
- In response to a request for media suggestions, a mobile agent may receive several and have to decide which ones for which it really wants to barter. Determining these media is currently done via the assumption that the first media that the Facilitator suggests are “better” than the next ones. *AgentSys* is not currently equipped to make “intelligent” suggestions based on anything more than textual keys.
- The robust set of GUIs for task specification is both a feature of the IMT and, to some extent, a limitation. Although exposing the agent paradigm to the user was deliberate, forcing the user to enter personal information and preferences can nonetheless be tedious. An approach in which the agent watches and learns from the user may be more useful. This would also let the agent be more proactive, allowing it to decide that certain information is needed and to take the initiative to go and find it.
- The IMT Status Monitor GUI effectively displays to the user the migration and execution of a single remote mobile agent but cannot currently display multiple executing agents. The GUI could be given a layered tabbed frame to enable switching between viewing different agent contexts.
- While negotiation techniques between agents allow proposing and counter-proposing, once an agent commits to something, there is no guarantee that it will be carried out. A more complex protocol such as the *Contract Net Protocol* [DAV83] could address such issues.

6.2 Suggestions for Future Research

There is tremendous potential for the mobile agent paradigm. Mobile agents are especially suited for applications like Internet commerce and information retrieval and those involving mobile users. During the course of this thesis research several issues – some specific to this implementation and some more general – have arisen which would make interesting further study:

- Use a network traffic simulation and embed the analysis engine into the Agent Inception System allowing the AIS to dynamically recommend access modes to the user. For example,

the AIS might pre-process and analyze an itinerary and recommend that the user *not* use a mobile agent for some of the sub-itineraries but rather use the Web browser manually. In this case the itinerary would be truncated before being embedded into a LiveAgent.

- Design and implement visual mechanisms such that the IMT supports the following concepts in an easily understood way: dynamic events, agent decision making (e.g. branches of control), interactions with other-user agents, interaction with remote applications.
- Create a Java applet interface to the AIS so that users may continue to manage their agent resources and information even while located away from their home base. Only a Java-enabled Web browser would be required to download the (more limited?) management applet. Agent messages and media would possibly have to be re-routed to the users current location.
- Track and monitor the progress of emerging inter-agent communication standards such as KQML and FIPA. Should either of these standards stabilize and proliferate the *AgentTransact* protocol should consider supporting both the syntax and semantics.
- Analyze security issues that put the large-scale adaptation of such a mobile agent platform in jeopardy. Exploit as much as possible both Voyager and Java mechanisms for secure object transmission and execution.
- In terms of demands on both the operating system and windowing system create a smaller version of the AIS and IMT that runs effectively on computing devices such as laptops and PDA's. This would exploit the current trend to smaller, more portable, devices.

References

- [ADO85] Adobe Systems Incorporated, <http://www.adobe.com>
- [AGE98] The Agent Society, <http://www.agent.org/>
- [BAR95] Barbuceanu M., Fox M., "COOL: A Language for Describing Coordination in Multi-agent Systems", In *First Int'l Conf. Multi-agent Systems*, 1995
- [BER93] Berners-Lee T., Connolly D., "Hypertext Markup Language-2.0, IETF HTML Working Group", available at <http://www.cs.tu-berlin.de/~jutta/ht/draft-ietf-html-spec-01.html>
- [BIM95] Bimbo A., Rella L., Vicario E., "Visual Specification of Branching Time Temporal Logic", *Proc. 11th Int. Symposium on Visual Langs.*, pp.61-68, IEEE, Germany, Oct. 1995.
- [BIR84] Birrell A., Nelson B., "Implementing Remote Procedure Calls", *ACM Trans. Comp. Systems*, 2, 1, pp39-59, Feb. 1984
- [BON98] Bonbeau E., Henaux F., Guerin S., Snyers D., Kuntz P., Theraulaz G., "Routing in Telecommunications Networks with Ant-like Agents", in Albayrak S., Garijo F. (eds.), pp.60-71, *Proc. Intelligent Agents for Telecom. Applications'98 - 2nd Int'l. Workshop*, Paris, July, Springer-Verlag, Berlin, 1998
- [BRA97] Bradshaw J., "An Introduction to Software Agents", in *Software Agents*, pp.3-46, J.Bradshaw (ed.), MIT Press, Cambridge, 1997
- [BRA97b] Bradshaw J.M., Dutfield S., Benoit P., Woolley J.D., "KAoS: Toward an Industrial-Strength Open Agent Architecture", in *Software Agents*, Bradshaw J.(ed.), MIT Press, Cambridge, 1997
- [BRE98] Breugst M., Hagen L., Magendanz T., "Impacts of Mobile Agent Technology on Mobile Communications System Evolution", *IEEE Personal Communications Magazine*, August, 1998
- [BRO91] Brooks, R.A., "How to build complete creatures rather than isolated cognitive simulators," in K. VanLehn (ed.), *Architectures for Intelligence*, pp. 225-239, Lawrence Erlbaum Associates, Hillsdale, NJ, 1991.
- [BUH97] Buhr R.J., Casselman R., *Use Case Maps for Object-Oriented Systems*, Prentice Hall, New Jersey, 1996
- [CHA97] Chang D., Covaci S., "The OMG Mobile Agent Facility: A Submission", *Proc. MA'97*, pp.98-110, Rothermel K., Popescu-Zeletin R. (eds.), Springer-Verlag, New York, 1997
- [CHE94] Chess D., Harrison C., Kershenbaum A., "Mobile Agents: Are They A Good Idea?", IBM Technical White Paper, available at <http://www.research.ibm.com/agents/publications.html>
- [CHE95] Chess D., Grosz B., Harrison C., Levine D., Parris C., Tsudik G., "Itinerant Agents for Mobile Computing", *IEEE Personal Communications*, Oct. 1995.

- [CHE96] Cheong F.C., Internet Agents: Spiders, Wanderers, Brokers, and Bots, New Riders Publishing, Indianapolis, 1996
- [COR95] OMG, "Common Object Request Broker Architecture and Specification", rev.2, August, 1995
- [DAV83] Davis R., Smith R.G., "Negotiation as a metaphor for distributed problem solving", *Artificial Intelligence*, vol.20, pp.63-100, 1983
- [DAV96] Davenport G., "Smarter Tools for Storytelling: Are They Just Around the Corner?", *IEEE Multimedia*, 3(1), pp.10-14, 1996
- [DCO98] Distributed Component Object Model Specification (DCOM/1.0), Microsoft, available at <http://www.microsoft.com/com/dcom-f.htm>
- [DEA97] Dean D., Felten E., "Secure Mobile Code: Where do we go from here?", *DARPA Workshop on Foundations for Secure Mobile Code*, Monterey, CA, March 26-28, 1997
- [DEC97] Decker K., Pannu A., Sycara K., Williamson M., "Designing Behaviors for Information Agents", *Proc. Autonomous Agents '97*, pp. 404-412, ACM, Marina Del Rey, 1997
- [DIL98] Dillenseger B., "From Interoperability to Cooperation: Building Intelligent Agents on Middleware", in Albayrak S., Garijo F. (eds.), pp.220-232, *Proc. Intelligent Agents for Telecom. Applications '98 - 2nd Int'l. Workshop*, Paris, July, Springer-Verlag, Berlin, 1998
- [FAL95] Falchuk B., Hirzaila H., Karmouch A., "A Temporal Model for Interactive Multimedia Scenarios", *IEEE Multimedia*, pp.24-31, 2(3), 1995
- [FAL96] Falchuk B., and Karmouch A., "Feature Set and Architecture of a Multimedia News System over an ATM Network", *Multimedia Tools and Applications - An International Journal*, 2(1), Kluwer Academic, pp.11-33, January 1996
- [FAL97a] Falchuk B., Karmouch A., "A Mobile Agent Prototype for Autonomous Multimedia Information Access, Interaction and Retrieval", *Proceedings of Multimedia Modeling '97*, pp.33-48, Singapore, November 1997
- [FAL97b] Falchuk B., Karmouch A., "AgentSys: A Mobile Agent System for Digital Media Access and Interaction on an Internet", *Proc. of Globecom97*, pp.1876-1880, Phoenix, November 1997
- [FAL97c] Falchuk B., Karmouch A., "Agentizing the Internet - The Practical Design of a Networked Mobile Agent System", *Proceedings of WebNet'97 - World Conf. of the WWW, Internet & Intranet*, Toronto, October 1997
- [FAL98a] Falchuk B., Karmouch A., "Saving Time and Reducing Network Traffic Using a Graphical Agent Inception System", *Proc. SBT/IEEE Int'l Telecommunications Symposium '98*, pp.560-565, Sao Paulo, August, 1998
- [FAL98b] Falchuk B., Karmouch A., "Visual Modeling for Agent-based Applications", *IEEE Computer*, Vol.31, No.12, December, 1998

- [FAL99] Falchuk B., Karmouch A., "The Mobile Agent Paradigm Meets Digital Document Technology - Designing For Autonomous Media Collection", *Multimedia Tools and Applications*, Vol.8, No.1, Kluwer Academic Publishers, January, 1999
- [FAR96] Farmer W., Guttman J., Swarup V., "Security for Mobile Agents: Issues and Requirements", *Proc. 19th Nat'l. Info. Sys. Security Conf.*, pp.591-597, Baltimore, October, 1996
- [FIN97] Finin T., Labrou Y., Mayfield J., "KQML as an Agent Communication Language", in *Software Agents*, Bradshaw J. (ed), MIT Press, Cambridge, 1997
- [FIP98] Federation for Intelligent Physical Agents, <http://www.cselt.stet.it/ufv/leonardo/fipa.htm>
- [FOR97] Ford B., Karmouch A., "An Architectural Model for Mobile Agent-based Multimedia Applications", *Proc. of Canadian Conf. Broadband Research '97*, Ottawa, April, 1997
- [FRA97] Franklin S. and Graesser A., "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", pp.21-35, *Proc. Third Int. Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, 1997
- [GEN97] Genesereth M., "An Agent-based Framework for Interoperability", in *Software Agents*, Bradshaw J. (ed), pp. 317-345, MIT Press, Cambridge, 1997
- [GRA98] Breugst M., Busse I., Covaci S., Magendanz T., "Grasshopper: A Mobile Agent Platform for IN Based Service Environments", *Proc. IEEE IN Workshop*, Bordeaux, May 10-13, 1998
- [GRE98] Greenberg M., Byington J., Harper D., "Mobile Agents and Security", *IEEE Communications Magazine*, pp.76-85, July 1998
- [GRU93] Gruber T., "A Translation Approach to Portable Ontology Specifications", *Knowledge Acquisition*, 5(2), pp.199-220, 1993
- [HOO98] Hooda A., Karmouch A., Abu-Hakima S., "Agent Negotiation for Supporting Personal Mobility", in Albayrak S., Garijo F. (eds.), pp.190-203, *Proc. Intelligent Agents for Telecom. Applications '98 - 2nd Int'l. Workshop*, Paris, July, Springer-Verlag, Berlin, 1998
- [IBM97] IBM Tokyo Research Laboratory, <http://www.ibm.co.jp/trl/aglets>
- [IEEE97] IEEE Internet Computing Magazine, 1(2), March-April, 1997
- [ISO84] International Organization for Standardization, "Basic Reference Model for Open Systems Interconnection", ISO7489, 1984
- [JAC96] Jacobs N., Shea R., "The Role of Java in InfoSleuth: Agent-based Exploitation of Heterogeneous Information Resources", available at <http://www.mcc.com/projects/infosleuth>, 1996
- [JEN95] Jennings N., Wooldridge M., "Intelligent Agents: Theory and Practice", *The Knowledge Engineering Review*, 10 (2), 115-152, 1995.

- [JEN98] Jennings N., Sycara K., Wooldridge M., "A Roadmap of Agent Research and Development", *Autonomous Agents and Multi-agent Systems*, 1, pp.275-306, Kluwer Academic Publishers, Boston, 1998
- [KAC98] The KACTUS Project Documentation, University of Amsterdam, Dept. Social Science Informatics, available at <http://www.swi.psy.uva.nl/projects/NewKACTUS/>
- [KAR97] Karjoth G., Lange D., Oshima M., "A Security Model for Aglets", *IEEE Internet Computing*, 1(4), pp.68-77, July-August, 1997
- [KIN97] Kinnery J., Zimmerman D., "A Hands-on Look at Java Mobile Agents", *IEEE Internet Computing*, 1(4), pp.21-30, July-August, 1997
- [KNO97] Knoblock C., Ambite J., "Agents for Information Gathering", in *Software Agents*, Bradshaw J. (ed), MIT Press, Cambridge, pp.347-372, 1997
- [LAU91] Laurel B., *Computers as Theatre*, Addison Wesley Publishing Co., New York, 1991
- [LUK97] Luke S., Spector L., Rager D. and Hendler J., "Ontology-based Web Agents", *Proc. Autonomous Agents '97*, pp.59-66, Marina Del Rey, February, 1997
- [LYC98] LycosTM Search Engine, <http://www.lycos.com>
- [MAE94] Maes P., "Agents that Reduce Work and Information Overload", *Communications of the ACM*, pp.30-40, 37(7), 1994
- [MAG97] Magedanz T., "Mobile agents – Basics, Technologies, Standards and Applications", Workshop Tutorial, *IEEE IN Workshop '97*, Colorado Springs, May, 1997
- [MIT98] Mitsubishi Electric ITA, "Concordia: An Infrastructure for Collaborating Mobile Agents", available at <http://www.meitca.com/HSL/Projects/Concordia>
- [NAD97] Nodine M., Unruh A., "Facilitating Open Communication in Agent Systems: the InfoSleuth Infrastructure", in *Intelligent Agents IV - Proc. 4th Int'l Workshop - ATAL '97*, Singh, Rao, Wooldridge (Eds.), pp. 281-295, Providence, July 24-26, 1997
- [ODY98] Odyssey API, General Magic, available at <http://www.genmagic.com/agents>
- [OMG91] Object Management Group, Common Object Request Broker Architecture and Specifications, document number 91.12.1, 1(1), 1991
- [OUS94] Ousterhout J.K.(ed.), *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, New York, 1994
- [PHA98] Pham V.A., Karmouch A., "Mobile Software Agents: An Overview", *IEEE Communications Magazine*, pp.26-37, July 1998
- [PIT95] Pitt J., Anderton M., Cunningham J., "Normalized Interactions Between Autonomous Agents: A Case Study in Inter-Organizational Project Management", available at <http://medlar.doc.ic.ac.uk>, 1995

- [REP95] Repenning A., Sumner T., "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages", *IEEE Computer*, 28(3), March, 1995, pp.17-25
- [ROB95] Robinson A., Cook D., Tanimoto S., "Programming Agents with Visual Rules", *Proc. 11th International Symposium on Visual Langs.*, pp.13-20, IEEE, Germany, Sept. 1995.
- [ROB96] Network Working Group INTERNET DRAFT, "A Method for Web Robots Control", available at <http://info.webcrawler.com/mak/projects/robots/norobots-rfc.html>
- [RON97] Ronnquist R., Low C., "Analyzing Expert Assistants through Interaction Diagrams", *Proc. Autonomous Agents'97*, pp. 500-501, ACM Press, Marina Del Rey, 1997
- [RUS97] Rus D., Gray R., Kotz D., "Transportable Information Agents", *Proc. Autonomous Agents'97*, pp. 229-236, ACM Press, Marina Del Rey, 1997.
- [SDL92] Specification and Description Language (SDL), CCITT Z.100, International Consultative Committee on Telegraphy and Telephony, Geneva, 1992
- [SHO93] Shoham Y., "Agent Oriented Programming", *Artificial Intelligence*, 60(1), pp.51-92, 1993
- [SHO97] Shoham Y., "An Overview of Agent-Oriented Programming", in *Software Agents*, Bradshaw J. (ed), MIT Press, Cambridge, pp.217-289, 1997
- [SMI97] Smith D., Cypher A., Spohrer J., "KidSim: Programming Agents without a Programming Language", in *Software Agents*, Bradshaw J.(ed.), MIT Press, Cambridge, pp.165-190, 1997
- [STA94] Stallings W., *Data and Computer Communications*, Prentice-Hall, New Jersey, 1994
- [STA90] Stamos J., Gifford D., "Remote Evaluation", *ACM Trans. Prog. Lang. And Systems*, pp537-565, 12(4), Oct. 1990
- [STO94] Stoyenko A., "SUPRA-RPC: Sub-program PaRAMeters in Remote Procedure Calls", *Software Practice and Experience*, pp.27-49, 24(1), Jan. 1994
- [SUN94] Sun Microsystems, The Java Language: A White Paper, *Sun Microsystems White Paper*, available at <http://www.sun.com>, 1994
- [SUN98] The Source for Java Technology, available at <http://java.sun.com>, 1998
- [SYC96] Sycara K., Pannu A., Williamson M., Zeng D., Decker K., "Distributed Intelligent Agents", *IEEE Expert*, pp.36-47, 11(6), 1996
- [TAC95] Johansen D., van Renesse R., Schneider F., "An Introduction to the TACOMA Distributed System Version 1.0", available at <http://dslab3.cs.uit.no:1080/Tacoma/index.html>, 1995
- [TAK95] Takeda H., Iino K., Nishida T., "Agent Organization and Communication with Multiple Ontologies", *Int. Journal of Cooperative Information Systems*, 4(4), pp.321-337, Dec. 1995
- [TAK96] Takeda K., Inaba M., Sugihara K., "User Interface and Agent Prototyping for Flexible Working", *IEEE Multimedia*, pp. 40-50, 3(2), Summer, 1996

- [UMA93] Umar A., Distributed Computing: A Practical Synthesis, Prentice-Hall, New Jersey, 1993
- [VOY97] Voyager Core Package Technical Overview, Object Space, available at <http://www.objectspace.com/voyager>
- [WEI96] Weitzman L., Wittenburg K., "Grammar-based Articulation for Multimedia Document Design", *Multimedia Systems*, pp.99-111, Springer Verlag, New York, 1996
- [WHI94] White J.E., "Telescript Technology: The Foundation for the Electronic Marketplace", General Magic White Paper, General Magic, available at <http://www.genmagic.com>, 1994
- [WOO98] Wooldridge M.J., Jennings N.R., "Pitfalls of Agent-Oriented Development", *Proc. 2nd Int'l. Conf. on Autonomous Agents*, pp.385-391, Minneapolis, USA, 1998
- [YAH98] YahooTM Search Engine, <http://www.Yahoo.com>

Appendix A

Pseudo-Code

The following is pseudo-code describing the algorithms and operation of various entities in the AEE and AIS systems. All of these entities were implemented completely in Java using the JDK1.1.5. Additional explanations are provided when the commented pseudo-code is not enough to describe the operation. These entities are also referenced and explained fully in the body of this thesis.

FacilitatorAgent.class

The facilitator maintains an internal message board for its own purposes and to allow agents to message asynchronously. The ResponseBoard is where it posts all its responses to agents (e.g. rather than messaging directly with them). When the agent asks for the response to its query (e.g. sends a receive_msg) the facilitator then reads from its response board. It also maintains tables of Offers made to agents, and Suggestions to be made to agents. This creates a persistent states while still allowing asynchronous communication between agent and facilitator.

```
Public class FacilitatorAgent extends Agent{

Public class FacilitatorAgent(){

    Super();
    Populate the media database with pricing, etc.;

} //constructor

public agentMessage message(perf, ontology, sender, receiver, cont,...){
    //Perf is the performative sent by the agent.
    //Supports following in-coming msgs: help_with_doc, inform, acquire
    // receive_msg

    if(perf = receive_msg) Return message from interal response board;

    else if(perf = help_with_doc){
        if(no media available here) return "Decline";
        else{
            build suggestions vector for this calling agent;
            get first suggestion;
            if(suggestion = null) return "decline";
            else post suggestion to the response board;
        }
    } //help_with_doc

    else if(perf = inform){
```

```
        if(cont = more){
            get next suggestion for this agent;
            if(no suggestions left) return "decline";
            else return "suggest" and suggestion data;
        }
        if(cont = done) clean up all internal data for this agent;
    } //inform

    else if(perf = acquire){
        store agreed upon price for this media;
        post ack to response board;
    } //acquire

    else if(perf = negotiate_for){
        if(amt_offered by agent acceptable){
            record a sale; post response to board;
        }
        else{
            perf = ctr_propose;
            make a counter proposal;
            store the offer; post the response;
        }
    } //negotiate_for

    else if(perf = decline and policy = negotiate_for) clean up data;

    else if(perf = accept and policy = negotiate_for){
        clean up data;
        record the sale;
    }

    else if(perf = ctr_propose and policy = negotiate_for){
        if(agent offer less than low price) post a decline;
        if(agent offer acceptable) post an accept;
        if(agent offer not good enough) post a new offer in ctr_propose;
    }

    else if(perf = suggest_server) suggest a server in suggest message;

    else return "unsupported performative!"

} //message

public String extractFromContent(data, pos){

    //This extracts the 3 parameters out of an input like "media, car, 2"
    //This is very useful for parsing KQML-type parameter passing

    type = data.substring(0,data.indexOf(", "));
    name = data.substring(data.indexOf(", ")+2,data.lastIndexOf(", "));
    value = data.substring(data.lastIndexOf(", ")+2,data.length());

    if(pos==0) returnedvalue=type;
    if(pos==1) returnedvalue=name;
    if(pos==2) returnedvalue=value;

    return returnedvalue;
}
```

```
//extractFromContent

private void PostToResponseBoard(agentMessage){
    post agentMessage to the internal Response Board
} //PostToResponseBoard

private int getLowLimit(mediaName){
    return the lowest value that the named media can be sold for;
} //getLowLimit

private int getStartingPrice(mediaName){
    return the value that the named media should start bidding at;
} //getStartingPrice

private int getLastOffer(agentName, mediaName){
    return the last offer made by agentName for the named media;
} //getLastOffer

private void storeOffer(agentName, mediaName, price){
    remove any previous offers by this agent for this media;
    add new offer for this agent and this media for this price;
} //storeOffer

private void RecordSale(agentName, mediaName, price){
    add a record to Sales Vector for this agent, this media, and price;
} //recordSale

private int getRecordedSalePrice(agentName, mediaName){
    price = the price agreed upon for this agent and this media;
    return price;
} //getRecordedSale

private agentMessage getAndDeleteResponseFromBoard(name, policy){
    get the first occurrence of an agentMessage matching name,policy;
    return null if not found;
    make temporary copy of this record;
    delete the permanent record;
    return (a copy of) the response;
} //getAndDeleteResponseFromBoard

private suggestion getAndDeleteSuggestion(name){
    if no next suggestion for agent with name=name, return null;
    get next suggestion, delete it from Vector, and return it;
} //getAndDeleteSuggestion

private void BuildSuggForAgent(agentname, content){
    //This method is open and can be customized to add more intelligence
    //and robustness. For now it simply adds some canned suggestions.
    add one or more suggestions to the Suggestion Vector;
} // BuildSuggForAgent

private void cleanSuggestions(agent){
    delete any suggestions remaining for agent;
} //cleanSuggestions

private void cleanResponseBoard(agentName, policy){
```

```

        delete any responses remaining for agentName with policy;
    }//cleanResponseBoard

    private void cleanOffers(agent){
        delete any offers still remaining for agent;
    }//cleanOffers

}//facilitatorAgent

```

liveAgentMgr.class and WAMthread.class

The live agent manager is a GUI to support user-driven interactions on stored classes of agents. The user either launches a live mobile agent that is already stored (e.g. has an itinerary) or binds an agent instance with a named itinerary and then launches it. The WAMthread class is used to create a mobile agent in a separate thread of control to allow the GUI to continue to work at full speed

```

public class liveAgentMgr extends Frame{

    public LiveAgentMgr(WebAccessManager, StatusManager){

        build GUI for this Manager;
        fill the Itinerary list;
        fill the agent Class list;
        fill the live Agent list;

    }//constructor

    public boolean BindSave() {
        create new Live Agent with named class and itinerary;
        persist new agent;
    }

    public boolean BindSaveLaunch() {
        create new Live Agent with named class and itinerary;
        persist new agent;
        launch new agent;
    }

    public boolean LaunchFromDB() {
        load named Live Agent from database;
        launch agent;
    }

    public void invokeWAM() {
        WAMThread = new WAMThread(WebAccessor, taskName, agentName);
    }

    public void invokeMobileAgent() {
        MobileAgentThread = new MobileAgentThread(Itinerary, agentName);
    }

    public void readItineraryFromDisk() {
        Create new itinerary Vector;
    }
}

```

```

        Open named itinerary file from disk;
        For each sub-itinerary read
            Add to Itinerary;
        }//for
    }

```

```

} //liveAgentMgr

```

```

//-----

```

WAMThread is a thread class that it used to send a web-based task to the WebAccessor Manager. For example, when a web-site is to be spidered or Yahoo™ to be quick-searched. This simply takes the load off of the LiveAgentMgr and allows it to continue processing while the spider works in an independent thread.

```

class WAMThread extends Thread{
public WAMThread(WebAccessor, taskName, agentName){
    super();
    start();

    run(){
        WebAccessor.sendTask(taskName, agentName);
    } //run
}
} //WAMThread class

```

```

//-----

```

The MobileAgentThread is a thread class that is used to launch a new mobile agent instance. This simply allows the LiveAgentMgr to continue without delays or blocking.

```

class MobileAgentThread extends Thread{
public MobileAgentThread(ItineraryVector, agentName){
    super();
    start();
} //constructor

    run(){
        mobileAgent = new MobileAgent(agentName, homeIP, itinerary);
        mobileAgent.launch();
    }

} //MobileAgentThread

```

MobileAgent.class

The *MobileAgent* class is an extension of the *Voyager Agent* class. This gives it several complex capabilities including a *moveTo* command, *dieNow*, etc. The *atAEE* method is executed when the agent arrives at any AEE. The mobile agent follows its programmed itinerary. The only exception is when it asks for a server-suggestion from a *Facilitator*. In this case the agent receives a previously unknown AEE and inserts it into its itinerary in order to visit it. The mobile agent is started with a name and an *Itinerary*.

```

MobileAgent extends Agent{

Public MobileAgent(name, home_IP, Itinerary){

    Extract Address from the Itinerary into AddressBook;
    Extract username, city, interests, email, topic, author, date;
    Do any local processing or information gathering;

} //constructor

public void launch (){

    next();

} //launch

private void next(){

    if(more addresses to visit in this itinerary){
        get next address from AddressBook;
        inform home base that we are going to move;
        moveTo the next address then execute atAEE method
    else
        idle();
    }

} //next

//the atAEE method is executed at each AEE that the agent visits. At each
AEE, only the portion of the itinerary intended for this server is executed.

public void atAEE(){

    inform the home base we have arrived;
    parseItineraryComponent();
    next();

} //atAEE

// parses a sub-itinerary. The main itinerary contains sub-itineraries
describing the itinerary at each AEE. This method reads the correct
itinerary and carries it out.

public void parseItineraryComponent(){

    //read the sub-itinerary intended for this AEE. Supports the
    //following keys: STATUS, ACQUIRE, TERMINATE, MIGRATION, ADDRESS, WEB,

```

```
// ASK_FOR=server, ASK_FOR=document

for(each entry in this sub-itinerary){
    switch key:
    case(TERMINATE): dismiss this agent;
    case(STATUS): send the status message to home base;
    case(ACQUIRE) helpWithDocConversation();
    case(ASK_FOR=server) askForConversation(server);
}

}

}

//parseItineraryComponent

//implement the help_with_doc conversation here. This is a stateful exchange
of messages with the facilitator in order to acquire (hopefully) helpful
media that fit the document parameters.

public void helpWithDocConversation(){

    perf=help_with_doc; cont=media keywords;

    while(true){
        facilitator.message(perf, cont, ...);
        if(no response is needed) break;
        returnedMsg = facilitator.message(receive_msg,...);
        if(returnedMsg = "decline") break;
        if(returnedMsg = "suggest"){
            add this suggestion to our list;
            if we have enough suggestions
                perf = "inform"; cont = "done";
            else perf = "inform"; cont = "more";
        }
    }

    Decide which suggestions are worth keeping;
    For( each suggestion from facilitator) NegotiateFor(suggested media);
    For( each suggestion from facilitator) Acquire(media);

}

//implement the negotiate_for protocol. After a media is suggested, the
agent must commence a negotiate_for conversation to actually bid for it. The
agent and facilitator exchange bids until one or the other quits or an
agreement is made.

public void negotiateFor(){

    perf=negotiate_for; cont=mediaName; offer=10;
    while(true){
        facilitator.message(perf,cont,offer,...);
        if(no response needed) break;
        returnedMsg = facilitator.message(receive_msg,...);
        if(returnedMsg = reject or accept) break;
        else{
            if (this counter proposal acceptable) perf="accept";
            else make a counter offer to facilitator;
        }
    }
}
```

```

} // negotiateFor

// this implements the ask_for protocol. Here we implement only the
ask_for(server) protocol in which the agent request that the facilitator
suggest another AEE to visit.

public void askForConversation(){

    // implements only the "ask_For(server)" message
    returnedMsg = facilitator.message(suggest_server,...);
    returnedMsg = facilitator.message(receive_msg,...);
    if(returnedMsg = decline) return;
    else{
        insert the suggested server address into address_book;
        continue processing this itinerary;
    }
} // askForConversation

```

ModelingTool.class and supporting Classes

A class for displaying images. It places the Image into a canvas so that it can moved around by layout managers, will get repainted automatically, etc. No mouseXXX or action events are defined, so it is most similar to the Label Component.

```

public class ImageLabel extends Canvas {
} // class ImageLabel

```

```

//-----

```

A class that supports Images as Components that can be repositioned with the mouse. As with all Components, the effects may be eventually undone by the layoutManager if it is not off (null) or chosen appropriately.

```

Public class Icon extends ImageLabel{

Public Icon(){
    SetDefaults();
} // constructor

public boolean handleEvent(event){
    if(ignoreEvents=true) return true;
    else return the parents handleEvent result;
} // handleEvent

public boolean handleIconEvent(event, container){
    // called from handleEvent method of the parent Container (e.g. canvas)
    if mouse is not over an icon then return;
    translate the icon along with the mouse movement;
} // handleIconEvent

public boolean mouseDown(event, x, y){
    draw border around icon;

```

```

}
public boolean mouseUp(event, x, y){
    remove border from icon;
}
public boolean mouseDrag(event, x, y){
    center the icon around x,y;
}

```

```

} //Icon

```

```

//-----

```

An extension to the Panel class to allow us to add Icon instances to a blank "canvas"-like area. To support draggable icons however, the `handleEvent` method must call the appropriate method from the Icon class.

```

public class MyDrawingPanel extends Panel{

public boolean handleEvent(Event e){
    //below line added to deal with Icon class
    if (Icon.handleIconEvent(e, this)) return(true);
    //above line for Icon class only - regular code follows
    return true;
} //handleEvent
} //MyDrawingPanel

```

```

//-----

```

This class is used to provide a customized selection of web-pages and IP's to the user of the mobile agent modeling system.

```

public class MyEntry extends Dialog{

public MyEntry() {
    build the entry GUI;
    populate the selection pull-down with the most often used URL's and IP
} //constructor

} //MyEntry

```

```

//-----

```

`MyTextArea` is simply a frame to display messages to the user

```

public class MyTextArea extends Frame{

public MyTextArea() {
    build GUI;
} //constructor

public boolean set(message){
    set the GUI to display the message;
} //set
} //MyTextArea

```

```

//-----

```

```
Public class ModelingTool extends Frame{

Public ModelingTool(parent, ItineraryName){

    Get handle to ItineraryManager parent;
    Create the top panel for the matrix of buttons;
    Create the MyDrawingPanel instance for the drawing panel;
    Create other panels for buttons;
    Initialize;

    Stage1();

} //constructor

//in the stage 1 method we prepare the GUI for the addition of "places" onto
the canvas. In this stage the user simply adds agent-servers, web-servers,
home, or yellow-page places.

boolean stage1(){
    set the label for user's benefit;
    set STAGE variable to 1;
    disable all buttons;
    enable agent, unknown-agent, home, web, yellow-pages
    disable all tool buttons;
    enable stop button;
} //stage1

//in the stage2 method, the user has finished adding the physical places to
the canvas and is adding place-information to them. This includes IP
addresses, URL addresses, and other information. The user utilized the "left"
and "right" tool buttons to navigate between the places he has added.
Furthermore, we now know the total number of places so the Itinerary vectors
can be initialized.

boolean stage2(){
    set the label for the users benefit;
    VEC = new Vector(NumberOfPlaces);
    Vector[] SUBVEC = new Vector[NumberOfPlaces];
    For each place{
        SUBVEC[place] = new Vector(maxOperations);
        VEC.addElement(SUBVEC[place]);
    } //for

    disable all buttons;
    disable all tool buttons;
    enable bubble button for adding place-information;
    enable forward, back, and stop tool buttons;
} //stage2

// in stage 3 method all places and place information is entered and the user
is adding icons to the itinerary.

boolean stage3(){
    set the label for users benefit;
    enable all buttons;
} //stage3
```

```
//this method executed whenever stop tool button pressed
boolean stop_method(){
    if(STAGE==1) STAGE=2;
    else if(STAGE==2) STAGE=3;
}

//this method is called whenever the bubble is pressed. The user presses the
bubble either to add icon-information (parameters) or to add place-
information.

boolean bubble_method(){
    if(STAGE==2){
        create new MyEntry instance with label, "Enter place info";
    }
    if(STAGE==3){
        create new MyEntry instance with label, "Enter icon Parameter";
    }
}

//this method used to add elements into the itinerary
boolean setItin(position, value){
    //position is the sub-vector identification
    //value is the string to add to this SUBVEC
    add value to the SUBVEC at position;
}

//pass the itinerary up to the Itinerary Manager. It is at the Itinerary
Manager that it may be persisted to disk.

boolean apply_method(){
    if(STAGE==1 or STAGE==2) return "scenario not yet complete" error;
    else{
        for each sub-itinerary element pare it down if necessary;
        pass itinerary upto the parent - e.g. the ItineraryManager;
    }
}

public boolean action(event, arg){

    if(apply button) apply_method();
    if(go button) go_method();
    if(back button) back_method();
    if(stop button) stop_method();
    if(bubble button) bubble_method();

    if any other icon button is pressed{
        add icon to canvas;
        set appropriate variables;
        add appropriate information to Itinerary Vector;
        if (icon button is bubble and previous icon was send status){
            add a "status=message" to the itinerary;
        }
        if (icon button is bubble and STAGE==2){
            add an "ADDRESS=data" to the itinerary;
        }
    }
}
```

```

} //action

} //ModelingTool class

```

StatusAgent.class and StatusMgr.class

```

//This class is simply a user-side liason between the remotely executing
agent and the static StatusMgr instance that posts messages to the status
board.
//
//          msg                      msg
// remote_agent ----> statusAgent -----> StatusMgr

public class statusAgent extends Agent{

public statusAgent(StatusMgr){
    postMessage("started.");
} //constructor

public void postNull() {
    statusMgr.agentnull();
}

public void postMigratingTo(name, location){
    statusMgr.agentMigratingTo(name,location);
}

public void postAt(name, location){
    statusMgr.agentAt(name,location);
}

public void postMessage(message){
    statusMgr.setStatus(message);
}

} //statusAgent

//-----

// The status manager is a GUI that visually displays status messages from
remote agents. These messages are expressed textually on the canvas or, in
the case of agent migration, visually as a moving agent-icon on a conceptual
network graphic.

public class statusMgr extends Frame{
public statusMgr(){
    build GUI;
    agentNull();
} //constructor

public boolean send_message(message, agent){
    if (agent unreachable) return error;
    else send message to remote agent;
}

public void setStatus(message) update the GUI and append the message;

```

```

public void agentMigratingTo(name,location){
    update the GUI with the name of location;
    update the icon on the GUI to indicate migration;
}

public void agentAt(name,location){
    update the GUI with the name of location;
    update the icon on the GUI to indicate agent at AEE;
}

public void agentNull() {
    update the GUI with text;
    update the icon on the GUI to indicate agent in waiting mode;
}

```

Supporting Class structures

A *suggestion* is a structure that encapsulates an agents name and a suggested element. The facilitator uses these structs to record the suggestions that it has made to an agent.

Class **suggestion** extends Object{

```

    String AgentName;
    String data;

    Suggestion(name, info){
        This.AgentName = name;
        This.data = info;
    }
}

```

//-----

An *offer* is a structure that encapsulates an agent name, media name, and the value of an offer. This struct is used by the facilitator to record its offers made to agents during a negotiation for a media.

```

public class offer{

    String agentName;
    String mediaName;
    int    lastOffer;

    public offer(agent, media, offer){

        this.agentName = agent;
        this.mediaName = media;
        this.lastOffer = offer;

    } //offer constructor
}

```

//offer Class def'n

//-----

A *mediaLimit* is a structure that is used to organize media. It encapsulates a media's name and that media's associated starting sale price and lowest limit price. The start price is first selling price offer that the facilitator will make for this media. The low limit is the lowest it will go for this media before declining altogether.

```
public class mediaLimit{

    String mediaName;
    int    startAt;
    int    lowLimit;

    public mediaLimit(name, start, low){

        this.mediaName = name;
        this.startAt = start;
        this.lowLimit = low;

    }//mediaLimit constructor

} //mediaLimit Class def'n
```

```
//-----
```

An *agentMessage* is a structure for encapsulating the KQML parameters of a message as well as the content. In lieu of a KQML parser. The *agentMessage* is used by the *mobileAgent* when it sends messages to the facilitator or other agents. It fills out the variables then passes the structure to the receiver. The facilitator does likewise when returning messages to agents. Only the basic KQML parameters are used here to keep it simple. *In_reply_to* and *reply_with* are under-utilized by design.

```
public class agentMessage{

    String performative;
    String ontology;
    String sender;
    String receiver;
    String in_reply_to;
    String reply_with;
    String content;
    String policy;

    public agentMessage(){

        this.performative="";
        this.ontology="";
        this.sender="";
        this.receiver="";
        this.in_reply_to="";
        this.reply_with="";
        this.content="";
        this.policy="";

    } //agentMessage constructor
```

```
//agentMessage Class def'n
```

WAM.class

//The WAM is a graphical tool that both displays the status of Web-based agent tasks and lets users to Quick searches of the internet. It invokes the WebGrep class to spider web pages. The WAM is invoked by the LiveAgentManager who passes it an itinerary name which the WAM reads from the storage and passes to the WebGrep.

```
WAM extends Frame{
```

```
Public WAM(){
```

```
    Build the GUI for this class;
```

```
    }//constructor
```

```
public boolean action(event, arg){
```

```
    if(clear button) clear the fields;
    if(Yahoo button) start a WebGrep with Yahoo parameters;
    if(Excite button) start WebGrep with Excite parameters;
    if(cbc button) load the CBC NewsWorld page;
```

```
}//action
```

```
public boolean sendTask(itineraryName, agentName){
```

```
    start the flashing globe thread;
    open the named itinerary file from disk;
    while (read itinerary){
        create Keywords vector of task-keys;
        create WebSites vector containing each WebSite to spider;
    }
```

```
    for(each webSite to spider){
        for(each keyword to search for){
            instantiate WebGrep for this sub-task;
        }
    }
```

```
}//sendTask
```

WebGrep, Spider, and related p-code

WebGrep implements a search of a web-tree of files to find files that contain a pattern. It searches all files reachable underneath the root given. It also obeys the Robot exclusion protocol and will not intrude where it should not go. The search is basically **depth-first** starting with the file given. The program builds a **to-do-queue** of pages to visit and maintains a **done-queue** of all the pages it has seen so that it does not repeat pages. The sizes of these queues affect the operation on a rule-of-thumb basis. We use a to-do


```

        Increment hit count; If past MAXHITS then break;
        Else if line contains "<form" break;
    }//while
    writeHTMLfile();

} //searchExcite

public void findIt(pattern, spider){
    create and nullify new thread array threads[nThreads];
    for(;;){
        nAlive=0;freeSlot=-1;
        for(j=1 to nThreads){
            if thread[j]==null freeslot=j;
            else if thread[j] notalive then freeslot=j; thread[j]=null;
            else ++nAlive;
        }//for

        if(no threads still alive and no more elements to spider) break;
        if(free slot avail. And more to spider to){
            threads[freeSlot] = new WebGrepThread(pattern,spider,this);
        }//for
    }//findIt

public void writeHTMLfile(){
    open a file with name "test_web.html" where test is replaced by name;
    if target file does not already exist{
        write header stuff for top of file such as agent name, etc.
    }
    if(EXCITE) write all lines from EXCITE_OUTPUT;
    else if(YAHOO) write all lines from YAHOO_OUTPUT;
    else write all lines from regular output;
    close file;

} //writeHTMLfile

public void bomb_out{
    if the threads array is non-null{
        for each thread in array stop it or nullify it;
    }
    send "done" message to parent;
    stop the parents flashing icon;

} //bomb_out

} //WebGrep

//-----

Class WebGrepThread extends Thread(pattern, spider, parent){

    //this class searches one html file and then exits.
    //Separate thread so it can be killed whenever we wish.
    Run(){
        Start a new TimeKiller to watch and kill this thread;
    }
}

```

```

        Get the next web page name from Spider;
        Make connection to web page;
        If(connection no good) kill thread; break;
        For(;;){
            Read a line from web page;
            If line is null break;
            If line contains pattern store page name and break;
        }//for

    }//class WebGrepThread

//-----

A Spider is class that generates the index of links on a page of HTML. It
only considers links that start with "http://". It returns a list of links
found to the caller. Broken links are returned as errors.

Class Spider(baseURL){
}

search(){
    read HTML file and return a list of URLs in that file;
}
//-----

Timekiller can watch a thread for a timeout in milliseconds. If this timeout
is reached the thread is killed with stop(). Otherwise it may be killed
manually.

Class TimeKiller implements Runnable(Thread, millis){

    WatcherThread = new Thread (this);
    WatcherThread.start();

TimeKiller(millis){
    This(currentThread, millis);
}

public void synchronized run(){
    if(thread enabled){
        do{
            loop=false;
            wait(millis);
        }while(enabled=true and loop=true);
    }
    if(enabled and targetThread.isalive()) targetThread.stop();
}
}

```

Appendix B

Publications based on this Thesis

1. "Visual Modeling for Agent-based Applications", B.Falchuk and A.Karmouch, *IEEE Computer*, Vol.31, No.12, December, 1998
2. "The Mobile Agent Paradigm Meets Digital Document Technology – Designing for Autonomous Media Collection", B.Falchuk and A.Karmouch, *Multimedia Tools and Applications*, Kluwer Academic Publishers, Vol.8, No.1, January 1999
3. "Saving Time and Reducing Network Traffic with a Graphical Mobile Agent Inception System", B.Falchuk and A.Karmouch, *Proceedings of SBT/IEEE Int'l Telecommunications Symposium '98*, Sao Paulo, August 1998
4. "A Mobile Agent Prototype for Autonomous Multimedia Information Access, Interaction and Retrieval", B.Falchuk and A.Karmouch, *Proceedings of Multimedia Modeling '97*, pp.33-48, Singapore, November 1997
5. "AgentSys: A Mobile Agent System for Digital Media Access and Interaction on an Internet", B.Falchuk and A.Karmouch, *Proceedings of IEEE Globecom97*, pp.1876-1880, Phoenix, November 1997
6. "Agentizing the Internet - The Practical Design of a Networked Mobile Agent System", B.Falchuk and A.Karmouch, *Proceedings of AACE WebNet'97 - World Conference of the WWW, Internet & Intranet*, Toronto, October 1997