Spring 2018

Agent-based Computing in Java

Michael SYMONDS
San Jose State University

SAN JOSE STATE UNIVERSITY

MASTERS THESIS

Agent-based Computing in Java

Author:
Michael SYMONDS

Supervisor:
Dr. Jon PEARCE

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Computer Science

April 30, 2018

SAN JOSE STATE UNIVERSITY

Abstract

Department of Computer Science

Master of Science

Agent-based Computing in Java

by Michael SYMONDS

Agents are powerful, autonomous entities capable of performing simple, or vastly complex, operations individually or in groups of agent systems. Their capabilities extend significantly as mobile agents distributed across a network. Agent-based computing is a widely used technology with a broad range of applications, particularly in distributed computing and agent-based modeling. Many types of systems can be designed using the different architectures that define how they act, communicate, migrate, and more. This paper surveys agent-based computing, their architectures, and efforts at the standardization of certain aspects of the technology. It explores an existing framework called Jade through the lens of a demonstration based on the Sugarscape model, implemented using Jade's library. Finally, it presents a new framework, called NOMAD, a simple barebones framework which comprises the most essential components needed for a mobile agent framework. With it, a user can quickly and more deeply understand the vital challenges agent systems must address, such as communication and code mobility, and the solutions needed to be implemented. They'll be able to use the framework to extend its capabilities, create new components, and build powerful agent systems of their own.

Contents

Chapter 1

An Abstract Model of Agent-Based Computing

In object-oriented programming, agent-based computing is a well-developed and broadly used technology. The combination of autonomy that agents provide and mobility in the form of a mobile agents makes a powerful tool to use in several domains of application, including the modeling of highly complex simulations, distributed computing systems, software-as-a-service, and more.

Depending on the challenge being addressed, the architecture that the agent-based system might implement can take significantly different forms. Considering the agent itself, the environment it runs in, the services needed, and how groups of agents should interact, several questions must be addressed before an appropriate framework can be developed and used. From decades of development, several existing frameworks are available, some even as open source toolkits and libraries, to help the developer get a head start on the foundation needed to provide the right environment for the agents to perform their tasks. With a better understanding of agent-based computing, and the key components vital to those systems which many existing frameworks share, developers gain an advantage with creating a well-designed agent-based system robust enough to perform any needed tasks and lightweight enough to avoid overhead from unnecessary components. This paper will survey agent-based computing and mobile-agent systems, examine existing implementations of such systems, and introduce a new basic mobile framework, called NOMAD. The main purpose of NOMAD is to describe the essential mechanisms needed to implement a mobile agent system in the Java language.

The next few sections will describe the concepts of agents and agent systems, examples of applications of agent-based systems, the different strategies agents can take to solve problems, discuss the effort to standardize aspects of agent systems, and finally introduce a few examples of implemented agent frameworks to illustrate how different strategies, models, and standards can be combined into agent systems.

Chapter 2 will focus more deeply on the Jade (Bellifemine, Caire, and Greenwood, 2007) agent framework to see how its components function, with a more technical focus, through the lens of a demo implemented using the framework itself. The last chapter will focus on building a simple mobile agent framework from scratch, describing the essential components required and discuss some of the problems to be addressed when designing such a system. By the end, the reader should have a good sense of the concepts, technology, standards, and (In the case of the Java language) the implementation that goes into modern agent-based systems.

1.1 Defining Agents

The world of agent-based systems begins with the agent itself. In the simplest form, an agent is an entity which acts on behalf of the user. Like objects, agents possess both a state, through its data, and behavior through methods. But unlike objects, they are autonomous and goal-driven. Agents can have an understanding about their environment. Though they may possess a set of deterministic behaviors, they can select from among these behaviors in a dynamic order based on their goals and what they know about their environment. This can allow them to act in non-deterministic ways. The key qualities that separate an agent from a standard object are: *autonomy*, *reactiveness*, *pro-activeness*, and *social ability*. Though not a strict requirement, yet given the prevalence among agent systems, it could be argued that *mobility* should also be included as a key quality. Agents are typically described in terms of their internal behaviors and their external interactions with the environment (Agents, 2018) (and in the case of a multi-agent system, the interaction with other agents as well). Agents reside within a location usually referred to as a platform or container. The platform provides the services that facilitate discovery, communication, migration and more. Multiple platforms of agents can exist, whether in the same process, on different processes within the same hardware, or fully distributed across a network. Agents can be social and operate independently or collectively within their platform or over the network environment.

The *autonomy* that an agent possesses is typically enabled by having its own thread of execution in which it operates. It can detect aspects of its environment or more fully model it through representations defined through mechanisms of perception, event handling, or from messages received from other agents or entities. With this information, it can respond in a *reactive* fashion, or formulate and refine its strategy to control its environment through *pro-active* reasoning. Through communication and negotiation, agents can utilize their *social ability* to share information about their environment, services they perform, commands to follow, or coordinate other tasks to collectively achieve goals.

Agents can migrate to other platforms and environments, even over a network. These so-called mobile agents can suspend their operation, migrate to another location with its own code and attributes, and resume operation while keeping its state. The idea of an agent retaining its state is the subject of some interest. Agents are described as having weak or strong mobility. Agents with weak mobility can take their code and attributes, but do not resume their action at the same line of instruction where they were suspended. Typically, these agents are activated and resume operation by starting at the very beginning of their behavior cycle. Agents with strong mobility, by contrast, can retain their state in a way where mechanisms can also capture the agent's execution state, which includes the program counter and the stack where it was executing (Cabri, Leonardi, and Zambonelli, 2018). A hybrid version of this, implemented in the Jade demo discussed in Chapter 2, uses a state machine model of agent behavior to allow post-migration resumption of activity at a checkpoint in its behavior cycle. With this and the above traits all taken together, agent-based systems can be constructed in myriad ways to serve a vast array of services and applications.

1.2 Applications of Agent-based Systems

Many types of applications take advantage of the unique benefits that agents offer. For more than two decades agent systems have been developed and used for increasingly complex real-world applications. Most applications of agent-based computing fall into two broad categories: distributed computing and agent-based modeling. Many applications can even span both categories, such as agent-based supply chain optimization applications that model resource distribution strategies in the cloud. Below is a small, but by no means exhaustive, set of examples.

1.2.1 Distributed Computing

Multi-agent systems are a natural fit for distributed computing. Agents can encapsulate highly complex processes, operate autonomously, and migrate or even clone themselves throughout the network by command or at will. Applications of distributed computing span a number of areas such as the sciences (Korpela et al., 2001) (Hebert, 2015), business and e-commerce (Lange and Oshima, 1998) (Karabey and Adar, 2014), control systems (Aguilar et al., 2001), robotics and autonomous systems (Perugini et al., 2003), Healthcare (Moreno and Garbay, 2003), security, network analytics (Voorde, 2016), military and defense, the Internet of Things (Yu, Shen, and Leung, 2013), web services (Booth et al., 2004), even the web itself was at one point in consideration to be remade as an agent-based system (Berners-Lee et al., 2004).

One of the key advantages agents bring to a distributed system is mobility. Imagine a very large database located remotely where complex calculations need to be made on its data. Trying to pull that data through the network is both a burden to the network and a significant security risk of exposure is the data is sensitive. Instead, agents can be sent to the data, perform the necessary tasks, and return with the result in a much more efficient way that minimizes exposure over the network. With their autonomy, agents can redistribute themselves to different parts of the network to balance out processing resources. Or they can duplicate themselves if their task is suited to parallelization.

1.2.2 Agent-based Modeling

The other area where agent-based systems are most commonly used is agent-based modeling (ABM). Several branches of science, from neuroscience and physics to economics, sociology, socio-economics and more have used ABM to great effect in measuring the response and behavior of autonomous agents within a given environment. ABM developed significantly in social sciences with the appearance of early agent-based programming languages like NetLogo in the 1990s. One of the first large scale models of note developed at that time was called Sugarscape (Epstein and Axtell, 1996). This model simulated and explored various social phenomena like seasonal migration and disease transmission, which is discussed in greater detail in chapter 2.

Other types of models examine optimal communication and team effectiveness, various aspects of social networks and culture, effects of economic policy, even models of human cognition. The wide use of such models has led to the development of several frameworks designed specifically to address both the potentially high degree of complexity of simulations and the challenge they pose to verification and validation of the results.

1.3 Architectures of Agents and Agent Systems

Depending on the task to be solved, the strategy used by the agent and the design of its location environment can influence how well the agents can perform the tasks at hand. The architecture used to create agents embedded in autonomous vehicles to navigate their surroundings may not be as effective when applied to detect objects in images, regulate the load of a power distribution system, or determine trends in the stock market. Multi-agent systems with a high degree of inter-agent cooperation and teamwork may require robust communication and synchronization services from their platform environment. Architectures defining how knowledge is represented and expressed may also play an important role with inter-agent communication and cooperation. Security strategies that demand complex considerations may play a role for vital components of the task. Different configurations of services for different tasks may be required, demanding strict systems of modularity to add and remove services, even dynamically.

Several different types of agent frameworks and environments have been developed over the years to address as many of these different scenarios as possible. Most of them can be sorted into a few abstract categories of architectures. These are *Reactive*, *Deliberative* (or *Pro-Active*), *Cognitive* and *Hybrid* types of architectures. When considering an agent architecture, the question to answer is: how does the agent need to interact with its environment or other agents? Each of these categories addresses distinct strategies of action and representation of the agent's environment. The sections below describe each type of architecture and give examples of what a specific architecture of that type might look like. From there, a more macro-level perspective is taken to look at the *Distributed* architecture, which describes the components of multi-agent systems and the elements needed for coordinating agent interaction and services are discussed.

1.3.1 Reactive Agents

Consider a robot given the task to move from one location to another while avoiding obstacles. The robot needs to have some model of its environment to make a plan of action. In many cases though, creating the symbolic representations that make up the elements of the environment, determining a plan of action based on that knowledge, and then executing that plan, is a highly complex process. The resources and time required to determine a course of action might cause the execution of that action to happen too late. The robot may then become paralyzed with inaction if the environment always changes faster than it can understand it.

One view is that an agent does not actually need any symbolic representation of its environment to interpret. Instead, by using a set of stimulus-response rules, the agent has a more direct connection to its environment and can react to it in a timely fashion. Agents whose architecture follows this paradigm are said to have a *reactive* architecture. One example of this is the *Subsumption* architecture (Brooks, 1986). This reactive architecture was developed as a hierarchical, layered system of task-achieving behaviors. The system must follow three basic requirements: agents should be able to cope with multiple goals, have multiple sensors, and be robust.

On what layer a task-achieving behavior might be in the hierarchy determines how specific that task may be. Though layers work in parallel, higher layers can have access to data in the layers below and are able to change that data intermittently, thus influencing the behavior of those layers. Though this architecture was influential for others that came later, it is generally limited in how

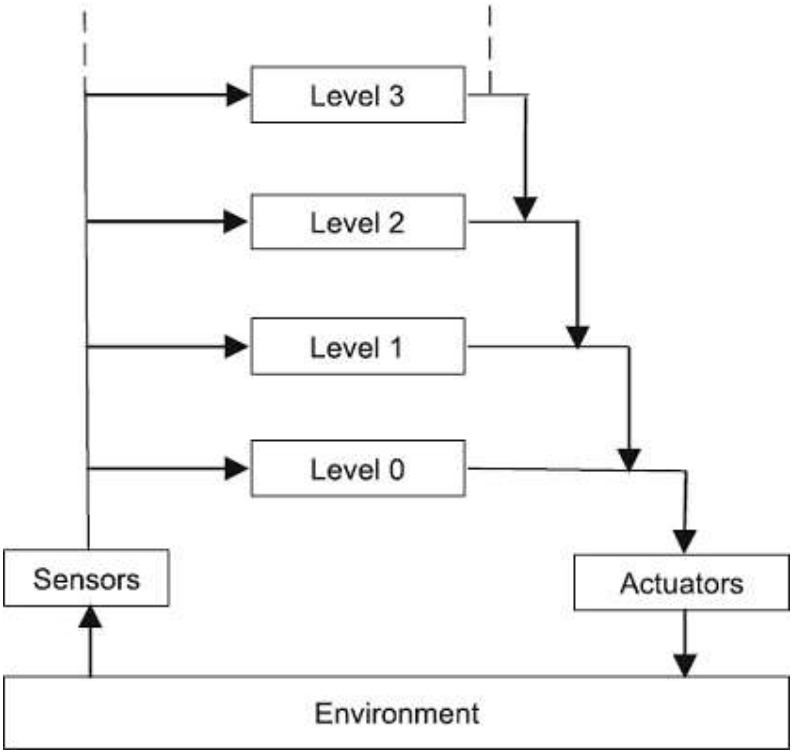

FIGURE 1.1: The Subsumption Architecture (Chong, Tan, and Ng, 2007)

the system can scale in complexity and its ability for explicit reasoning.

1.3.2 Deliberative agents

An agent that can hold a representative model of its environment, has a set of de- sires or goals, and a plan for how to achieve those goals, is said to operate using a deliberative architecture. Here the agent creates a set of possible actions based on interpretation of the environment model and must *deliberate* to choose which action to take. This includes potentially accepting a schedule of behaviors to take in an or- der that may need to be adjusted or abandoned as its belief about the environment it models changes.

There are many architectures that fall under this category, and most of them are based on a model called the *Belief-Desire-Intention* architecture, or BDI. Beliefs are the agent's informational awareness of the environment. Desires are some subset of the agent's overall set of goals. Intentions are the subset of available desires the agent commits resources to achieving. Unlike in the reactive architecture, the agent here has an explicit representation of all three of these concepts within its memory during runtime.

One of the most successful control systems built for practical reasoning based on BDI is the *Procedural Reasoning System*, or PRS. This is a framework for building reasoning systems to perform tasks in an environment. The basic concept of the framework is that the system has an interpreter which holds beliefs about the

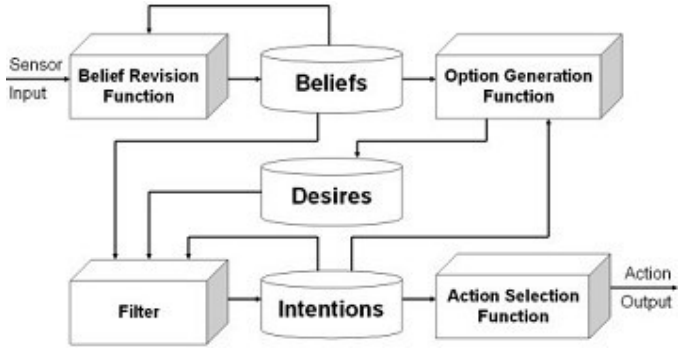

FIGURE 1.2: The BDI Architecture (I., 2014)

agent's environment, deciding which goals it should attempt to achieve, and which knowledge areas to apply towards achieving its selected goals. Knowledge areas are pieces of procedural knowledge which inform the agent how to perform some specific task, like determine a path through the environment or translate through that environment.

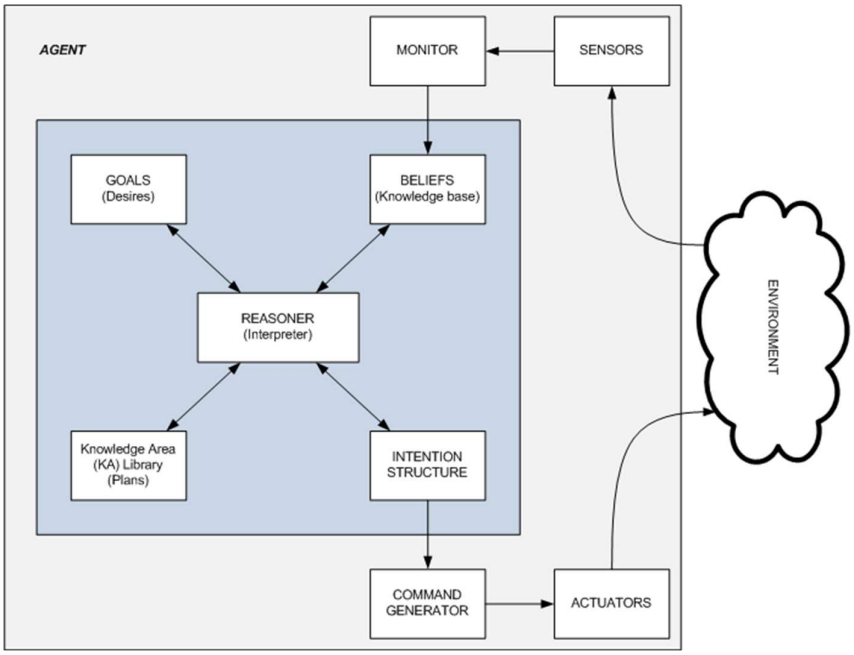

FIGURE 1.3: The PRS Architecture

1.3.3 Cognitive Architectures

BDI and PRS concern themselves with the decision process to determine actions based on the relationship between the goals an agent has, and the model generated from the input received about its environment. Cognitive architectures instead concern themselves with how the knowledge that symbolizes the environment is

represented and communicated both within the agent and between it and its environment, including other agents. Cognitive architectures embody computational structures underlying general intelligence.

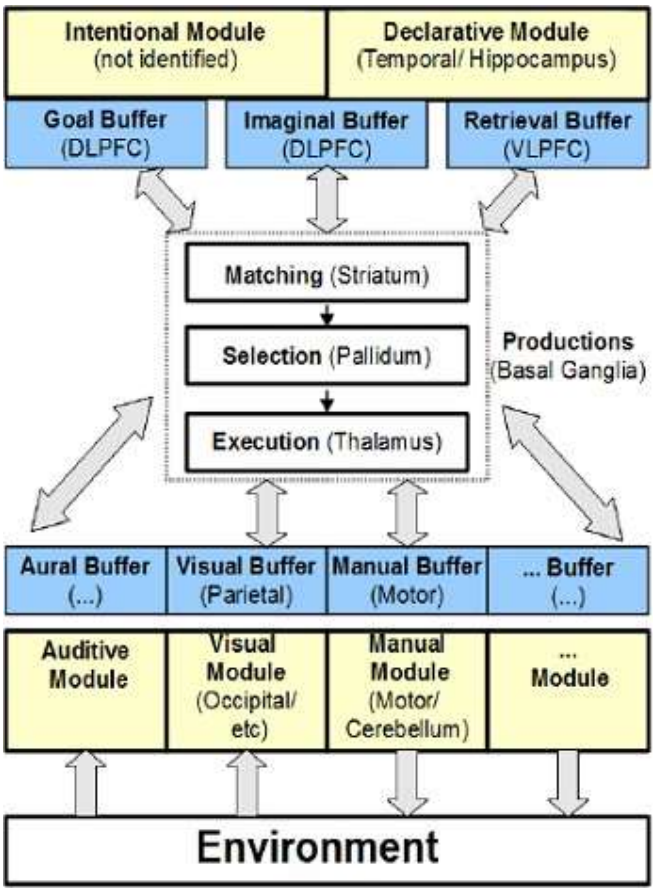

FIGURE 1.4: The ACT-R Cognitive Architecture (Haring, Ragni, and Konieczny, 2012)

But not just any structure will do. Agents that implement this architecture are primarily concerned with formalizing theories about the structure of the human mind and modeling those formalizations. Such theories try to explain how organisms (or agents) can detect and respond to their environment, how goals are acquired, how behavior related to those goals is executed, and how knowledge is learned and represented (Newell, 1990). A few examples of this are the Soar project, ACT-R (Adaptive Control of Thought—Rational), Brahms, and LIDA (Learning Intelligent Distribution Agent). Decision making for these examples incorporate several theories of cognition, though the emphasis for each may differ from a focus on AI to that of cognitive modeling.

1.3.4 Hybrid Architectures

Hybrid architectures, predictably, combine both deliberative and reactive (and potentially cognitive as well) architectures together into one agent. In this way, an agent can both reason about their environment and take the best action to achieve

its goals and still react efficiently to changes in that environment. A number of architectures designed to incorporate these two concepts, like TouringMachines (Ferguson, 1992) and InterRRaP (Muller and Pischel, 1993), consist of layers, similar to the subsumption architecture.

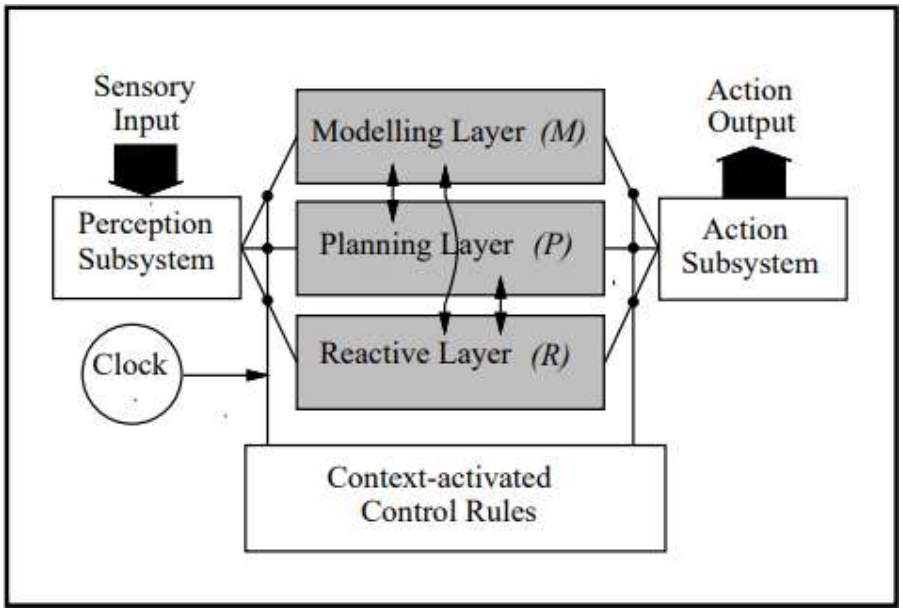

FIGURE 1.5: The TouringMachine Hybrid Architecture (Ferguson, 1992)

In the case of TouringMachines, the layers consist of the planning, modeling, and reactive layers. These layers run in parallel, are independently connected to the perception system with sensory input, model the environment at different levels of abstraction, and compete for control of the agent's actions. A set of global context-dependent controllers handle conflicts and final decisions the agent makes. In the case of InterRRaP, the layers are vertical. Communication is limited only to adjacent layers. Decision making is decentralized, where control spreads upwards through the layers until the most valid layer is reached and action is executed.

Another example of the hybrid architecture is the Brahms system (Sierhuis, Clancey, and Hoof, 1999), developed in 1998 by NASA Ames research center. Brahms incorporates both hybrid and cognitive architectures. With it, NASA developed several systems and applications including an Orbital Communications Adaptor Management System, an advanced multi-agent EVA communications system, and a Metabolic Rate Advisor personal assistant for astronauts to name a few.

1.3.5 Distributed Architecture

The architectures described above are from the perspective of a single agent. But when considering a collection of agents, either all in one platform location or distributed among several, a shift is needed to a higher perspective to describe architectures that include both the agents and the platforms they reside in and migrate be- tween. Though they are called architectures in this section, they can also be thought of as systems or protocols. These *distributed* architectures should describe systems that manage multiple agents and components needed to facilitate multiagent services, such as communication protocols, directory services, and migration.

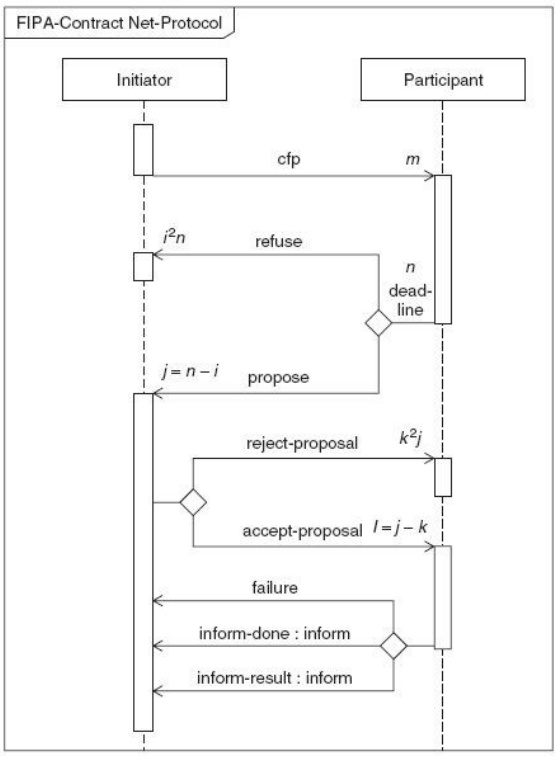

FIGURE 1.6: The Contract Net Protocol (Agents, 2018)

Different architectures may address different components. For example, one of the most widely used architectures in multiagent systems focuses on managing the collective behaviors of distributed agents. This is the *Contract Net Protocol* (CNP). In this system, an agent will decompose a given task into smaller subtasks. It will then broadcast task announcements with eligibility requirements out to other agents. Agents who can take on new tasks will go through an evaluation process to determine if their eligibility satisfies the requirements and respond with a bid if so. Bids are evaluated, and the contract is awarded to the winner. Those agents now have a manager-contractor relationship. As completed tasks come in, managers monitor the partial success of their tasks as they work towards a completed solution. Contractors who take on tasks can themselves follow this process, further breaking down tasks into simpler subtasks, and so forth.

CNP is limited, however, assuming the nature of tasks being resolved can be decomposed. Also, the communication structures do not scale to more complex interactions such as negotiation between agents. Other multiagent infrastructures, like RETSINA, focus on services that enable complex social interactions between agents which do allow for such features. In this architecture, RETSINA is less concerned with the architecture of the individual agent and more with the ability of the agent to communicate with other agents and with RETSINA's components to give the agent a sense of social awareness. This gives the developer flexibility in choosing coordination schemes and to define what the "social norms" are for the agent society. Other architectures such as DECAF, Teamcore, and Agentis likewise describe different structures of agent interaction models, each having varying focus on different features (Luck, Ashri, and D'Inverno, 2004). These include generic and reusable behaviors, principals of coordination which can provide certain guarantees of system

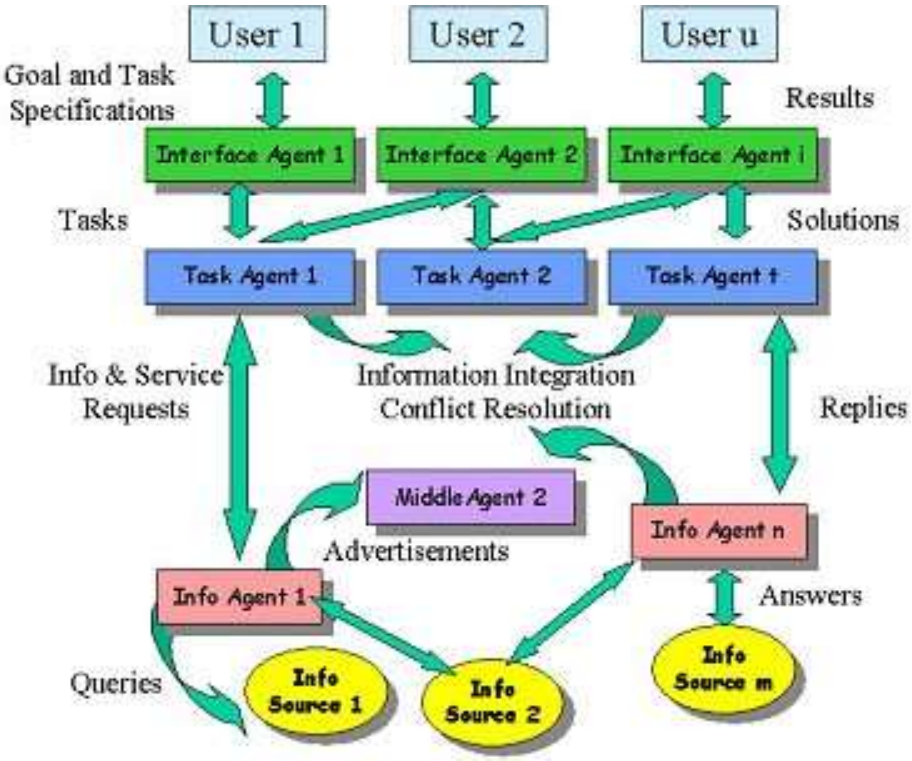

FIGURE 1.7: Retsina MAS (Sycara, 2012)

behavior, or employing standardized communication protocols to enable interactive applications, respectively.

It would be a long discussion indeed to examine the architectures for the various components of the multiagent system and choose which one is appropriate for a system to be designed. Because of this, many have pushed for standardization of the distributed architectures of multiagent systems. This would enable separately designed, discrete multiagent systems to have some degree of interoperability. Standardization would also make them potentially more accessible to inexperienced users learning how to use them. The next section will discuss the efforts to standardize aspects of multiagent systems.

1.4 Agent System Standardization

The development of agent-based computing evolved over the same period as the World Wide Web. One of the visions for the agent model was to utilize its features in the context of software-as-a-service. A client can request applications and services from a server which can be delivered upon request via agents. In this sense, a client can be not only an end user, but other applications as well. With the variety of architectures in development to address the wide array of strategies to employ multi agent systems, it was quickly realized that a set of standards would be greatly beneficial to help ensure the interoperability of agent systems. To date, there are, ironically, several standards and pseudo-standard technologies that have been developed and currently in use. Some, like the Knowledge Query and Manipulation Language (KQML) and the Foundation for Intelligent Physical Agents (FIPA) standards groups, focus on communication standards. Others like the Object Management Group Mobile Agent System Interoperability Facility (OMG MASIF) focus on mobility; and still others like Knowledge Interchange Format (KIF) and

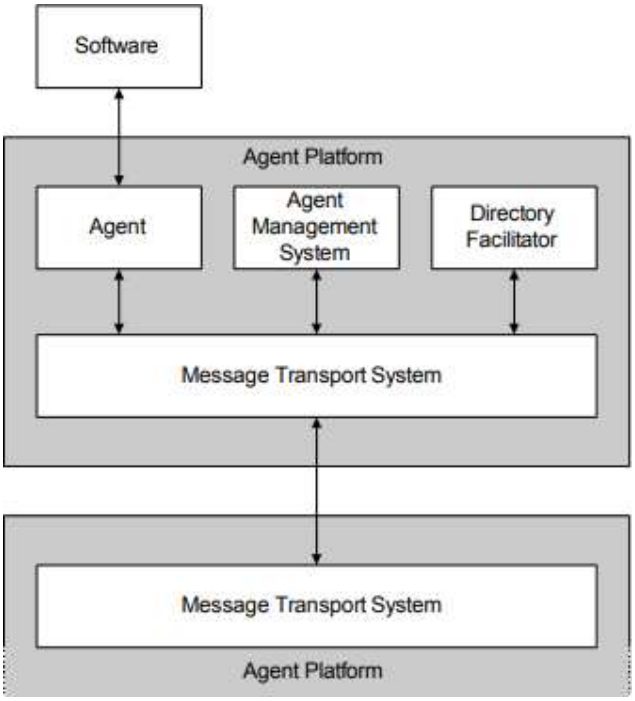

FIGURE 1.8: The FIPA Agent Management Reference Model

The Semantic Web focus on other enabling technologies like knowledge expression and ontologies. Of these standards, one of the most widely adopted is the FIPA standard, which is what this discussion will focus on, as communication is one of the key requirements when considering the interoperability of discrete software systems.

FIPA is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies. The multiagent system (MAS) envisioned by FIPA stems from a philosophy that agents should be supported by a generic distributed computer infrastructure, or set of middleware services (Agents, 2018). Agents that utilize the services of communication, data storage and retrieval, migration, directory services, and so forth should receive those services from entities within the platform they reside. For some services, such as communication, FIPA stresses that these should not be agent-based for reasons of efficiency. When passing a message to another agent, it would first need to pass the messaging agent a message before it then forwards the message to the intended recipient. Better that the service is implemented as a non-agent entity. So, the Message Transport System, or MTS is specified to facilitate that service. The Agent Management System, however, can be agent-based. This service oversees the agent lifecycle, registering and deregistering the agent from the local platform, and managing the agent directory, or white pages. Similarly, the Directory Facilitator can also be agent-based. This agent provides the directory of services, or yellow pages, allowing agents to find others by the services they provide.

Agent communication represents the heart of the FIPA multiagent system model. It standardizes an extensible library of "communicative acts" that allow representation of different intentions (ex. requesting, proposing, informing, querying, soliciting proposals, refusing, etc…). It also defines the structure of the message that is

sent between agents, including its properties (the encodings and the representation language) and information useful to identify and follow threads of conversation between agents, and to represent timeouts for the communication. It also defines interaction protocols which provide agents with a library of patterns to achieve common tasks such as delegating an action, calling for a proposal, etc...

In all, FIPA has completed several specifications that span the categories of agent communication, agent transport, agent management, abstract architecture and applications. But currently only a subset of 25 of those specifications have made it to the standardization stage. In the sections below, an overview is given of 3 of the specifications (the Agent Management, Message Structure, and Message Transport specifications) commonly used by multi-agent implementations which conform to the FIPA standard. You will see these standards implemented in Chapter 2 as the Jade Framework is explored. Keep in mind that implemented frameworks are not bound to conform to all FIPA (or any) standards and may do so to only a subset or even just one.

1.4.1 FIPA Agent Management Specification

The Agent Management Service specification is described in terms of logical components, each of which possess a set of capabilities. The details of how these components are implemented is left to the developer, but the capabilities of each are well defined. It describes an agent system contained within an *Agent Platform* (AP). This AP can be as small as a single-process platform with lightweight threaded agents to a fully distributed platform built on middleware standards.

A required component of the AP is the *Agent Management Service* (AMS). The AMS is a pre-defined agent which controls access to, and use of, the AP. No more than one can exist in a single AP. The AMS manages a directory of all Agents registered to it. Agents must register with an *Agent Identity* (AID), a label which distinguishes it from all others. The AID must carry certain re- quired parameters such as the agent's globally unique name, its transport address, and resolvers (a list of name resolution service addresses). The AMS must also provide directory service for agents to locate each other which is called the *white pages service*.

Optionally, the AP can also have a *Directory Facilitator* (DF). Unlike the AMS, multiple DFs may exist within an AP. If an agent provides a service, that service can be registered with the DF. Other agents may then query the DF for to find out the services being offered by the agents who registered. This is called the *yellow pages service*. The *Message Transport System* is the default service for communication between agents on different APs.

Finally, there is the *Agent* component itself. As stated above, the agent must have a globally unique name, with its identity encapsulated in the AID. It must be registered to the AP and list a transport address at which it can be contacted. It must communicate using an *Agent Communication Language* (ACL). An agent has a physical life cycle which is managed by the AP.

1.4.2 FIPA Message Structure and Transport Specification

It's assumed that every Agent Communication Language (ACL) message in an agent system contains at least *sender*, *receiver*, and *content* parameters. The FIPA ACL Message Structure Specification simply requires that the message also contains a *performative* parameter as well. The performative defines a type of communicative act.

FIPA lists a total of 22 such performatives in the Communicative Act Library Specification, encompassing every conceivable communicative scenario between agents. These are acts such as *inform*, *request*, *propose*, *accept proposal*, *query if*, *subscribe*, and so forth.

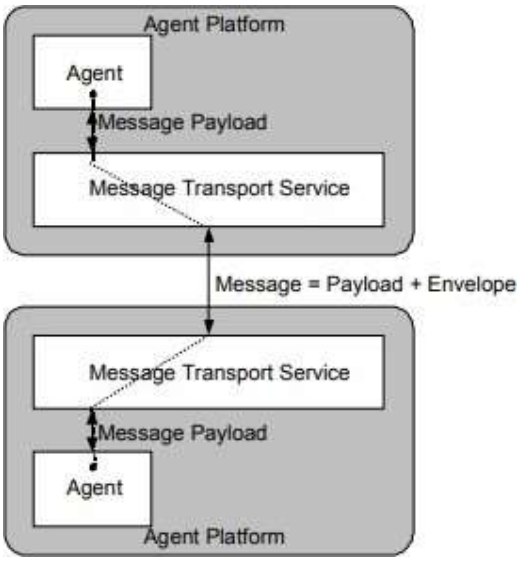

FIGURE 1.9: The Message Transport Reference Model

The Transport Specification describes the ACL message structure as a combination of the payload, which holds the ACL message of the agent communication, and the envelope which holds transport information only. It also specifies an Agent Communication Channel entity which carries out message transport tasks directly to the agents on AP. It also has access to the AMS and DF. We can thing of the ACC as part of the MTS. Once a message is received by the ACC from an agent, it delivers the message to the remote ACC at the destination(s) specified in the envelope. The destination ACC then delivers the message to the local agent whose AID is listed as the intended recipient.

1.5 Examples of Implemented Agent Architectures

The number of architectures, standards, protocols, and other technologies have given rise to several implementations which have been used to solve a very wide range of needs. There are far too many to cover here but a few will illustrate how different combinations of models lead to unique and interesting frameworks. Some brief examples are JACK, which uses a BDI model with FIPA ACL standards, Brahms which combines BDI and Cognitive architectures, but does not incorporate any official standard, or BOND, a FIPA-compliant mutliagent system where agents can be created statically using the toolkit's API or dynamically during runtime using BOND's blueprint language. The choice of toolkit to use for a developer lies within what purpose the agent system will be used for, how much that system will scale (even across network platforms), the interoperability needed between software systems or agents, and so forth.

Below three frameworks are introduced. The first, NetLogo, is an agent-based programming language designed to be widely accessible to those who are new to agent-based computing, even children. The other two, Aglets and Jade, are multi-agent framework implementations which illustrate the Reactive and Deliberate architectures respectively.

1.5.1 NetLogo

NetLogo (Tisue and Wilensky, 2004) is a multiagent programming language with an integrated modeling environment. It's designed to model complex systems at a level suitable for scientific research, but with an environment accessible enough for even k-12 students to quickly grasp its language and interface to start experimenting right away.

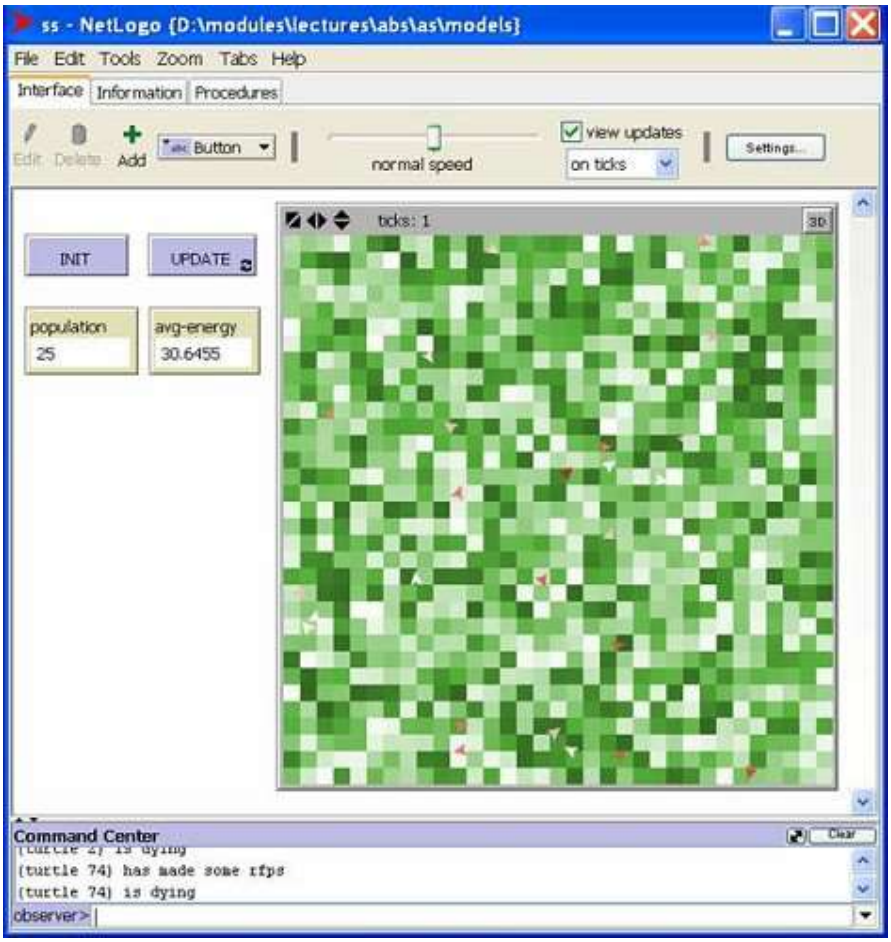

FIGURE 1.10: Screenshot of Sugarscape in NetLogo (Pearce, 2018)

NetLogo originated as a combination of StarLisp and Logo, then was re-written entirely in Java, and now is mostly written in Scala with some parts still in Java. Its agent model consists of *turtles* (Agents) and *patches* (the Environment). Patches are typically aligned in some n x m grid and populated with symbolic turtles that can traverse the grid. Both turtles and patches can be given actions using the coding language while attributes and parameters of the experiment being created

can be adjusted dynamically using buttons and sliders automatically generated in the provided GUI.

```
to reproduce
  ask turtles [
    if energy > 50 [
      set energy energy - 50
      hatch 1 [ set energy 50 ]
    ]
  ]
end

to check-death
  ask turtles [
    if energy <= 0 [ die ]
  ]
end

to regrow-grass
  ask patches [
    if random 100 < 3 [ set pcolor green ]
  ]
end
```

FIGURE 1.11: Sample code from the NetLogo language

Since its inception in 1999, NetLogo has amassed a model library with over 150 pre-built simulations. These simulations cover several fields including Biology, medicine, physics, chemistry, economics, social sciences, computer science, mathematics, and more. There are also several existing extensions such as having two NetLogo systems connect to each other from different systems via peer-to-peer, one which adds BDI and FIPA ACL message passing, even one which enables the Cognitive Architecture to turtle agents.

1.5.2 Aglets

The Aglets framework (Lange and Oshima, 1998) is a mobile agent toolkit written entirely in Java that implements the reactive architecture. Aglet agents are built with the intention of being a generalization and extension of Java Applets. Aglets are lightweight, mobile, and follow an event-driven model of execution. They are designed to work well in web server environments, delivering content and resources on demand.

Four key abstractions make up the model underlying the API: the *aglet* (agent), its *proxy*, a *context*, and the *identifier*. The *aglet* agent is autonomous and can be created, disposed, cloned, dispatched (moved), retracted, and de/activated as core functions. The *proxy* is the aglet's representative. As a layer of protection, the proxy acts as a go-between to handle all communication and access to public methods on its behalf. It can also hide the aglet's real location, adding another layer of security. The *context* is the location where aglets operate locally. It can be thought of as a container which holds local agents and provides services and execution

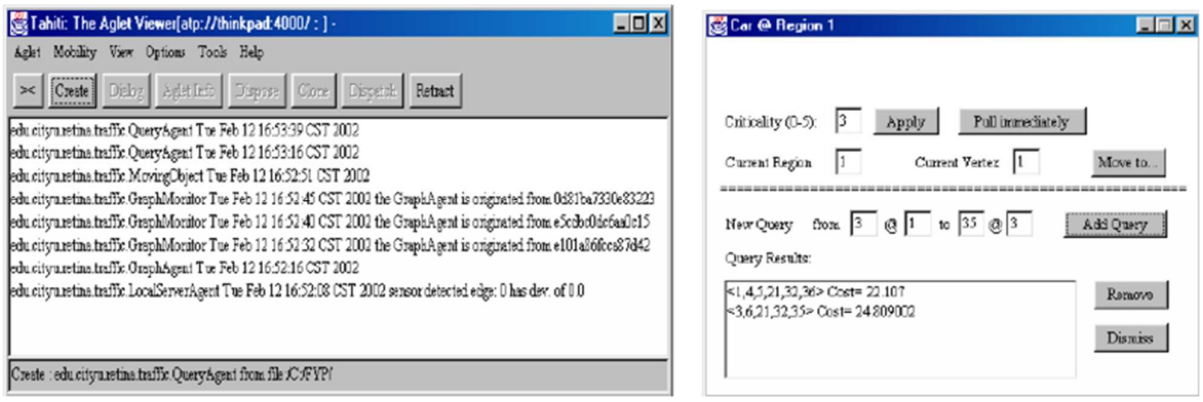

FIGURE 1.12: Tahiti interface in Aglets (Lam, Kwan, and Ramamritham, 2002)

management. Several contexts can be hosted within one server process, with multiple server processes able to be run on a platform, or node, in a network. A single, globally unique, immutable *Identifier* is bound to each aglet in a context.

As mentioned above, Aglets has an event-based programming model which is implemented through event-listeners. The developer has three types of listeners available to catch certain events and prompt responsive action. A *Clone listener* al- lows an aglet to take action just before, after, or when a clone is actually created. A *Mobility Listener* allows an aglet to act just before an aglet is either dispatched or retracted, or just after the aglet arrives at its destination context. A *persistence listener* allows it to take action just before its deactivated to allow it to save the state or any other serializable data. Any additional action the aglet may perform is defined in the overridden run() method which is called after the aglet is activated in a context.

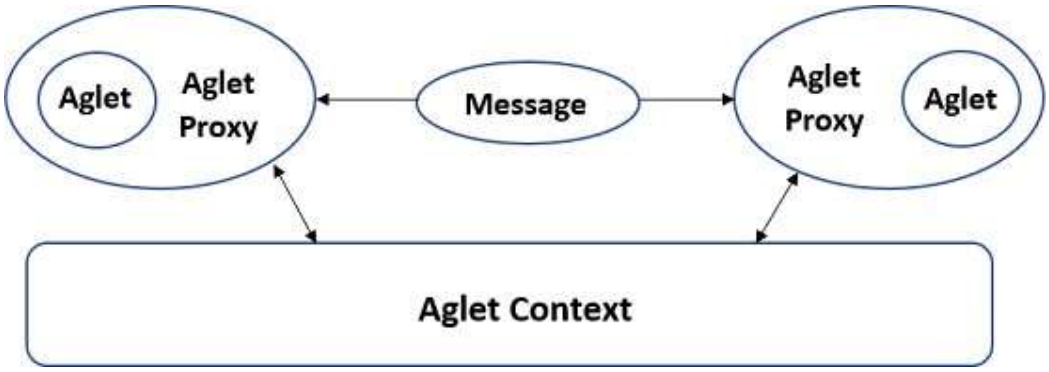

FIGURE 1.13: Aglet Message Passing and Access through Agent Proxy

Communication is facilitated through message passing. An aglet will get a handle on another aglet's proxy using the context resource. The messages passed are objects that allow for synchronous or asynchronous passing between agents. The agent's proxy receives and handles the message first through a handleMessage(Message msg) method the developer would override. The agent can choose a future reply which creates a response that the recipient can handle asynchronously.

An aglet can be dispatched to, or recalled from, a new context. This is accomplished by first deactivating the aglet. It's then serialized, transmitted through the network, deserialized, and then activated at the new context. Activation calls the run() method which starts the aglet from the beginning, so the only opportunity to save any kind of state before transfer falls within the onDispatching(MobilityEvent ev) method which is invoked before deactivation. Aglets can migrate themselves or be moved by request from their local context. They also have the ability to cancel requests for migration by throwing an exception during the onDispatching(MobilityEvent ev) method.

1.5.3 Jade

Like Aglets, Jade (Bellifemine, Caire, and Greenwood, 2007) is written in Java, but instead uses the concept of behaviors for programming agents. Behaviors can also take the form of registered services that other agents can seek out and request, provided they possess the relevant ontology. The other major difference from Aglets is that Jade conforms to several FIPA standards, particularly the agent management, communication language, and ACL Message specifications.

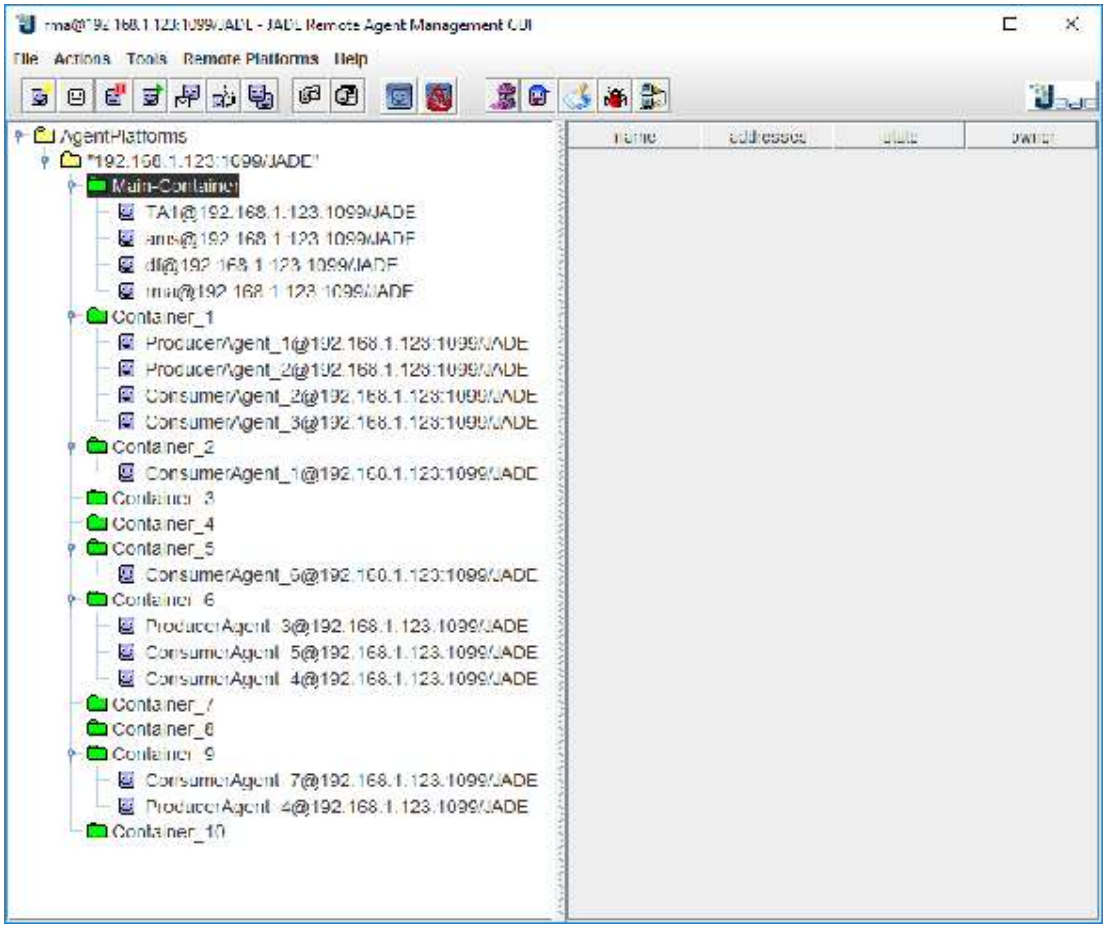

FIGURE 1.14: The Jade Interface

Jade can operate in both fixed and mobile environments. It incorporates a technology called *LEAP*, which allows it to split its operations into a lightweight front end which can run on mobile devices and a remote backend. Communication happens with asynchronous message passing. Agents can be dynamically discovered using the white or yellow page services, even targeted in groups by specifying a property (ex. all agents interested in news updates) as the message destination. Jade provides security features with mechanisms to authenticate and verify rights assigned to agents, and other extensible features.

Jade supports agent mobility, along with the execution state of an agent. It also supports code migration for destination hosts which do not know about the agent. This system works dynamically through Java's Class Loader to maintain network efficiency.

As a middleware, Jade provides both the environment to maintain a multiagent system, and an API library easy to use for creating your own agents, ontologies, concepts, and behaviors. Premade agent examples are also provided, along with several premade FIPA-compliant interaction templates.

In the next chapter, a deeper look is taken at the Jade framework for a better understanding in how the architectures and standards are implemented by the major components. These will be examined through the lens of a Sugarscape concept demo implemented using custom agents built with the Jade library.

Chapter 2

Sugarscape In The Jade Framework

2.1 A More In-Depth Overview of the Jade Framework

As mentioned in the previous chapter, Jade conforms to the FIPA Agent Management Specification. The main environment Jade provides is a Platform, which is created in the JVM, and holds at least one container it calls the main container. Multiple containers may exist within a platform, which is fully connected, and containers may exist on different physical systems in the network. You may also link platforms together, which allows for inter-platform communication between agents.

The main container keeps two required agents which provide all platform services for the agents. The first is the Agent Management Service, or AMS agent. This agent is responsible for managing the agent lifecycle and agent directory, or white page services. The other is the Directory Facilitator, or DF agent. this agent is responsible for keeping the service directory or Yellow page services. A set of static methods can be used to access services from these agents or can be requested by sending an ACL message.

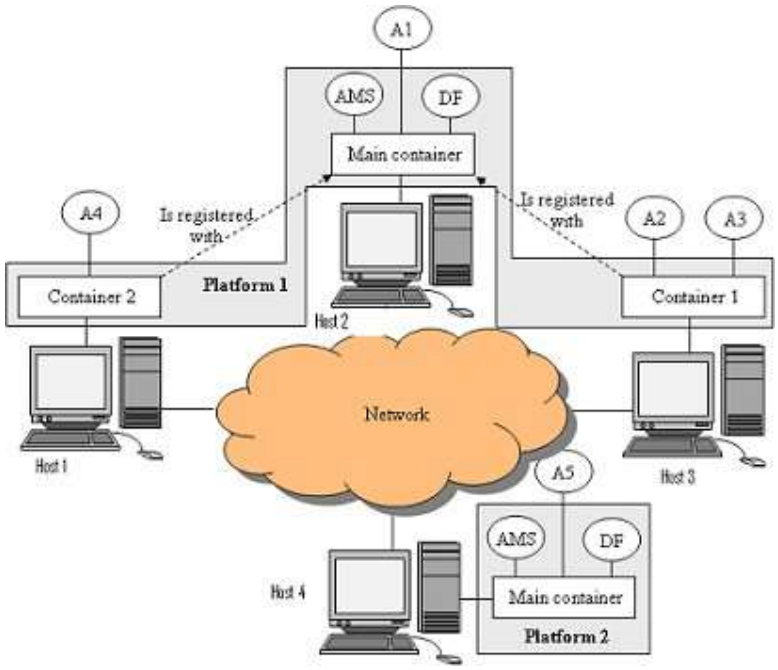

FIGURE 2.1: The Jade Platform (Grimshaw, 2010)

As a middleware, Jade's internal architecture comprises extensible, separated modules which is achieved by writing the code as a collection of independent aspects or services. This results in what they refer to as a distributed coordinated filters architecture. The services are what has been previously described, such as the Message Transport service, the AMS, DF, migration services, and so forth. Every container in a platform sits on a node where these services are kept and managed by a Service Manager. The result is a modular framework which scales by adding or removing services to suit the target platform (desktop, mobile, etc…). All of this, of course, is hidden from the developer who only needs to concern themselves with implementing the agent system desired.

2.2 Sugarscape

Sugarscape is a popular model for agent-based social simulation. The seed of the idea for Sugarscape came from a paper presented in 1969 called Models of Segregation, by Thomas Schelling, an economist and professor (Schelling, 1969). Schelling, this first person to use Agent-based models, was interesting in seeing how certain preferences of individuals in social populations affected who they wanted to live next to. On paper, he used "o"s and "+"s to represent demographic pairs of a population (eg. whites and blacks, boys and girls, etc…) which he distributed randomly along a line on the page. Then, based on rules defining how content an individual subject (symbol) is based on the ratio of symbols in its "neighborhood" (the number of symbols within a window that extends some number to either side of its position), the subject may choose to move to a different position on the line to satisfy the desired ratio.

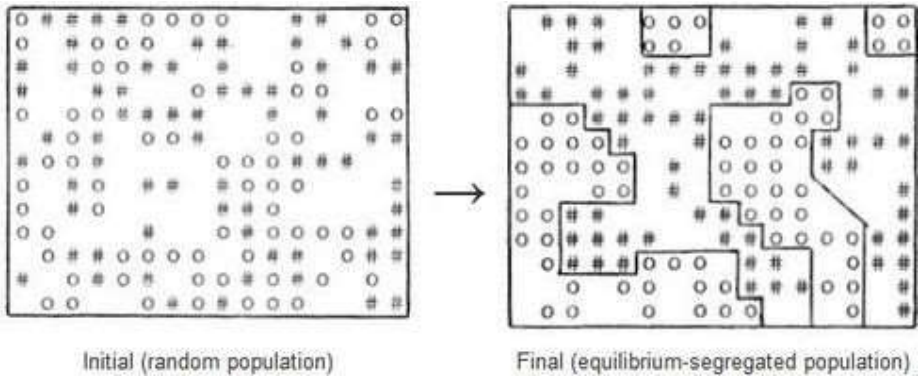

FIGURE 2.2: Segregation Diagrams from a grid version of Schelling's model (Schelling, 1969)

The behavior of these symbols changed as the rules of the symbol's *tolerance* and size of window defining its neighborhood changed, including the ratio of overall symbols in the model. The interesting emergent behavior of these experiments was that neighborhoods still became segregated even when a high tolerance for mixed neighbors was set. As you may see, the model used in the paper has the hallmarks of an agent-based model: individuals acting on a set of rules and reacting to changes with respect to its environment.

Years later in 1996, the book *Growing Artificial Societies: Social Science From the Bottom Up* (Epstein and Axtell, 1996) was published by Joshua M. Epstein and Robert Axtell. In the book, Epstein and Axtell took Schelling's ideas and created a model

that came to be known as The Sugarscape. This model considered a 51x51 grid where each square contained some random amount of sugar. In stepwise fashion, agents which were distributed around the grid searched their area around them up to a limited distance, moved towards squares that produced sugar, then consumed the sugar in that square. In the process of traversing patches and consuming sugar, the agents may act on the environment or other agents in some way, such as spread disease or pollution, trade sugar for spice or information, gain other resources, skills or tools, reproduce or terminate, or any other action befitting the context of the scenario being modeled.

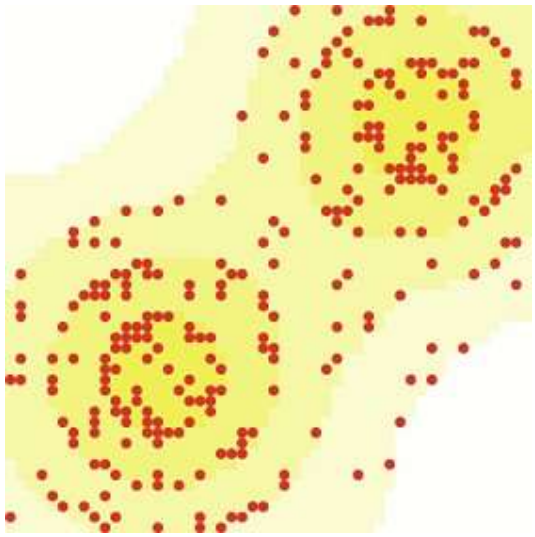

FIGURE 2.3: Epstein and Axtell's Sugarscape Immediate Growback model (implemented in NetLogo) (Li and Wilensky, 2009)

A huge variety of simulations are possible with such an abstract model, and so this has become one of the most widely used across many fields of sociology, science, economics, and more. NetLogo, as mentioned in the previous chapter, has 3 variants of the Sugarscape model among its open library: "Immediate Growback", "Constant Growback" and "Wealth Distribution". There are more implementations on other open-source software as well.

This paper presents the Sugarscape model implemented in the Jade framework. The concept of this implementation tries to remain as faithful to the original as possible. The environment consists of several containers within a single platform. Static sugar *producing* agents and mobile sugar *consuming* agents are randomly distributed among the containers. Producers and consumers will generate and consume, respectively, a small amount of sugar at randomly different rates and amounts per agent. Sugar consumers will broadcast a request for sugar to all producers and receive responses back. The consumer must decide the largest amount of sugar it will gain from a producer, with the restriction that it must be in the same container as the producing agent to accept the proposal. If a *consumer* agent must migrate to a different container to be local to the desired producer, it must pay an additional cost of sugar required to migrate to that agent's location. It must select the best option, balancing the travel cost against the sugar it will gain. The consumer will travel to the producer with the best offer (if needed), accept that agent's proposal, and consume the sugar it receives. If a consumer cannot keep its supply of sugar above 0,

it will terminate. In the absence of a formal grid through which to traverse, the cost an agent pays to migrate to a container, or "patch", might vary, to approximate the sense of *distance* that the original implementation defines.

The next section will explore how the Sugarscape implementation plays out while looking closer at the inner workings of the Jade framework from a user's perspective, illustrating how agents' behavior, communication, and migration services are handled in the framework.

2.3 Initializing the Demo: The Travel Agent

For the user, creating an agent in Jade is as simple as creating a class which extends the base Agent class, then implementing the required setup() method. This method is intended to handle initialization tasks the user might want to perform. The main actions and services an agent provides are typically implemented as behaviors which are described in the next section. However, if an agent is simple and performs only a single task, it can use the setup() method to perform this task. At the end, the agent can be terminated by calling the doDelete() method. Should there be a need to perform any special persistence or clean-up operation prior to the agent's termination, this can be accomplished by overriding the takeDown() method which is invoked by the framework before the agent is terminated.

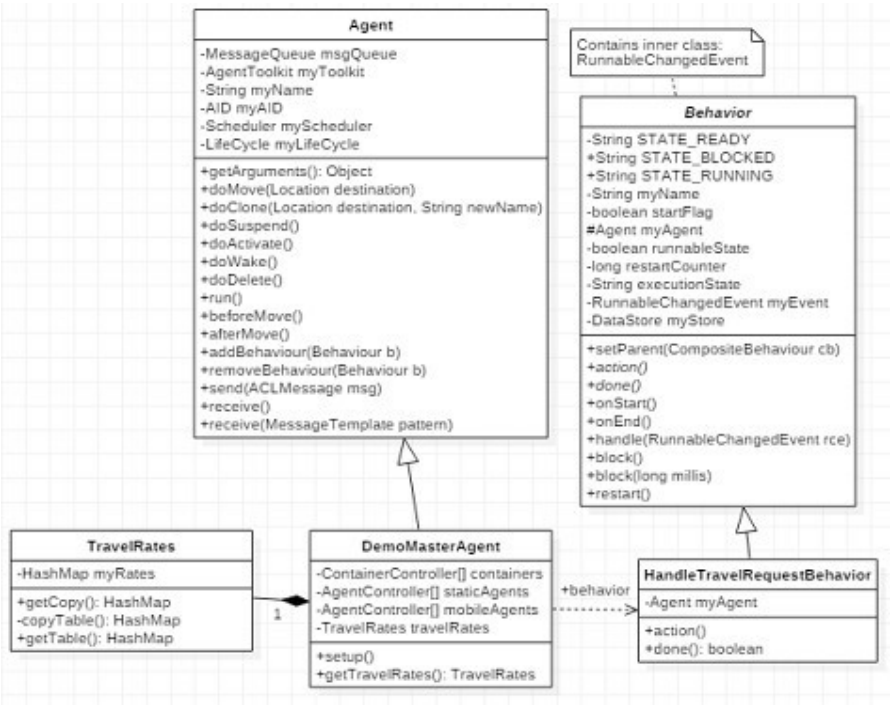

FIGURE 2.4: The DemoMasterAgent Class

There may be a complex setup to implement in the agent-based model, with a potentially wide array of agent types to initialize and varying numbers of each type. Several containers may also need to be generated to distribute the agents among. Regardless of size or complexity, a single agent can be created to handle the entire setup process and begin the simulation. Jade provides a handle to the runtime

instance that can be used to programmatically create containers. This can be accomplished with

LISTING 2.1: DemoMasterAgent: Creating a new Container in setup()

```
jade.core.Runtime runtime = jade.core.Runtime.instance();
String newContainerName = "Container_n";
Profile newContainerProfile = new ProfileImpl();
newContainerProfile.setParameter(Profile.CONTAINER_NAME, newContainerName);
newContainerProfile.setParameter(Profile.MAIN_HOST, "localhost");
ContainerController cController =
        runtime.createAgentContainer(newContainerProfile);
```

The container controller's createNewAgent(String agentName, String className, Object[] args) method can then be used to generate the agents you need. Arguments can be passed to the agent though the Object array parameter during initialization. These can be retrieved with the getArguments() method from the Agent class. Note that the createNewAgent() method does not return a reference to the agent itself but instead an AgentController object used to activate the agent. This adds a layer of protection and avoids exposing any direct reference externally, unless the agent is explicitly programmed to do so. Agents are distributed by randomly selecting one of the existing container controllers each time a new agent is created. The controller is used to add the agent to its container.

LISTING 2.2: DemoMasterAgent: setup()

```
public void setup() {
...
  System.out.println("Starting producer agents");
  for(int i = 0; i < STATIC_AGENT_COUNT; i++){
    String agentName = "ProducerAgent_" + (i+1);
    int pos = (int) (Math.round(Math.random().(CONTAINER_COUNT-1)));
    try {
      staticAgents[i] = containers[pos].createNewAgent (agentName ,
      "examples.myStatic.StaticAgent", new Object []{}); //arguments
      staticAgents[i].start();
    } catch (StaleProxyException e){
      e.printStackTrace();
    }
  }
  ...
}
```

All of this can be accomplished within the setup() method. As a shortcut, the DemoMasterAgent class will pull double duty to also act as a travel agent and provide the service of notifying agents of the cost in sugar required to migrate from one container to another. The sugar cost for traveling from each container to every other is created in a map for the travel agent to manage. The service itself of responding to travel cost requests will be provided as an implemented behavior, which will be registered to the DFAgent. How to register a service will be covered in the next section. In summary, the travel agent's setup() method will:

- create some given number of containers

. create some given number of sugar producer and consumer agents and dis- tribute them by randomly selecting a container controller to initialize each of them into

- create a map of travel costs for inter-container migration
- register itself as a travel agent service with the DF agent on the platform
- add the behavior of the service itself to its behavior list

The next section will look at the creation of the Sugar Producer agent with a closer look at how behaviors and services operate from the user's perspective.

2.4 Behaviors, Registering a Service, and Message Handling: The Sugar Producer Agent

The SugarProducerAgent class is slightly more complex than the DemoMasterAgent agent. The goal of the sugar producer agent is to periodically generate sugar, respond to sugar requests from consumer agents with a proposal that includes some amount of sugar it's willing to offer, and to deliver the proposed amount of sugar to the consumer agent that has accepted its proposal. In terms of setup, this agent needs only to register its service with the DF agent, then add the behaviors for generating sugar and handling request and proposal response messages to its behavior schedule.

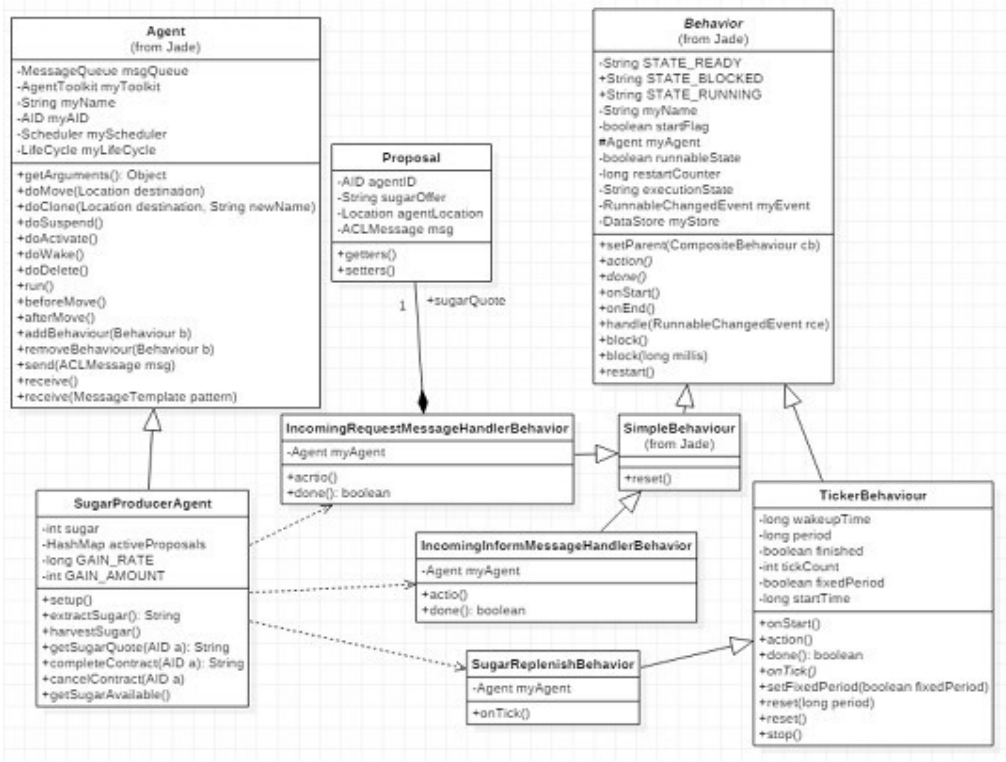

FIGURE 2.5: The SugarProducerAgent Class

2.4.1 Behaviors

A Jade agent has a list of scheduled behaviors which are executed in a round-robin fashion. The agent schedules behaviors by adding them, at the bottom of the list. Behaviors are executed one at a time by calling their action() method. When a behavior reaches the end of its method, if it considers itself complete, then it is removed from the Behaviors list, otherwise it's rescheduled by being placed back at the bottom of the list and executed again when it gets back to the top.

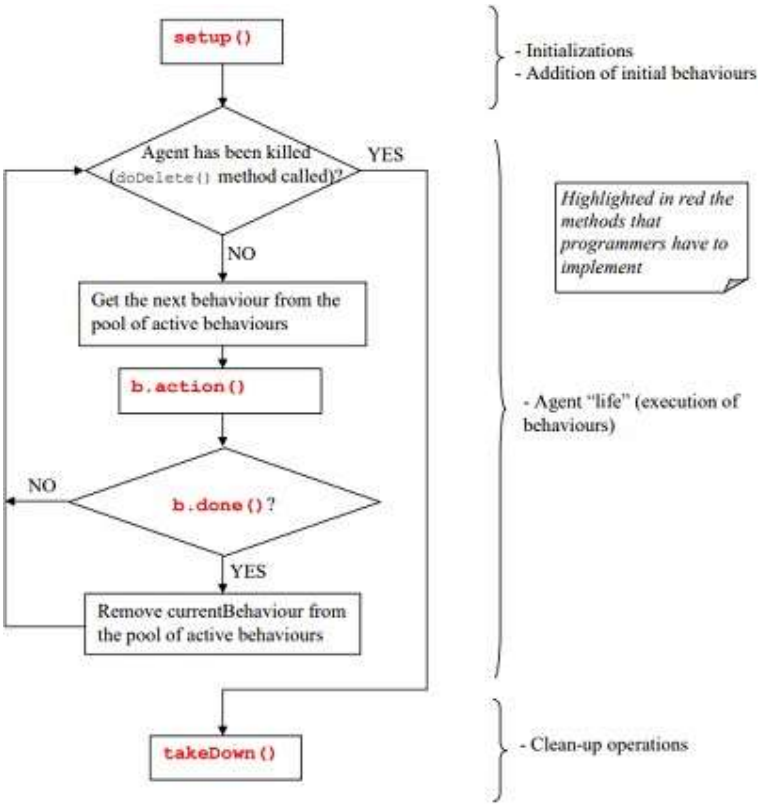

FIGURE 2.6: The Jade Agent Path of Execution (Caire, 2009)

The most basic extension from the Behaviour class is the abstract *SimpleBehaviour* class. From this behavior, three other main abstract behavior types are most commonly extended from: the *OneShot Cyclic*, and *Composite* behaviors. The *OneShot* behavior is just like *simple* behavior except the done() method always returns true, ensuring the behavior is performed only once. In contrast, Cyclic behavior's done() method always returns false. The *Composite* behavior consists of any combination of the other behaviors. Jade provides other abstract behavior templates as well based on this core set, such as *TickerBehaviour*, *WakerBehaviour*, *SequentialBehaviour*, *Parallel- Behaviour*, and others.

The first behavior needed is the agent's *SugarReplenishBehavior*. The rate at which the agent generates sugar should be at a measured and consistent interval. To accomplish this, the behavior extends the TickerBehaviour Class. This behavior cycles through a waiting period of a length specified by the user, then executes its task; at which point it waits again, then performs the task, waits, and so forth. A constructor is built which will call the super constructor and pass in a reference to the

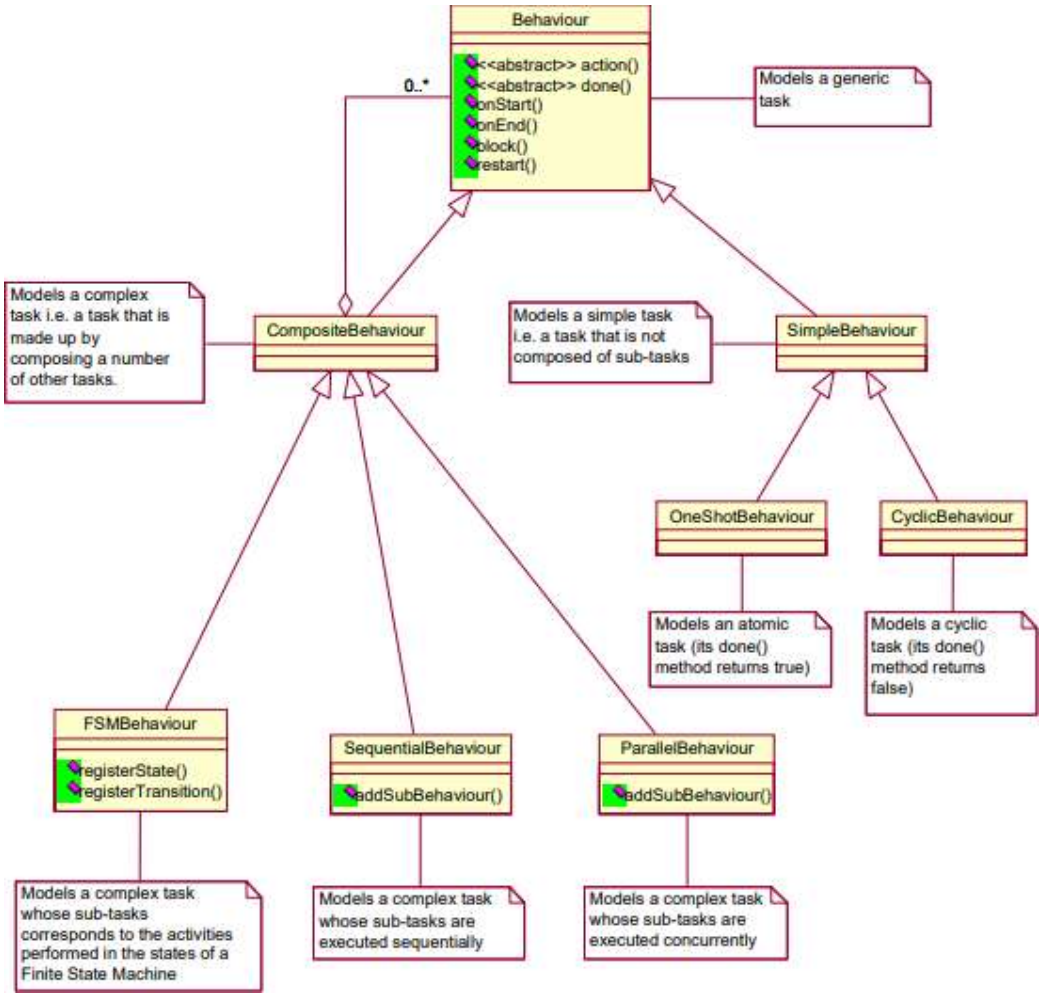

FIGURE 2.7: Jade Behavior Hierarchy (Caire, 2009)

agent to connect the behavior with and a long value to specify the amount of time the behavior will wait until is executes the desired task.

```
public SugarReplenishBehavior(Agent a, long period) {
    super(a, period);
}
```

The *TickerAgent*'s onTick() method is overridden, where inside is defined the agent's action. Because the behavior has a reference to the *SugarProducerAgent* agent, it can call an implemented method defined there which will increase the agent's sugar variable by some given amount.

```
@Override
protected void onTick() {
    ((StaticAgent)myAgent).generateSugar();
}
```

This completes the creation of the simple behavior. It can be added to the agent's scheduler during its setup() method. An instance of this behavior is constructed,

passing in a reference to the agent and a value (representing milliseconds for the tick intervals) as constructor arguments, then the base Agent Class's addBehaviour(Behaviour b) method is used to append it to the bottom of the agent's behavior schedule.

The behavior scheduler executes behaviors sequentially in a single thread. This means the action() method of a behavior completes and returns before the next behavior's action() method is invoked. This model has several advantages such as avoiding synchronization issues and provides improved performance; and the user always has the option of setting a composite behavior with concurrent children scheduling by extending the *ParallelBehaviour* class if performing multiple tasks concurrently is desired.

2.4.2 Registering a Service

Now that the *SugarProducer* agent will produce sugar, it needs to be found by the *SugarConsumer* agents. A service is "published" through the DF (Directory Facilitator) yellow pages service. This requires invoking a static register() method of the DFService class and passing it a *DFAgentDescription* object, which describes the services being provided. The *DFAgentDescription* object requires the Agent's AID and a *ServiceDescription* object which must include the service type and the service name, and optionally the languages and ontologies required to use the service. This demo does not need to concern itself with languages or ontologies, the agent only needs to be discoverable by *SugarConsumer* agents. The service is registered as follows:

LISTING 2.3: SugarProducerAgent: setup()

```
Protected void setup() {
   ...
   DFAgentDescription agentDescription = new DFAgentDescription();
   agentDescription.setName(getAID());
   ServiceDescription sdesc = new ServiceDescription();
   sdesc.setName("SugarProducer_" + this.getLocalName());
   sdesc.setType("sugar-maker");
   agentDescription.addServices(sdesc);
   try{
      DFService.register(this, agentDescription);
   catch(FIPAException e) {
      ...
   }
   ...
}
```

Any agent which solicits the yellow pages service specifying "sugar-maker" as a service type will receive a list of all agents within the platform, regardless of which container they are located, who have registered this service.

2.4.3 Messaging

Once the *SugarConsumer* agent gets a list of service providers, it will send each a request for a proposal. A behavior is needed to handle these requests, as well as the acceptance (or rejection) of that proposal. There are many options for how to accomplish this. As mentioned in chapter 1.5.3, Jade's message model conforms to the FIPA ACL message specification. The FIPA ACL Message Structure Specification described in chapter 1.4.2 requires a message performative be included in the message

content. As a feature of the framework, Jade provides the ability to filter messages by performative type, allowing the agent to pull only the next message which has a matching type from the message queue. If we think of a proposal request aligning with the performative "request" and the acceptance or rejection of that proposal with the performative "inform", then two discrete behaviors can be made which monitor only messages with the respective performative type to handle.

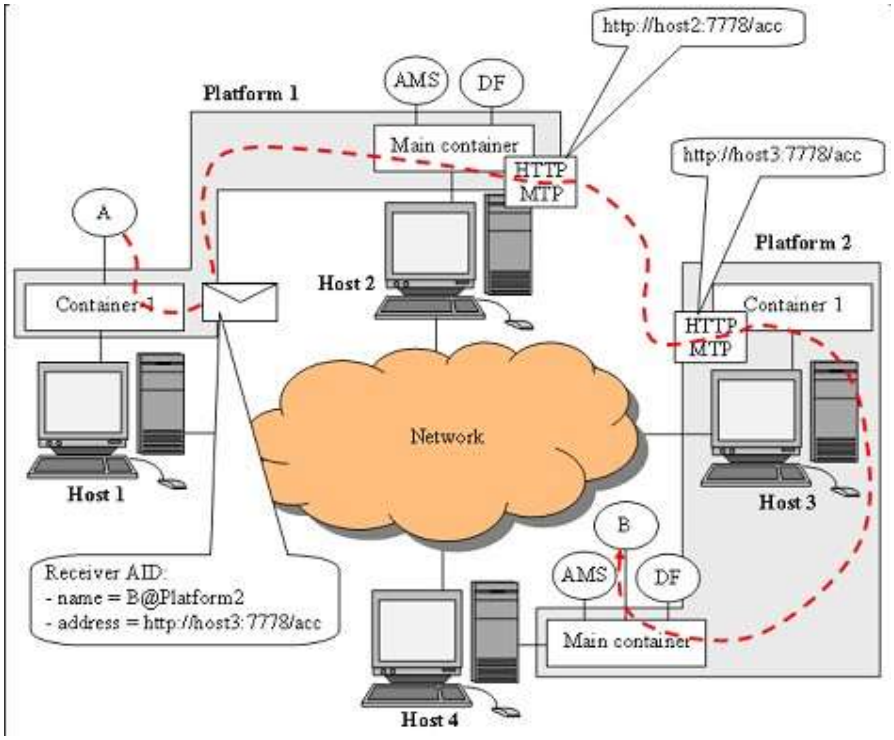

FIGURE 2.8: Message Routing in the Jade Platform (Grimshaw, 2010)

Starting with the behavior for handling proposal requests, a new behavior called *RequestHandlerBehavior* can be made, which will extend the *SimpleBehaviour* class. Inside its constructor, it will call its super constructor and pass a reference to the agent as an argument which is standard practice and will be needed later. There are two ways to retrieve a message from the message queue, by using receive() or blockingReceive(). The latter blocks the agent thread while waiting for a message to enter the queue. While there may be cases where this is useful, using it here would prevent the agent from its other behaviors and could not generate sugar or handle "inform" messages. So the asynchronous receive() method is used where it will pass in a *MessageTemplate* object as an argument. This will specify the communication act, or performative, needed to filter the messages by.

LISTING 2.4: RequestHandlerBehavior: action()

```
Protected void  action(){
  . . .
  ACLMessage msg ;
  MessageTemplate mt =
  MessageTemplate . MatchPerformative ( ACLMessage . REQUEST ) ;
```

```
    // Get a message from the queue or
    // wait for a new one if queue is empty
    msg = myAgent.receive(mt);
    if (msg == null) {
        block();
        return;
    } else{
        // handle message
        . . .
    }
    . . .
}
```

If there is no message in the queue which matches the desired performative, it will call block() which sends the behavior to a list of behaviors in a blocked state. It will stay there until a new message enters the queue and wakes this behavior out of its blocked state to check again. Once a matching message is received, it will extract the content which can be a String or an object. Jade messages also have the feature of creating an automatic reply message using createReply(), which automatically creates a new Message object and populates the receiver slot with the Agent ID (AID) of the agent the original message was received from. The *SugarProducer* agent will then generate a proposal object which will set an offer of sugar to be some portion of its supply, or zero if it has none, then change the message's performative to "inform" and send the message using the send(ACLMessage msg) method of the base Agent class.

LISTING 2.5: RequestHandlerBehavior: action()

```
    protected void action(){
        . . .
    else {
        . . .
        // handle message
        String sugarQuote =
                ((StaticAgent) myAgent ) . getSugarQuote (msg.getSender());
        Proposal p = new Proposal(myAgent.getAID()) ;
        p.setAgentLocation(myAgent.here());
        p.setSugarOffer(sugarQuote) ;
        ACLMessage replyMsg = msg.createReply();
        replyMsg.setPerformative(ACLMessage .INFORM);
        try {
            replyMsg.setContentObject(p);
        } catch(IOException e) {
            e.printStackTrace();
        }
        myAgent . send ( replyMsg ) ;
    }
    . . .
}
```

The behavior to handle the proposal response is virtually the same, the only difference being the performative of the message filter is now set to "inform" and

the action taken should reflect the acceptance or rejection of the proposal, which can be a simple string with either "accept" or "reject" as the message content.

2.5 State Machines, The Yellow Pages, and Mobility: The Sugar Consumer Agent

The *SugarConsumer* agent does most of the work in this demo, and thus, is the most complicated to set up. There are different behaviors it needs to perform but those behaviors depend on the state of the agent. It could be waiting to receive proposals from the agents it sent requests to, it could be trying to decide between those proposals. It may need to migrate, or not, depending on the location of the agent with the best proposal. These behaviors should be scheduled only when they're appropriate.

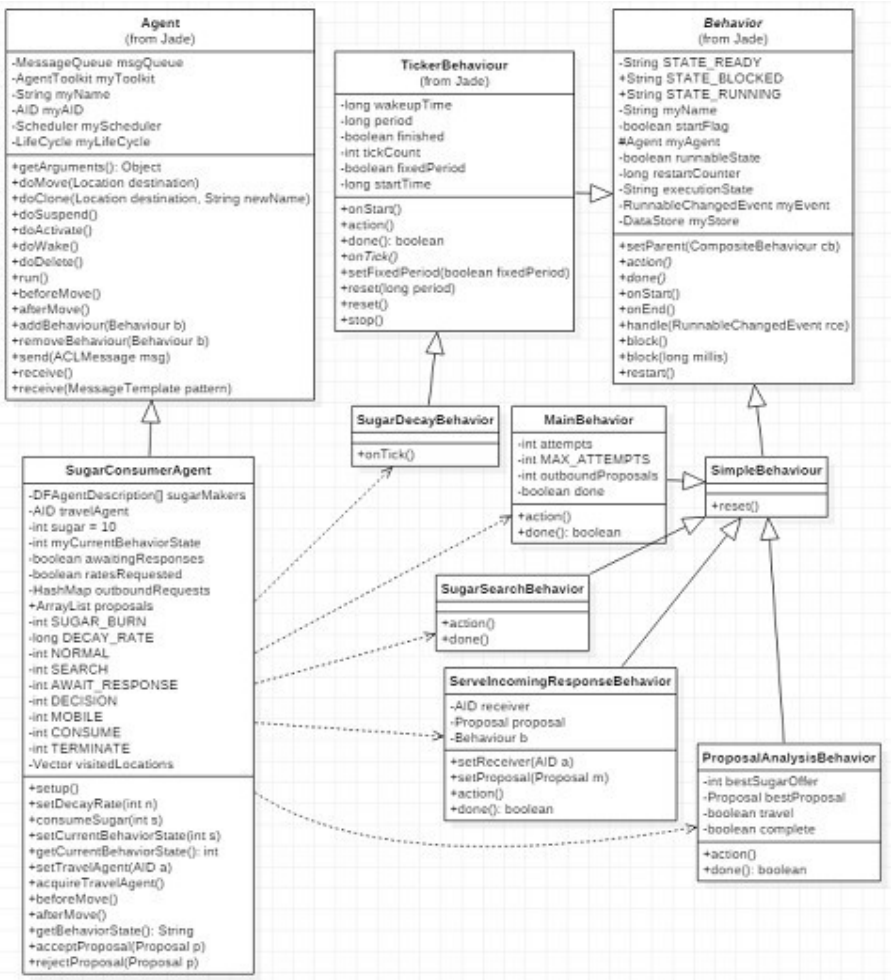

FIGURE 2.9: The SugarConsumerAgent Class

2.5.1 Creating a State Machine

One solution is to create a central behavior for the agent, causing it to operate like a state machine. This behavior is in charge of monitoring the agent's current

state and, if needed, schedule the right behaviors when needed. Considering the agent's goals, the following states are defined to determine its behavior:

- NORMAL: The agent's starting state. This state is reached when the agent is started for the first time or it has just completed consuming sugar and ready to begin searching again for more sugar.
- SEARCH: The agent gets a list of *SugarProducer* agents and sends each of them a proposal request.
- AWAIT_RESPONSE: This state keeps monitoring the messages until proposals are received from all requested agents.
- DECISION: The agent examines all proposals, gets travel costs from the *TravelAgent* for consideration with non-local *SugarProducer* agents, and selects the best proposal.
- MOBILE: This state is activated only if the best proposal requires the agent to migrate to a different container.
- CONSUME: The agent accepts the proposal, receives the sugar, and consumes it.
- TERMINATE: This state is activated if the agent's sugar supply reaches zero.

Separate from any behavior related to these states, the agent will have a behavior always running which periodically consumes its sugar supply by some small amount. This behavior is built and scheduled in the exact same way the *SugarReplenishBehavior* for the *SugarProducer* agent was made in the previous section. The only difference is that the method called in the *SugarConsumer* agent will reduce its sugar supply rather than increase it.

The states described above are each represented by a unique integer value. The current state is explicitly kept by the *SugarConsumer* agent. The main behavior can keep track of the current state through a switch-case and define the appropriate action for the given state. Typically, when the behavior related to a state finishes, it can shift, or even force, the agent to the next state and even schedule new behaviors, or some other action depending on whether it finished successfully or not.

2.5.2 The Yellow Pages Service

In the NORMAL state, the agent checks to see that its message queue and behavior schedule are nominal before switching to the SEARCH state and scheduling its associated behavior. Once in the SEARCH state, the first task is to query the DF's yellow pages service for a list of *SugarProducer* agents by service type. This is very similar to the process of registering for a service, only now we are calling the static method search() from the *DFService* class. The parameters of this method require a reference to the agent requesting the service and a *DFAgentDescription* object specifying one or more of a service type, language, ontology, protocol, or service description. Optionally, a *SearchConstraints* object may also be included as an argument with optional parameters that allow you to specify the max number of results returned or the max depth which limits the recursive depth of the search over the DF federation graph.

LISTING 2.6: SugarSearchBehavior: action()

```
protected void action(){
  ...
   DFAgentDescription template = new DFAgentDescription ();
  ServiceDescription templateSd = new ServiceDescription();
   templateSd.setType("sugar-maker");
  SearchConstraints sc = new SearchConstraints();
  sc.setMaxResults(new Long(100));
  try{
      DFAgentDescription [] results =
     DFService.search(myAgent, template, sc);
  } catch(FIPAException e ) {
  // handle exception...
  }
  ...
}
```

FIPA specifications of the yellow pages service specify that a *DFAgentDescription* matches the search template if all fields specified in the template are present and match the values and should be included in the results.

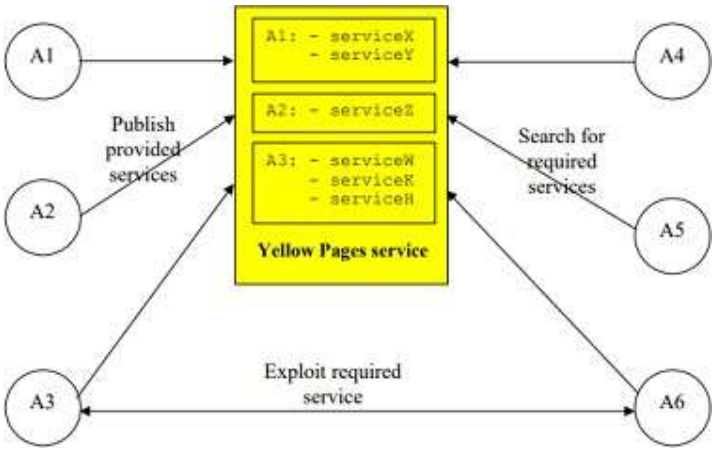

FIGURE 2.10: Jade Yellow Pages Service (Caire, 2009)

Each *DFAgentDescription* object in the results list will contain the AID of the service provider as well as a list of the services matching the template. The AID is extracted and used to generate a message to send the agent of the AID, requesting a proposal. When creating the message, it must pass in the performative as a required constructor argument. The performative "request" is used as the *SugarProducer* agent is programmed to respond to messages with such performatives, but any performative can be used that suits the application's needs. Then the message recipient is set with the AID from the *DFAgentDescription* object and sent using the base *Agent*'s send(ACLMessage msg) method.

LISTING 2.7: SugarSearchBehavior: action()

```
protected void action(){
  ...
  DFAgentDescription dfd = results[i];
```

```
    AID provider = dfd.getName();
    ACLMessage msg = new ACLMessage (ACLMessage . REQUEST );
    msg.addReceiver(provider);
    msg.setContent("sugarquote");
    myAgent.send(msg) ;
    . . .
}
```

In the demo, the AID of the agent set to receive the request for a proposal is passed to a list that tracks outbound proposal requests awaiting response. A behavior needs to be scheduled to handle the message response, once the proposal arrives, to remove the agent's AID from the tracking list. Once all proposal requests are sent out, the consumer agent can transition to the AWAIT_RESPONSE state to continue.

When all proposals are finally received, the tracking list will be empty which causes the agent's state to transition to the DECISION state. In this state, the associated *ProposalAnalysisBehavior* behavior has the agent look over each proposal to see how much sugar is being offered. In the case of proposals from non-local agents, the consumer agent must calculate the offset of the sugar they offer by the cost of sugar required to migrate to that agent. To get the travel rates, the agent uses the yellow pages to look for agents providing the "travel-rates" service in the same way it looked for agents that provided the "sugar-maker" service. A request message is sent to the *DemoMasterAgent* agent, and then receives back a table of current travel rates to all containers. With this, the agent can calculate the net amount of sugar being offered less the cost to travel if needed. Assuming the agent is local, the consumer agent can simply change the state to CONSUME. Otherwise, if migration is necessary, once the best proposal is chosen, the agent will then change the state to MOBILE. The appropriate behavior is then scheduled depending on the new state.

2.5.3 Mobility

On a scale of weak to strong, as discussed in Chapter 1.1, Jade describes their agent mobility model as not-so-weak. Once at the new destination, agents do not completely start over as though they were first created, as weak agents do. Neither are they able to capture every aspect of their state, including the execution environment, program counter, or the execution stack as would be defined for strong mobility. They are instead able to capture and serialize the agent's state in terms of explicit non-transient variables defined in the agent's class. Included in that state data is the agent's scheduler which holds the scheduled behaviors. Once the agent is activated in the new destination, it can start executing the next behavior in the schedule. With the use of the state machine, as has been setup for the consumer agent, it can extend that strength of mobility a bit further.

The Jade *Agent Mobility Service* provides intra-platform mobility. With this service, any agent can migrate to any other container in the platform, but not a container belonging to another platform. An agent can query the AMS (via the white pages service) or DF (via the yellow pages service) for a location(s) which it can use as a destination to migrate to. The location is used as a parameter and must be an object from a class which implements the Location interface. The two classes provided for that in Jade are *ContainerID* and *PlatformID*. The *ContainerID* is used for the intra-platform mobility service. The *PlatformID* is reserved for the Inter-Platform Mobility Service, an add-on that must be explicitly included when launching the Jade application. The inter-mobility service does have the ability to move an agent to a container

in a different platform. It accomplishes this by packing up the agent, and all of its class files, as a content object using the ACLMessage system. In this system the agent, and its code, must be gathered into a single, potentially large, package to be moved. This carries some disadvantages, such as increased resource utilization over the network and a potential degree of redundancy in situations where some, but not all, of the agent's class code may already exist at the destination platform. The demo functions entirely within one platform, so the focus will just be on the intra-platform mobility service.

An agent that wishes to use mobility services is required to have registered the *jade-mobility-ontology* ontology. Briefly, ontologies are a set of concepts and symbols used to express some specific domain of knowledge or information. These concepts and symbols are structured using a syntax that is represented by a language. For instance, while FIPA does not require using any specific language, it does recommend a type called the Semantic Language (SL) for communicating with the AMS and DF. Ontologies and languages are used to help an agent parse and decode information it receives from another agent or platform through ACL Message content, or to help it perform actions defined in classes related to the ontology. While they are a useful and advanced feature of the Jade platform, this demo only requires that the existing mobility ontology be registered to use the related service, not to create its own, and any in-depth discussion about ontologies falls outside the scope of this paper.

An agent can be moved directly or indirectly. That is, it can move itself, or by others. One way to move an agent is by sending a FIPA request message to the AMS with the ontology of the message set to *jade-mobility-ontology* and the language set to *FIPA-SL0*. The content slot must be set with a *MoveAction* object, which in turn must be set with a *MobileAgentDescription* that must include the name of the agent being moved (which may be our agent or some other) and the destination as a *Location* object. Once received, the AMS will locate the agent to be moved and perform the *move-agent* action on it.

The simpler and more direct way, and the method employed in this demo, is for the agent to move itself by calling its doMove(Location destination) method. To spare the agent from having to query the AMS for the location of the agent whose proposal it wants to accept, the *SugarProducer* agent includes its *ContainerID* object with the *Proposal* object. The agent extracts the location from the proposal, pays the required cost in sugar to travel to the destination and, if the cost does not put its sugar supply to zero, switches its state to MOBILE and calls its doMove() method.

LISTING 2.8: ProposalAnalysisBehavior: action()

```
public void action(){
  // extract the location object from  the proposal
  Location destination = bestProposal.getAgentLocation();

  //pay the cost of movement now
  String from = myAgent.here().getName(); // current location
  String to = bestProposal.getAgentLocation().getName(); // destination
  int cost = ((MobileAgent)myAgent).travelRates.get(from).get(to);
  ((MobileAgent)myAgent).consumeSugar(cost);

  // if agent is not dead , then proceed with migration
  if(!((MobileAgent)myAgent).getBehaviorState().equals("TERMINATE")){
    ((MobileAgent)myAgent).setCurrentBehaviorState(
```

```
            ((MobileAgent)myAgent).MOBILE);
        myAgent.doMove(destination);
    }
}
```

Once migration has been initiated, the agent will change its internal state from ACTIVE to TRANSIT. Before the actual migration happens, the agent's beforeMove() method is invoked at the source location to give the user the opportunity to have the agent perform any necessary tasks prior to the agent's serialization. This may include things such as closing any instances of a GUI, releasing local resources, or addressing any transient variables belonging to the agent it will not bring with it. Upon arrival to the destination location, but before the agent and its behavior scheduler is activated, the agent's afterMove() method is then invoked. This allows the user to reestablish the agent's resources, services, re-register ontologies or services, or any other action required. As the state of the state machine is set to MOBILE, the main behavior, which is still scheduled and running, will correctly detect that the agent has arrived at the new destination. It will message the *SugarProducer* agent to accept the proposal, update its state to CONSUME, and schedule the consume behavior to catch the *SugarProducer* agent's response with the sugar and consume it. This completes a cycle in the life of the agents of the SugarScape demo.

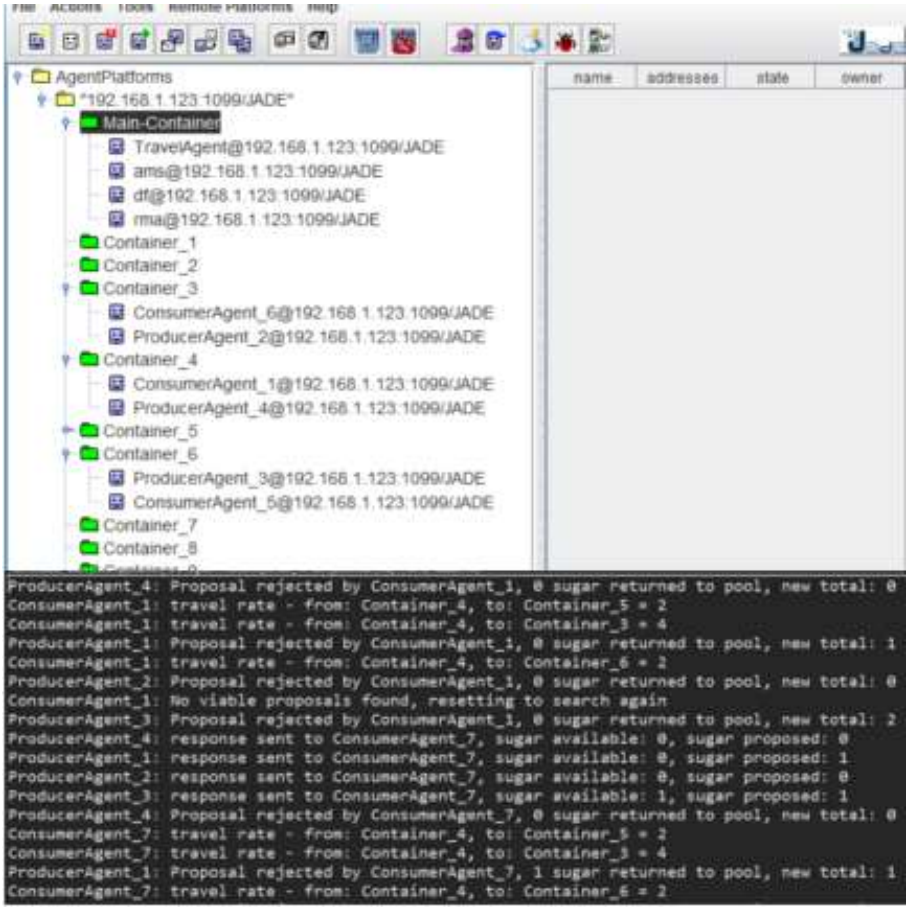

FIGURE 2.11: Sugarscape Demo in Jade. ConsumerAgent_7 has died because the agent in its container (4) has no sugar to offer and the other producers cannot offer enough to offset the cost of travel.

Jade is a very robust framework, and there are a great deal more features it includes which cannot be covered here but are quite useful. Much can also be said about the underlying services which power the framework itself, unseen by the user. Many of these features and services are modular as they are only useful under certain situations and not needed by every user, particularly given certain resource- limited platforms like mobile. This brings up the interesting question: what are the most essential components necessary to create a mobile agent frame- work? The next chapter will discuss the components vital to creating a simple mobile framework, the critical problems which need to be addressed, and present a complete implementation of such a framework.

Chapter 3

Implementing NOMAD: A Basic Mobile Agent Framework in Java

Depending on the complexity of the agent-based software, there are many advantages to starting with an existing framework. The user is spared from the time- consuming work of implementing the robust and feature-rich toolkits that exist today. Open-source libraries are available in a variety of architectures. Some of these architectures have modularity and versatility, many with libraries that have well-documented APIs and strong communities. But there is always a case where no single implementation can address all the needs one may have to address in an application. For the variety of frameworks available, each is built to provide only a limited scope of features. Some of those features may overlap between frameworks, like message passing, mobility, deliberative reasoning, ontology-defined services, security, and more. But how each framework implements such features can be significantly different and impact how a developer might choose one over another. There are many cases where no framework can address all issues. Or even if there were, it may come with a host of other unnecessary features and boilerplate code to be dealt with to use what is needed. The time required to use a complicated framework, the overhead needed to use the features desired, or other required features not needed that drain resources, may make using an existing implementation more trouble than it's worth.

In such cases, the solution may be to implement a basic framework from scratch. This allows control over the balance between a stable, feature-rich application that is also lightweight and can execute as needed. Luckily, building an agent-based system isn't too hard. It simply needs the agent itself and a platform to provide the services it needs. It's when mobility is incorporated that more care is needed. This is especially the case if agents operate asynchronously. This chapter will focus on building a simple framework called NOMAD, that will provide the most essential elements which define an agent-based application. It will incorporate only the absolute essential components for a mobile agent framework. These components are the asynchronous agent, the platform, communication, and mobility, each of which will be covered in the sections below.

3.1 An overview of the NOMAD Framework

Even with the goal of building the most basic framework possible, certain choices need to be made in terms of architecture. Typically, these choices are determined in the context of the problem that the application will address. The goal of NOMAD is to provide the core essential services to maintain a working mobile agent application, but modular and flexible enough in structure to allow the user extensibility for new features and existing components. To test the features of the framework, a simple

agent demo program will be built on top of it. This program will create several static and mobile agents, each with the goal of interacting with as many other agents as each can find. Mobile agents will be able to migrate to other platforms while static agents must remain in place. This should be sufficient to showcase the core features to be included.

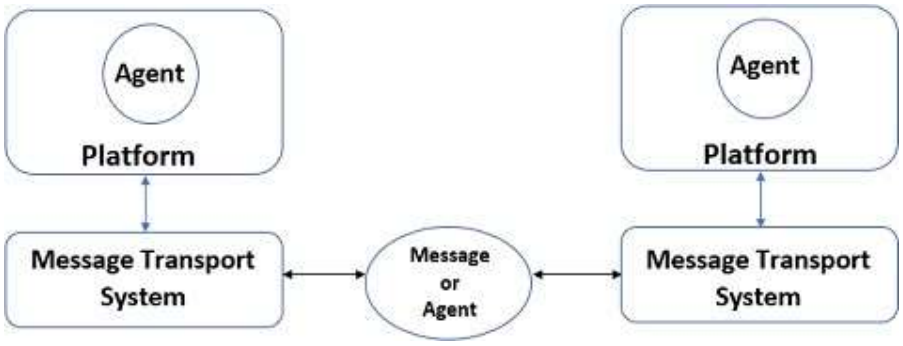

FIGURE 3.1: The NOMAD model

NOMAD will be based on a Reactive architecture. More specifically, there will be no planning or behavior features that will enable pro-active behavior or reasoning. To create one requires defining a specified design which then locks the user into the chosen method. Moreover, the user always has the option of adding their own approach within the existing space for the agent's reactive action. Each agent in NOMAD operates in its own thread. Its lifecycle is managed by the platform it resides in. The platform also provides other services to the agent, such as migration, communication, and directory services. In comparison to the Jade framework discussed in the previous chapter, it fulfills the roles of the Platform, container, DF agent, and AMS agent combined. Similar to the way a Jade container sits over a node that provides all the actual services, each platform in NOMAD has a single node called the Message Transport Service (MTS). The MTS executes the network services and related tools for communication and transport, as well as File I/O services for persistence features and logging.

As a more in-depth look is taken at each of these components, we'll discuss the potential pitfalls, consider alternative approaches, and propose possible extensions and new components to add to the framework. At the end, NOMAD will be a ready-to-go framework that can be built on and shaped as the user requires.

3.2 The Agent: Asynchronous Autonomy

As the framework should be versatile enough to allow a user to create many different kinds of agents, one cannot predict what form those types of agents may take. The simplest approach is to create an abstract agent class that enables mobility, communication, and its own thread in which to run tasks which the user can simply extend and build what they need over it. The Agent class is specified as abstract because the user should not be able to create agents directly from it. This would cause them to introduce more code into the class and risk unintentionally breaking its functionality in some way. Implementing Serializable is necessary for any class that may migrate over the network or have its information written to file, which will be discussed later.

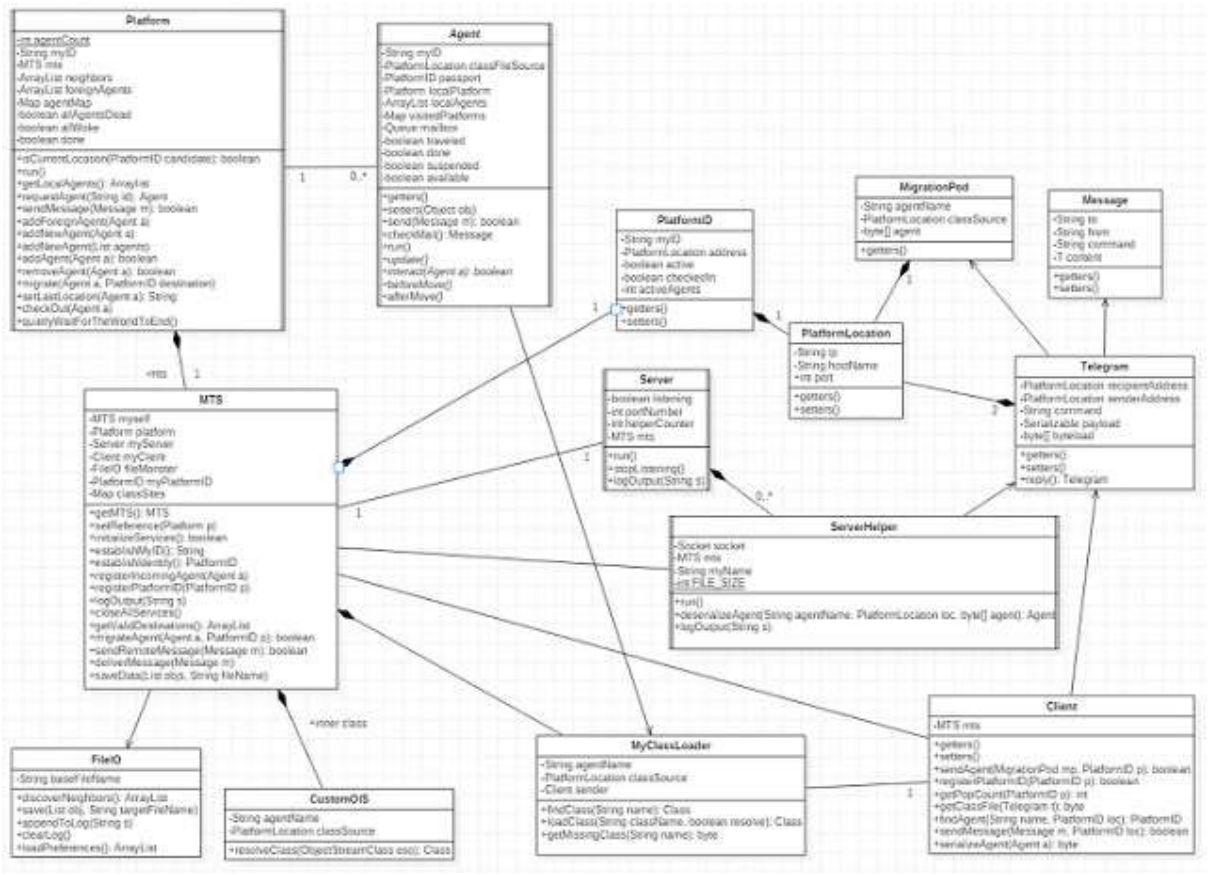

FIGURE 3.2: The NOMAD Framework

The essential attributes the agent will need to have is an ID, which may be as simple as a String or its own object class. For the sake of stable message and agent mobility, the ideal situation is a system which can guarantee that each agent's ID is unique within the system. If we allow a platform to connect dynamically at runtime, it becomes very difficult to make such a guarantee without some sort of separate ID system that can assign aliases to agents. This would be similar to a DNS lookup system which keeps a table of web URLs that map to unique IP addresses. For now, the name is kept as a String and the Platform assigns an ID to the agent. An agent's ID consists of the name of the platform it originated from (the host name of the computer) concatenated with the port number that the local server is listening on, followed by a "-n" where n is a unique integer. The platform tracks this value as a static counter. An example of an agent ID is "myComputer8888-1". The combination of host name, port, and number help to ensure agent names are as unique as possible.

A reference to the agent's local platform is assigned to it and used when it needs to call up services the platform provides. A PlatformLocation object is also assigned, called classFileSource. This object holds information about a platform location, such as the host name, IP address and port number. The agent should always know which platform it originated from so that if it arrives to a platform that does not happen to have its class files in the JVM's class path, the destination platform's class loader can know where to retrieve

them from. The agent also needs to know which platforms it's already been to, so a HashMap keeps that information for it. Lastly, the agent needs some boolean flags to indicate various states such as *traveled*, *done*, *suspended*, and *available*.

As Agent extends Thread, it's required that it implements the run() method. Here is where the agent performs any tasks as defined by the user. But once the method is done, the agent thread dies (unless it's a daemon thread) and the agent becomes inert except to any outside calls to its methods. In other words, it cannot run its own run() method again to allow continuous action. To avoid this, a looping behavior should be used, where the user can then control when that loop stops. To accomplish this, a while loop is used where the conditional argument is the agent's *done* attribute. Inside the loop, an abstract update() method is called which the user must implement to decide how the agent should behave. When the user decides the agent is done and should terminate, they must check out with the platform by calling the platform method checkout() which will set the agent's *done* flag to true and break the loop, ending the thread.

The agent will also need abstract methods related to migration. One, called beforeMove(), will allow the user to act before the actual migration of the agent happens. This will let them save any information requiring persistence, manage objects and data related to any transient variables, or even have the option to cancel the migration itself. The other will be afterMove(), which gives the agent a chance to perform necessary task before it is activated in the new platform, such as reconnecting or instantiating variables.

An abstract method called interact(Agent a) will also be included. An agent can get access to another agent directly by querying the Platform, and use it to call the interact() method on the other agent. This is a simplified form of agent communication included as a simple example of how agents can interact. For the purposes of the demo, this method is implemented in the MarketAgent agent example to "say hello" to the target agent. In practice, however, it's not recommended for actual use outside of debugging as there are security risks in allowing agent to have direct access to each other. Additionally, there are other risks such as the agent migrating and terminating in the middle of their interact() method being executed. The preferred way to communicate is to do so via message passing, as this can be handled asynchronously when the agent wants, which will be covered in the communication section.

LISTING 3.1: The Agent Class

```
package platform;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public abstract class Agent extends Thread implements Serializable{

    private String myID;
    private PlatformLocation classFileSource;
    private PlatformID passport;
    private Platform localPlatform;
    private ArrayList<Agent> localAgents;
    private Map<String, Boolean> visitedPlatforms =
```

```
                    new HashMap<String, Boolean>();
    private boolean traveled = false;
    private boolean done = true;
    private boolean suspended = false;
    private boolean available;

    public Agent ( ) { }
        . . .
       // getters and setters
        . . .
    public void run ( ) {
        while ( ! done ) {
                while (suspended) {
                        //suspended while migrating
                }
                if (traveled){
                   //agent has awoken for the first time since migration
                   if (localPlatform.getMyID()
                                .equals(this.getPassport().getHostName())){
                        // I landed where I should have
                   } else {
                        // something went wrong , I 'm not where I intended to be
                         if(this.getVisitedPlatforms().get(localPlatform.getMyID()) != null &&
                         this.getVisitedPlatforms().get(localPlatform.getMyID())){
                            localPlatform.logStatus(this.getMyID() +
                                    " : something went wrong and I 'm back where " +
                                    " I emigrated from , so I'm shutting down now.");
                            localPlatform.checkOut(this);
                        }
                   }
                   this.setTravelled(false);
                   this.setPassport(null);
                   this.visitedPlatforms.put(localPlatform.getMyID(), false);
                }
                if(!done )
                        this.update();
                }
                this.interrupt();
            }
            public abstract void update();
            public abstract boolean interact(Agent a);
            public abstract void beforeMove() ;
            public abstract void afterMove();
}
```

3.3 The Platform: Managing the Agent Lifecycle

Like the Agent, the Platform class runs in its own thread, but does not need to extend Serializable. As a nexus for agents that are arriving, departing, and changing

states, fulfilling various requests and services to those agents, accessing shared resources, and coordinating the update and retrieval of data through the MTS, the Platform must engage in a delicate dance of multitasking. In terms of architecture and the ecosystem of connected platforms, there are 2 main approaches to take. The first considers each platform to be independent, decentralized entities that form *peer- to-peer* relationships with each other. In addition to being a more basic structure, an advantage of this is that launching the application or connecting it to other platforms remains the same process no matter where it's run. There is also better fault tolerance as there's no critical node that, were it to fail, would collapse the entire network of platforms.

The other option is to have one platform designated as a "master platform" where all others register to it. The advantage here is that directory and other information services can be centralized, as opposed to the peer-to-peer structure where a plat- form may need to query multiple others to find an agent. There are hybrid versions of this as well, for example a peer-to-peer structure where certain platforms can be designated managers of services like the agent directory or alias mapping. As NOMAD is focused on being a starting point for building your own framework, it's built with a basic peer-to-peer structure.

On construction, the platform gets a reference to the MTS singleton object and uses it to establish its own identity. The platform identity is kept in a PlatformID object. This object holds the platform's name which, similar to the agent, is the computer's host name concatenated with the port number used by the server. It also holds a PlatformLocation object which encapsulates its hostname, IP address and port number, plus a count of the active agents running locally, and flags to help keep track of its registration status with other platforms. Once a user creates the agents they want and adds them to the platform instance, they call the platform's start() method which is the launch point for the core application. The agents are not started at this point, however, the platform will handle that once it has initialized all of its components. Instead the platform creates the agents' IDs, registers them to its directory, sets their local platform to its location, sets their classFileSource location and adds the local platform location to the agent's visited location history. In the demo, a number of static and mobile MarketAgent agents are created using this process.

```
public synchronized void addNewAgent(Agent a){
   a.setMyID(this.getMyID() + "_" + (Integer.toString(++ this.agentCount)));
   a.setlocalPlatform(this);
   a.getVisitedPlatforms().put(this.getMyID(), true);
   a.setDone(false);
   a.setName(a.getMyID()) ;
   a.setClassFileSource(mts.getMyPlatformID().getlocation());
   synchronized(agentMap) {
      agentMap.put(a.getMyID(), a);
   }
}
```

Once inside the run() method, the platform has the MTS initialize all of its services. This begins with starting the File I/O services to enable logging. All agent, server, client, and platform status messages get piped to the logging service through the MTS. The File I/O service will then read in from a separate XML file information about other platforms that will also be running. It will attempt to connect to, and register with, these platforms once the server has finished initialization. It builds

the neighbor platforms list, passes the list back to the platform and then proceeds to launch the server.

The server too runs in its own thread. Once running, the MTS will then launch a client and iterate through the neighbor platforms list, sending its own PlatformID to each of them. the server, client, and network transport will be discussed in more detail in the next section. Once it's able to successfully send its PlatformID to all neighbors on the list, the MTS completes its initialization and passes control back to the platform.

LISTING 3.2: The Platform Class: run()

```
public void run () {
   // initialize Message Transport Services
   // (file/log, TCP/IP, preference/neighbor data loading)
   if(mts.initializeServices()){
      int checkedIn = 0;
      int attempts = 1;
      this.logStatus(
            "Verifying all neighbor platforms have checked in...");

      while (checkedIn < this.neighbors.size() && attempts < 5) {
         this.logStatus("Check in query " + attempts + " of 5...");
         checkedIn = 0 ;
         synchronized(neighbors){
            Iterator<PlatformID > iter = neighbors.iterator();

            while (iter.hasNext()){
               if(iter.next().isCheckedIn())
                  checkedIn++;
               }
            }

            this.logStatus(checkedIn + " of " +
                        this.neighbors.size() + " platforms checked in");
                        attempts++;

            if(checkedIn < this.neighbors.size()){
               try{
                  Thread.sleep(2000);
               } catch (InterruptedException e) {
                  e.printStackTrace();
               }
            } else{
               break ;
            }
         }
         if(checkedIn >= this.neighbors.size()){
            this.logStatus(
               "Initialization successful, waking agents to begin simulation...");
            // start services for all registered agents
            agentMap.forEach((key, value) -> {  –
              agentMap.get(key).start();
```

```
            });
            allWoke = true;
            if(foreignAgents.size() > 0){
                for(Agent a : foreignAgents) {
                    this.addForeignAgent(a);
                }
            }
            this.setPriority(MIN_PRIORITY);
            while (!done) {
              if(this.agentMap.isEmpty()){
                 quietlyWaitForTheWorldToEnd();
              }
            }

            this.logStatus("Done called");
            if(!agentMap.isEmpty()){
               agentMap.forEach((key, value) -> {
                  Agent a = agentMap.get(key);
                  a.setDone(true);
                  a.interrupt();
               } ) ;
            }
            mts.closeAllServices();
            this.interrupt();
         } else{
            System.out.println("Error: Platforms not  checked in, exiting.");
            System.exit(1);
         }
   } else{
         System.out.println("Error initializing MTS services, exiting.");
         System.exit(1);
   }
}
```

Successful initialization means that all of the neighbor platforms are also up and running. At this stage, the platform will now wait until all other platforms have sent their PlatformIDs to it. Once received, the platform is now fully connected to all others. The local platform will now begin to register the agents it was given and start to activate them. It keeps track of the agents using a Map where the key is the agent's name. It iterates through the map and calls start() on each. When complete, the platform raises an internal flag to indicate that all agents have been activated. It does this because from the moment it sends its PlatformID out and receives a PlatformID back from the first neighbor platform, that neighbor platform could potentially already be fully connected and have already started all of their own agents. Moreover, those agents could already have attempted to migrate to this local platform. Incoming agents from other platforms are not allowed to be registered and activated until all native local agents have been. It will check a queue to see if any foreign agents are waiting to be registered. If so, it will register those agents as well and activate them into the pool. Adding a foreign agent in similar to adding a new agent except the platform will not give the agent a new ID or set its classFileSource variable. It will also call the agent's afterMove() method

and allow the agent to act before calling its start() method.

At this stage, the process of initializing the platform and deployment of its agents is complete. With the agent demo, agents should interact with as many other agents as possible. Once they cannot find any more agents whom they've not yet met, they will terminate. To facilitate this, the platforms should stay running until all the agents in all platforms are no longer running. A simple way to achieve this is to have each platform's thread go into a loop until its local agent list is empty. This occurs when all of the agents have either migrated to other platforms or terminated. It will then go into a looping method to start polling other platforms to see what their population sizes are. If it discovers that all other platforms have no running agents, then the platform will shut itself down, ending the application. Note that this is a condition specific towards the demo and is not ideal for general use.

As an alternative, a graphic user interface would enable user interaction with the platform, allowing the launch and termination of agents, create and send messages, or shut down the platform upon request. Shutting down the platform involves the MTS having the server stop listening and then closing its own thread. It's here that the user may wish to write any persistent information to file if there are settings or configurations that wish to be saved.

LISTING 3.3: The Platform Class: quietlyWaitForTheWorldToEnd()

```
public void quietlyWaitForTheWorldToEnd(){
    this.logStatus("Local pop 0 , quietly waiting for the world to end...");
    while(this.getAgentMap().isEmpty()){
        mts.getValidDestinations();
        if(allMyFreindsAreDead()){
            this.done();
            return;
         } else{
                try{
                    Thread.sleep(2000);
                } catch(InterruptedException e) {
                    e.printStackTrace();
            }
        }
    }
}
```

The other core service the platform provides is to facilitate migration. When an agent wants to move, it must decide which platform to migrate to. Depending on the circumstance, it needs information about the other platforms where it's possible to travel to. As discussed in the previous chapter, when a platform conforms to standardizations such as FIPA, it provides a white pages service for locating other agents, or a yellow pages service for locating by services provided, which includes the address information needed to migrate. For the peer-to-peer architecture, the decentralized nature of the structure forces an agent or platform to have to iteratively query each its neighbors to look for a specific agent. For NOMAD, and the purpose of the agent demo, a more simplified service is performed. A list is gathered of platforms with a population greater than zero. An agent can request this list and then decide for itself which location it would like to migrate to, calling the platform's migrate() method using itself and the chosen location as arguments.

Once received, the platform will shift the agent into a suspended state. It then calls the agent's beforeMove() method to allow it a chance to perform needed tasks

prior to migration as mentioned in the Agent section above. The platform then switches the agent to a migration state by setting its traveled flag. When the agent is restarted at the new platform, this state will trigger the agent to check that migration occurred as intended. The agent is then passed off to the MTS to perform the actual migration. If successful, the platform switches the agent to a termination state by setting the agent's done flag, then removing it from its suspended state and calling interrupt on its thread. This allows the agent's thread to end, completing its termination. The agent will be later removed by Java's automated garbage collection.

LISTING 3.4: The Platform Class: migrate()

```
public synchronized void migrate(Agent a, PlatformID destination){
  this.logStatus("emigrate request received, removing " +
      a.getMyID() + " from local roster");
  a.setSuspended(true);
  this.removeAgent(a);

  // allow agent to act before migration
  a.beforeMove();

  // have MTS migrate the agent to the chosen address, if present
  if(destination != null){
    this.logStatus("BestCandidate selected, migrating Agent " + a.getMyID()
      + " to Platform: " + destination.getHostName());
     a.setlocalPlatform(null);
     String last = this.setLastLocation(a);
     a.setPassport(destination);
     a.setTravelled(true);
     if(mts.migrateAgent(a, destination)){
        // migration was successful, we can stop the agent thread.
        a.setDone(true);
        a.setSuspended(false);
        a.interrupt();
     } else{
        // migration failed
        this.logStatus("Migration attempt failed, reactivating agent"
                        + a.getMyID() + " to local platform...");
        a.getVisitedPlatforms().put(this.getMyID(), false);
        a.getVisitedPlatforms().put(last, true);
        this.addAgent(a);
        a.setlocalPlatform(this);
        a.setPassport(null);
        a.setSuspended(false);
    }
  } else{
     this.logStatus("Address not valid, reactivating Agent "
                + a.getMyID() + " to local Platform");
     if(this.addAgent(a)){
        a.setSuspended(false);
     } else{
        this.logStatus("agent reactivation failed, shutting agent down." ) ;
```

```
            this.checkOut(a);
        }
    }
}
```

If the MTS reports instead that migration failed, the platform will roll back the agent's visited platform data, relink itself as the local platform, clear the agent's passport to null, and unsuspend the agent to resume normal function. Note that it does not turn off its traveled flag. When the agent resumes function, it will still perform its destination check, realize that it is still in its original platform (or more specifically, that it's not in its intended platform), notify the user of the error, and shut itself down. The user may instead wish to throw an exception or perform some recovery or additional task, if desired.

3.4 Mobility: Network Transport Service

From a top-level perspective, NOMAD takes an agent that wants to migrate, packs it into an object for transport, then inserts that object as a payload into a telegram along with a command instructing the destination platform how to handle the payload. The telegram is then sent to the destination platform, where the agent is appropriately handled and deserialized before being registered and activated.

LISTING 3.5: MTS: migrateAgent()

```
public boolean migrateAgent(Agent a, PlatformID p){
    byte[] ba = this.myClient.serializeAgent(a);
    MigrationPod mp = new MigrationPod(a.getMyID(),
                                       a.getClassFileSource(), ba);
    boolean result = this.myClient.sendAgent(mp, p);
    p.setActive(result);
    return result;
}
```

In the MTS, the agent is first serialized into a byte array, then packed into a MigrationPod object along with the agent's ID and the location object. The MigrationPod is then sent to a client to be transmitted.

The client creates a new Telegram object, setting the receiver slot with the destination location and using its own PlatformID location to set the telegram's sender slot. It then inserts the MigrationPod as the telegram's payload and sets the com- mand string as "agent". A socket is then opened to the destination platform and the serialized Telegram is sent.

LISTING 3.6: Client: sendAgent()

```
public boolean sendAgent(MigrationPod mp, PlatformID p) {
    Telegram t = new Telegram(p.getlocation(),
                              mts.getMyPlatformID().getlocation());
    t.setCommand("agent");
    t.setPayload(mp);

    try{
        mts.logOutput("Client: Opening connection to " +
                      t.getRecipientIP() + " (" + t.getRecipientHostName() + ") on port " +
                              t.getRecipientPort() + " ...");
```

```
        Socket cSocket =
                new Socket(t.getRecipientHostName(), t.getRecipientPort());
        ObjectOutputStream oos =
                new ObjectOutputStream(cSocket.getOutputStream());

        mts.logOutput("Client: migrating Agent " + mp.getAgentName() + " to " +
        t.getRecipientHostName());

        oos.writeObject(t);
        oos.flush();

        mts.logOutput("Client: Agent migration complete");
        cSocket.close();

        return true;

    } catch(IOException e){
        mts.logOutput("Client: Couldn't get I/O for the connection to " +
        t.getRecipientHostName());
        return false;
    }
}
```

It's a trivial exercise in Java socket programming to implement methods that transmit an agent from one network location to another. As noted in the Agent section above, any object being transmitted needs to extend the Serializable interface so that Java can encode the object into the stream of bytes that stores the object's state; and correctly recovers that state when deserialized at the destination. This capturing and reacquisition of state forms the minimal basis for an agent to have weak mobility.

The key challenge with agent mobility comes in situations where an agent is defined and activated at one platform but needs to move to a different platform running in a discrete JVM that does not contain the class files that define the agent. This is a common case in distributed computing and one of the distinct features that defines mobile agent applications. How is new code introduced to a running application?

The Java language stands apart from other compiled languages like C, C++, Go and others in that the Java Virtual Machine enables classes to be loaded dynamically during runtime rather than only during compile time. It loads classes only when it needs them. Java uses a hierarchy of *class loaders* to accomplish this. Each loader in the hierarchy builds paths to the class files the JVM will read from and load into the virtual machine. There are several kinds of class loaders, not all may be used, but even the simplest of applications use at least three. At the top of the hierarchy is the *Bootstrap* class loader which loads all the critical runtime classes needed to launch the virtual machine (and, ironically, usually implemented in C) (Horstmann and Cornell, 2000). Below that is the *Extension* class loader which is responsible for loading classes from installed optional packages or from jar files located in the ext directory in the JRE home library. Last is the *System* or *Application* class loader, which is responsible for loading the classes from the system classpath, all the libraries specific to the application being run. It's this class loader which can be utilized to allow a platform to load agents when the class files do not exist in that JVM.

Once a class is loaded by a classloader, any class called for by the related object automatically looks to the same class loader as its source for any additional classes needed. Using this feature, a custom class loader can be created to describe where the class loader can look to find the classes it needs. To do this, a class is created which extends ClassLoader, the abstract base class. Then, only two methods need to be overridden. The first is loadClass(String className, boolean resolve). By default, each class loader takes the class name and calls the same method of its parent class to try and find it there first. This means the bootstrap class loader looks through its classpaths first. If it cannot find it, the extension class path searches through its class paths and, if still not found, falls back to the application class loader. If it cannot find the class, it will throw a ClassNotFound exception. If it does, the resolve boolean tells it to find any classes referenced by the current class in question and load those as well. In this customized version of the class loader, it will still call the same method on its parent, but if not found in any of the of the parent class loaders, it will instead call an overridden version of its findClass() method.

LISTING 3.7: The MyClassLoader Class

```
public class MyClassLoader extends ClassLoader{

    String agentName;
    PlatformLocation classSource;
    Client sender;

    public MyClassLoader(String an, PlatformLocation loc){
        sender = new Client();
        classSource = loc;
    }

    @Override
    public Class findClass(String name) throws ClassNotFoundException{
        byte[]    classFile = getMissingClass(name);
        if(classFile != null){
            return defineClass(name, classFile, 0, classFile.length);
        } else{
            throw new ClassNotFoundException(name);
        }
    }

    @Override
    Protected Class<?> loadClass(String className, boolean resolve)
                                                    throws ClassNotFoundException {
        // check whether loader already has class
        try{
            return super.loadClass(className, resolve);
        } catch(ClassNotFoundException e) {
            e.printStackTrace();
            return this.findClass(className);
        }
    }

    private byte[] getMissingClass(String name){
```

```
        System.out.println("In getMissingClass(" + name + " ) ");
        Telegram t = new Telegram(classSource);
        t.setCommand("class");
        t.setPayload(name);
        return sender.getClassFile(t);
    }
}
```

To load an instance of a class, the class loader needs to call its internal defineClass(String name, byte[] b, int off, int len) method. The overridden findClass() method of the custom class loader needs to therefore acquire the class needed as a byte array to pass to the define() method. To accomplish this, the custom class loader has been given attributes which represent the name of an agent, its class file source location, and a client through which it can contact the class file source location it needs to request the needed class file from. These attributes are linked when the instance of the class loader is created.

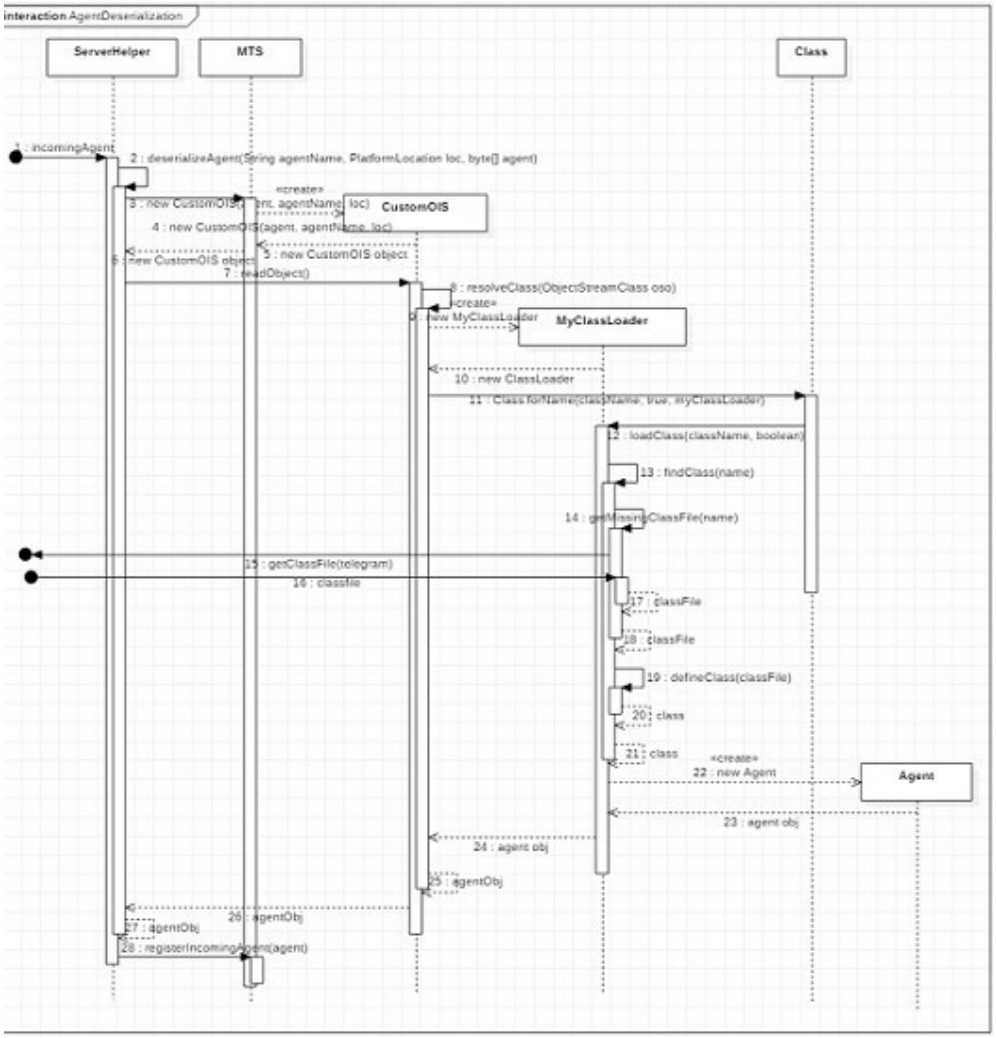

FIGURE 3.3: NOMAD: Deserialization Process

When a Telegram arrives at a destination platform and it includes a MigrationPod,

the server will assign a ServerHelper to handle the incoming transmission. The Telegraph is deserialized, its command string read, and is then forwarded to the appropriate handling block. Inside the block, the MigrationPod is extracted and deserialized. Note that the agent is still in its byte array form inside the MigrationPod object. The server now needs to deserialize the agent byte array using an instance of the custom class loader. But how can the class loader be linked to this process?

The answer lies with the ObjectInputStream class. The deserialization process is initiated by this class when its readObject() method is called on the input stream. As the input stream is read in, the ObjectInputStream instance detects the class name of the object it's attempting to deserialize, called a class declaration, and will call its resolveClass(ObjectStreamClass v) method in order to find an instance of the class to construct the corresponding object. This is the method that must be overridden in a custom class which extends ObjectInputStream so that the custom class loader can be used to find the class instead of the default application class loader.

LISTING 3.8: ServerHelper: run()

```
public void run ( ) {
   this.logOutput(myName + " started, new client connected");
   try{
      ObjectOutputStream oos =
                        new ObjectOutputStream(socket.getOutputStream());
      ObjectInputStream ois =
                        new ObjectInputStream(socket.getInputStream());

      Telegram t = null;
      try{
         t = (Telegram)ois.readObject();
      } catch(ClassNotFoundException e) {
         e.printStackTrace();
      }
      String cmd = t.getCommand();
      if(cmd != null){
         this.logOutput("received command: " + cmd);

         if(cmd.equalsIgnoreCase("agent")){
            MigrationPod mp = (MigrationPod)t.getPayload();
            Agent a = deserializeAgent(
                        mp.getAgentName(), mp.getClassScource(),
                                                mp.getAgent());
            this.logOutput("agent initialized successfully");
            mts.registerIncomingAgent(a);
            this.socket.close();
            this.interrupt();
      . . .
      // other command handling blocks
      . . .
}
```

NOMAD implements a class called CustomOIS that extends ObjectInputStream as an inner class of the MTS. The ServerHelper constructs an ObjectInputStream

object from this custom class, passing in the agent's byte array, name, and *classFileSource* object as parameters. When the readObject() method is called on the ObjectInputStream object, it reads in the class name of the object being deserialized and calls its internal resolveClass(ObjectStreamClass v) method. This method, which is overridden in the CustomOIS class, constructs an instance of the custom class loader, passing in the agent's name and *classFileSource* object. The static method forName(String name, boolean initialize, ClassLoader loader) is then called, passing in the name of the class it needs to load and the custom class loader as parameters. The custom class loader can now find the class at the agent's class file source location, returning an instance of it which resolves the deserialization, allowing the agent to be created.

LISTING 3.9: The MTS.CustomOIS Class

```
// inner class
public class CustomOIS extends ObjectInputStream {

   String agentName;
   PlatformLocation classSource;

   public CustomOIS() throws IOException {}

   public CustomOIS(InputStream in, String agentName,
                                PlatformLocation loc)throws IOException {
      super(in);
      this.agentName = agentName;
      this.classSource = loc ;
   }

   @Override
   protected Class resolveClass(ObjectStreamClass oso)
                          throws IOException , ClassNotFoundException {

       ClassLoader myCL = (ClassLoader) classSites.get(agentName);
       if(myCL == null){
         classSites.put(agentName, new MyClassLoader(agentName, classSource));
         myCL = (ClassLoader)classSites.get(agentName);
       }

      Class c = null;
      try{
          System.out.println("Looking for class " + oso.getName());
          c = Class.forName(oso.getName(), true, myCL);
      } catch(ClassNotFoundException ex){
          c = (Class)primitiveJavaClasses.get(oso.getName());
          if(c == null){
              System.out.println("Cannot locate " + oso.getName());
              throw ex;
          }
      }
      return c;
```

```
    }
}
```

When the custom class loader is first created for an incoming agent, it's kept in a table with the agent's name as a key. This is so that any subsequent classes the readObject() method discovers in the serialized byte stream also needing to be resolved, it will not cause new class loaders to be created but instead always refer to the first instance created, keeping the agent's class resources all within the same class loader. The agent's personal class loader may also be needed later too if the agent ever calls an internal method that references a class which has not yet been loaded into the local classpath as it runs tasks in the platform.

As a side note on security, additional features can also be implemented in the custom class loader with respect to controlling what classes can be loaded in the JVM. In any case where executable code is moved over a network connection and run at its destination, security implications must be considered. As a basic framework, NOMAD does not incorporate any such measures, and the user is encouraged to define policies within the class loader for determining what classes may or may not be loaded. The user can also explore Java's Security Manager Utility, which allows for defining policies about what specific access or operations can be permitted to threads during runtime, or whether remote code is allowed to be executed.

3.5 Communication: Getting the Message Out

As a general delivery system, the same Telegram object used to transmit agents to remote platforms can also be used to send messages to them as well. As the platform network follows a peer-to-peer structure, a little extra work is needed to find agent recipients if they're not local to the message sender. A Message object can be created by an agent and populated with the name of the agent it wishes to communicate with. There is also a generic attribute T which references any object or primitive that also needs to be delivered. This will provide a great amount of flexibility in the content an agent wishes to send to another. However, the user should be careful. When the agent recipient is on the same platform as the sender, anything can be attached to the message as content. But if the agent is remote, anything attached to the content slot must implement the Serializable interface even though the Message class itself already does. Otherwise, a NotSerializableException will be thrown while attempting to transmit the message and delivery will fail.

The agent needs only to call send(Message msg) from its base class. This will prompt a call to the platform which will check the message's recipient field to see if the intended agent is local. If so, the platform will push the message onto a message queue belonging to the agent. As the platform may be pushing a message onto the mailbox at the same time that the agent might be polling one from it, each of these actions are synchronized to prevent any thread errors from occurring. If the intended agent is not local, the message gets passed off to the MTS to find on which platform the agent resides and send the message there.

LISTING 3.10: Platform: sendMessage()

```
public synchronized boolean sendMessage (Message m) {
    if(this.agentMap.containsKey(m.getTo())){
        this.agentMap.get(m.getTo()).mailbox.add(m);
        return true;
    } else{
```

```
      return mts.sendRemoteMessage(m);
   }
}
```

To find the agent, the MTS gets a list of all neighbor platforms and iterates through them, using a client to send a Telegram to each of them with a *"findagent"* command and the name of the intended agent as the payload.

LISTING 3.11: MTS: sendRemoteMessage()

```
public boolean sendRemoteMessage(Message m) {
   Client c = new Client();
   Iterator<PlatformID > iter =
                        this.platform.getAllNeighborPlatforms().iterator();
    PlatformID winner = null;
    while(iter.hasNext() && winner == null) {
        PlatformID candidate = iter.next();
       winner = c.findAgent(m.getTo(), candidate);
   }
   if(winner != null){
      return c.sendMessage(m, winner);
   } else{
      return false;
   }
}
```

In the ServerHelper of the destination platform, the name is used and, if the agent is found, returns its PlatformID as a confirmation (or returns a payload of null otherwise). The client checks if the return payload has a PlatformID and returns it to the MTS.

LISTING 3.12: Client: findAgent()

```
public PlatformID findAgent(String name, PlatformID loc){
    Telegram t = new Telegram (loc.getlocation(),
                           mts.getMyPlatformID().getlocation());
   t.setCommand ("findagent");
   t.setPayload(name);
   Socket cSocket;
    Telegram ct = null;
   try{
       cSocket = new Socket(t.getRecipientIP(), t.getRecipientPort());
        OutputStream os = cSocket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream (os);
       oos.writeObject(t);
       oos.flush();
       InputStream is = cSocket.getInputStream();
        ObjectInputStream ois = new ObjectInputStream(is);
       ct = (Telegram)ois.readObject();
       oos.close();
       ois.close();
       cSocket.close();
   } catch (UnknownHostException e){
      e.printStackTrace();
```

```
    } catch(IOException e){
        e.printStackTrace();
    } catch(ClassNotFoundException e){
        e.printStackTrace();
    }
    return(PlatformID)ct.getPayload();
}
```

LISTING 3.13: ServerHelper: run()

```
public void run ( ) {
    . . .
    // Telegram deserialized
    . . .
    String cmd = t.getCommand();
    . . .
    // other command blocks
    else if(cmd.equalsIgnoreCase("findagent")){
        // if agent name is local, return myPlatformID
        // otherwise return null
        String name = (String)t.getPayload();
        Telegram response = t.reply();
        if(mts.isLocalAgent(name)){
            response.setPayload(mts.getMyPlatformID());
        } else{
            response.setPayload(null);
        }
        oos.writeObject(response);
        oos.flush();
        this.socket.close();
        this.interrupt();
    }
    . . . // e l s e
}
```

The MTS then has the client deliver the message to the destination.

LISTING 3.14: Client: sendMessage()

```
Public boolean sendMessage(Message m, PlatformID loc) {
    Telegram t =
        new Telegram(loc.getlocation(), mts.getMyPlatformID().getlocation());
    t.setCommand("message");
    t.setPayload(m);
    Socket cSocket;
    Telegram ct = null;
    try{
        cSocket = new Socket(t.getRecipientIP(), t.getRecipientPort());
        OutputStream os = cSocket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(os);
        oos.writeObject(t);
        oos.flush();
        cSocket.close();
        return true;
```

```
  } catch(UnknownHostException e){
      return false;
  } catch(IOException e) {
      // possible that payload contains a
      // non-serializable object or object
      // with non- serializable fields/references
      return false;
  }
}
```

Once the message has arrived, the ServerHelper extracts the message and passes it to its MTS, who then forwards it to the platform for final delivery. Once received, the agent is free to handle the content within the message as needed. A *command* String is included in the Message object. In the same way the server helper used the "agent" command in the Telegram's command slot to determine how to handle it, so too can the agent use the Message command slot to know what content to expect and how to use it, if necessary.

3.6 Future Work

Though only a basic framework, NOMAD contains all of the features needed to enable powerful mobile agent applications fit for agent modeling, distributed computing, message passing, file sharing, and more. In spite of this, the ways in which the framework can be extended and expanded are almost limitless. Depending on the needs of the user though, there are certainly different priorities in how the framework's existing features can be further developed to be more robust and fault-tolerant, or with new features to be added to expand its capability. There are a few improvements, however, that may have immediate benefit to users regardless of the domain to which NOMAD may be intended.

The first improvement would be the addition of a GUI or other type of interface to enable the user to perform essential functions on the platform. This would provide a simple means to launch new platforms and agents, send messages, close down agents and platforms, and other basic functions. This could even be implemented as a dedicated agent which launches by default when starting the application. Another useful feature would be a mechanism to designate a platform as a master node in the platform network. Organizing the network into such a hierarchy with a root master could make certain tasks, like directory services and information queries, more simplified if those resources were consolidated at a master node. Recovery policies could be implemented to move the services to another platform in the event that the master node somehow fails. Increased security as well, particularly with code mobility, would also certainly benefit the framework regardless of implementation.

For any other new features to be added, it's important to consider that the original purpose of NOMAD is to be extremely lightweight. Therefore, new features should be implemented in a modular way which can be added or removed easily as needed, such as through launch arguments, or a preferences file similar to the one NOMAD currently uses to specify which port to run the server on.

3.7 Conclusion

In an age where distributed computing is extremely widespread and common, Agent-based computing in its many forms stands as a very powerful model that provides flexibility, autonomy through its agents, and versatility. From the simple view of the agent as an autonomous entity that provides the abstract principle of agency, or acting on behalf of the user or software, comes a broad base of architectures and implementations used in a vast number of domains in business, science, and engineering. These implementations sometimes conform to established standards that extend their capabilities and interaction with other agent-based systems. How these implementations are designed have an impact on where their strengths lie in areas of planning, cognition, behavior, reaction, mobility, sociability, efficiency, security, and fault-tolerance. As new technologies develop, new opportunities for using agent-based systems are discovered, driving the need for using the agent paradigm in new ways.

By understanding the core principals and technologies of agent-based systems, particularly agent mobility, the user can take better advantage of the implementations of such systems in developing applications with them. Having a simple framework from which to begin experimenting with implementing agent systems, and building onto that framework to expand its capabilities as needed, NOMAD hopes to offer the user the advantage of more quickly mastering the concepts needed to bring agent-based computing into the next decade of advancement.

Bibliography

Agents, IEEE Foundation For Intelligent Physical (2018). *The Foundation for Intelligent Physical Agents*. http://www.fipa.org/index.html.

Aguilar, J. et al. (2001). "Application of the Agents Reference Model for Intelligent Distributed Control Systems". In: *Advances in Systems Sciences Measurement. WSES Electrical and Computer Engineering Series*, pp. 204–210. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.416.2595&rep=rep1&type=pdf.

Bellifemine, Fabio, Giovanni Caire, and Dominic Greenwood (2007). *Developing Multi-Agent Systems with Jade*. John Wiley and Sons, LTD., West Sussex, England. ISBN: 978-0-470-05747-6.

Berners-Lee, Tim et al. (2004). "Architecture of the World Wide Web, Volume One". In: URL: https://www.w3.org/TR/webarch/.

Booth, David et al. (2004). "Web Services Architecture". In: URL: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

Brooks, R.A. (1986). "A Robust Layered Control System for a mobile Robot". In: *IEEE Journal of Robotics and Automation* 2.1, pp. 14–23.

Cabri, Giacomo, Letizia Leonardi, and Franco Zambonelli (2018). "Weak and Strong Mobility in Mobile Agent Applications". In: URL: https://www.researchgate.net/publication/228604745_Weak_and_strong_mobility_in_mobile_agent_applications.

Caire, Giovanni (2009). "Jade Programming for Beginners". In: URL: http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf.

Chong, Hui-Qing, Ah-Hwee Tan, and Gee-Wah Ng (2007). "Integrated cognitive architectures: A survey". In: *Artificial Intelligence Review*, pp. 103–130. URL: https://www.researchgate.net/publication/225257926_Integrated_cognitive_architectures_A_survey.

Epstein, Joshua M. and Robert Axtell (1996). *Growing Artificial Societies: Social Science from the Bottom Up*. Brookings Institution Press. ISBN: 978-0-262-55025-3.

Ferguson, Innes A. (1992). "TouringMachines: an architecture for dynamic, rational, mobile agents". In: *Proceedings of the 11th International Conference on Cognitive Modeling*. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-273.pdf.

Grimshaw, David (2010). "JADE Administration Tutorial". In: URL: http://jade.tilab.com/doc/tutorials/JADEAdmin/JadePlatformTutorial.html.

Haring, Kerstin, Marco Ragni, and Lars Konieczny (2012). "A Cognitive Model of Drivers Attention". In: *Proceedings of the 11th International Conference on Cognitive Modeling*. URL: https://www.researchgate.net/figure/The-organization-of-information-the-cognitive-architecture-ACT-R-Anderson-1993-The_fig2_235801050.

Hebert, Scott (2015). "Beyond Market Mix Models - Enhancing Market Analytics through ABM: A Pharmaceutical Case Study". In: URL: https://www.anylogic.com/upload/conference/2015/presentations/sterlingsimulation.pdf.

Horstmann, Cay S. and Gary Cornell (2000). *Core Java Volume 2 - Advanced Features*. Sun Microsystems Press, Palo Alto, CA. ISBN: 0-13-081934-4.

I., Nunes (2014). "Improving the Design and Modularity of BDI Agents with Capability Relationships". In: *Engineering Multi-Agent Systems*. URL: https://link.springer.com/chapter/10.1007%2F978-3-319-14484-9_4.

Karabey, Isil and Ugur Guven Adar (2014). "Agent-Based E-Commerce and its Contribution to Economy". In: *Proceedings of International Academic Conferences*. URL: https://ideas.repec.org/p/sek/iacpro/0802194.html.

Korpela, Eric et al. (2001). "SETI@HOME: MASSIVELY DISTRIBUTED COMPUTING FOR SETI". In: *COMPUTING IN SCIENCE AND ENGINEERING*, pp. 78–83. URL: https://pdfs.semanticscholar.org/b2d4/071e17ef398059ca46c2e7369cbf3999e13b. pdf.

Lam, Kam-Yiu, Alan Kwan, and Krithi Ramamritham (2002). "RTMonitor: Real- Time Data Monitoring Using Mobile Agent Technologies". In: pp. 1063–1066. URL: https://www.researchgate.net/publication/221310408_RTMonitor_Real-Time_Data_Monitoring_Using_Mobile_Agent_Technologies.

Lange, Danny B. and Mitsuru Oshima (1998). *Programming and Deploying Java Mo- bile Agents with Aglets*. Addison Wesley Longman, Inc., Reading, Massachusetts. ISBN: 0-201-32582-9.

Li, J. and U. Wilensky (2009). "NetLogo Sugarscape 1 Immediate Growback model". In: URL: http://ccl.northwestern.edu/netlogo/models/Sugarscape1ImmediateGrowback.

Luck, Michael, Ronald Ashri, and Mark D'Inverno (2004). *Agent-Based Software Development*. Artech House, Inc., Norwood, Massachusetts. ISBN: 1-58053-605-0.

Moreno, A. and C. Garbay (2003). "Editorial: Software agents in health care". In: *Artificial Intelligence in Medicine*, pp. 229–232. URL: https://dl.acm.org/citation. cfm?id=2232622.

Muller, Jorg P. and Pischel, Markus (1993). "The Agent Architecture InteRRaP: Concept and Application". In: *German Research Center for Artifcial Intelligence DFKI*, URL: file:///C:/Users/Michael/Downloads/RR-93-26.pdf.

Newell, Allen (1990). *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts. ISBN: 9780674921016.

Pearce, Jon (2018). "Agent-Based Modeling in NetLogo". In: *San Jose State Univer- sity*. URL: http://www.cs.sjsu.edu/~pearce/modules/lectures/abs/as/Framework.htm.

Perugini, Don et al. (2003). "Agents in Logistics Planning – Experiences with the Coalition Agents Experiment Project". In: URL: http://aosgrp.com/featured-research/scientific_and_technical_pu/repository_of_workshop_pape/aamas_2003_workshop_w5/.

Schelling, Thomas C. (1969). "Models of Segregation". In: *The American Economic Review* 59.2, pp. 488–493. URL: http://www.jstor.org/stable/1823701.

Sierhuis, M., W.J. Clancey, and R. van Hoof (1999). "BRAHMS: A multi-agent programming language for simulating work practice". In: *RIACS/NASA Ames Re- search Center*. URL: https://brahms.ejenta.com/staticdocs/documentation/ papers/BrahmsWorkingPaper.pdf.

Sycara, Katia (2012). "Retsina - Multi Agent Systems". In: *Carnegie Mellon Univer- sity Robotics Institute Website*. URL: https://www.cs.cmu.edu/~softagents/ retsina.html.

Tisue, Seth and Uri Wilensky (2004). "NetLogo: Design and implementation of a multi-agent modeling environment". In: *Proceedings of the Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence*. URL: https://ccl.northwestern.edu/papers/2013/netlogo-agent2004c.pdf.

Voorde, Benny Van De (2016). "Cisco IT Tetration Deployment, Part 1 of 2". In: URL: https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/white_papers/Cisco_IT_Tetration_Deployment_Part_1_of_2.pdf.

Yu, H., Z. Shen, and C. Leung (2013). "From Internet of Things to Internet of Agents". In: *IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pp. 1054–1057. URL: https://ieeexplore.ieee.org/document/6682193/authors.