Chapter 1

Agent Toolkits

1.1 INTRODUCTION

Agent-based applications need a significant amount of enabling infrastructure before even one message is exchanged between agents. There is a large set of supporting services that must be made available throughout the agent-based system before the application developer can move on to focus on the actual application domain, be it eBusiness, Grid computing or Ambient Environments. These services range from basic communication to discovery, coordination, security, and so on. The sum of these services come together to provide an environment that can support an agent-based system, and as such can be considered as providing the *operating system* for agents. Furthermore, there are issues that cross the bridge between domain-independent infrastructural services and application specific services. These relate to the architecture of individual agents within the environment and specific coordination mechanisms that deal with issues such as negotiations [31, 2] or the creation of agent teams [36] and organisations [37, 20, 46].

Crucially, mainstream development would not be able to practically adopt agent-based approaches if the underlying operating systems had to be re-written each time or if new agent architectures and complex coordination mechanisms designed for every new application. It is widely accepted that if agent technologies would move from research labs to mainstream development, as part of the application developer's set of technologies for designing and developing distributed applications, appropriate toolkits were required that would provide the necessary support for developing agents and deploying the infrastructure required to support agent applications [10, 45]. As a result, the past years have seen significant efforts being undertaken into the development of appropriate infrastructure for agent-based systems as well as debate on what are the appropriate concepts to support the development of agent-based infrastructure [1]. Furthermore, such development efforts often do more than just designing and developing the enabling infrastructure. They also provide the necessary tools to aid in the development of applications that operate using that infrastructure. Such tools range from graphical development environments to management and monitoring services.

We use the term agent toolkits to describe software that provides the software for deploying an agent infrastructure as well as aids in the development of agent applications. Agent toolkits are intended to provide a significant proportion of the basic building blocks required to support an operational agent-based system. Ideally, this should allow the application developer to focus on those issues that are specific to the particular application being developed instead of issues relating to how the concepts of agents and multi-agent systems can find practical realization. Of course, like operating systems development, each toolkit represents the designer's particular beliefs or philosophy about how agent-based systems should operate.

The main aim of this chapter is to compare and contrast some of the most widely used and influential toolkits for agent-based systems development. At the same time, it also aims to illustrate some of the main challenges in developing such toolkits and the variety of methods with which these challenges have been tackled. As is seen from the reviews within this chapter, the richness of the agent paradigm makes it especially hard to strike the right balance between what should be implemented within the toolkits and what must be considered application-specific.

The form of such toolkits is as varied as the large number of toolkits available (Agentlink.org lists more than 100). Some are integrated development environments that provide a graphical interface; others also include networking capabilities providing some form of middleware, while still more are simply sets of APIs (application programming interfaces) that a programmer can integrate into their own solutions. Whatever the case they all necessarily employ some form of agent model and some even prescribe a certain methodological approach.

The chapter begins by outlining a set of criteria that are used to select the toolkits reviewed in this chapter, as well as a generic framework for comparing and contrasting them. Subsequently, each toolkit is presented and the ways it tackles each of the issues identified in the generic agent toolkit is discussed. Six toolkits are investigated into some detail, but there are also brief outlines of several other important toolkits. The chapter concludes with an discussion on the main toolkits drawing some conclusions about the current state of the art and possible future directions.

1.2 REVIEW METHOD

1.2.1 Selection Criteria

The toolkits described here have been selected based on three criteria. Firstly, they should tackle as wide a range of issues as possible in relation to application development in distributed, heterogeneous and dynamic computing environments, and should lead to realistic applications rather than the simulation of application or simplified prototypes. This allows us to touch on as many of the related subjects as possible and provide a wide range of examples of how similar problems are tackled through different approaches. Secondly, they should be well documented, and there should be several examples of their use in significant applications. Examples of the application of toolkits are essential since this is currently the only way to ensure some appropriate feedback on the viability of the toolkit, beyond

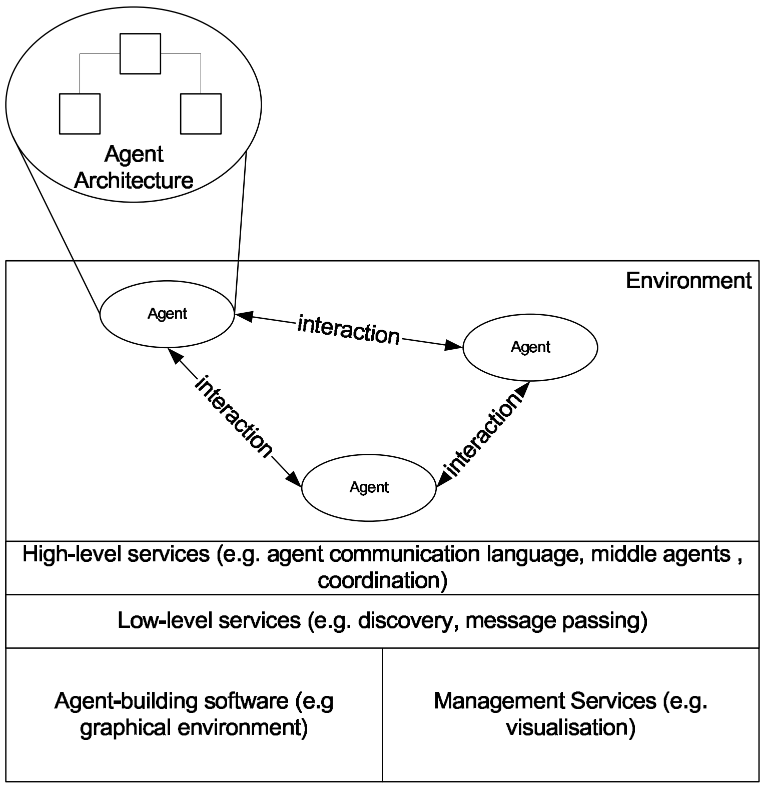

Agent Architecture
Environment
Agent
interaction
Agent
interaction
interaction
Agent
High-level services (e.g. agent communication language, middle agents , coordination)
Low-level services (e.g. discovery, message passing)
Agent-building software (e.g graphical environment)
Management Services (e.g. visualisation)

Figure 1.1 Generic Toolkit Framework

its conceptual structure. Finally, the toolkits should have a significant user community as evidence of their acceptance within the wider field of agent-based systems.

1.2.2 Generic Toolkit Framework

In order to evaluate the various toolkits a consistent method of describing them and eventually comparing and contrasting them is required. Towards this end a *generic toolkit framework* is proposed, which imposes a specific structure within which to place and relate the various issues that must be addressed. This structure is imposed with the caveat that there are a great variety of possible divisions of concerns and one view may favour a specific set of toolkits and disadvantage others. However, for the purposes of an effective comparison, it is necessary to commit to one. A division of concerns as illustrated in Figure 1.1 is employed and detailed below.

The first step is to separate the development of individual agents and their interface to the environment from coordination and communication between multiple agents. For each toolkit, therefore, we need to define how a single agent can be constructed and how that agent can perform actions that will affect its environment. The capabilities that are provided for individual agents are examined at this stage, such as planning or logical reasoning, as well as the specific architectures through which such capabilities are expressed.

When considering multi-agent systems, the focus is on discovery, communication, ontologies and any coordination mechanisms there may be for the agents. Capabilities provided in this context are divided into low-level services (e.g. enabling middleware, basic security functions) and high-level services (e.g. coordination mechanisms, complex security infrastructure). Low-level services are, in essence, generic services that any distributed system infrastructure requires. In this respect some of the important issues are the ability to transfer messages from one agent to another, low-level discovery mechanisms, such as the use of multicasting protocols to discover essential infrastructural services, and security mechanisms for encrypting messages. High-level services are those which are specific to the operation of an agent-based system. At this level the focus is on agent communication languages and protocols to support communication, agents that facilitate the discovery of other agents (usually termed middle agents [14]), ontologies, and coordination mechanisms.

Finally, we investigate the available management services for any resulting applications, relating to the monitoring of the application and debugging, and the software that is specifically aimed at aiding the development process such as an integrated development environment, and so on.

1.3 ZEUS

1.3.1 Background

The ZEUS agent toolkit has been under development since 1997 at BTexact. It is the result of practical experience gained while developing two real world multi-agent systems; one for business process engineering [35] and the other for multimedia information management [42]. At the time of writing, ZEUS is an open source project available under a license similar to the Mozilla public license and is written entirely in Java. A dedicated website holds more information about the toolkit, including manuals and developed examples, as well as links for downloading the toolkit itself [1].

According to the ZEUS philosophy, there are five issues that represent the main infrastructural problems that need to be tackled by an agent toolkit [34].

Information Discovery Information discovery refers to the methods that agents have at their disposal to find out information about other agents. It is usually resolved by providing services similar to the White Pages and Yellow Pages we use to find out addresses and telephone numbers of other

1 http://193.113.209.147/projects/agents/zeus/index.htm

people or companies. In ZEUS, this issue is addressed through what are called utility agents that provide just such services.

Communication For agents to be able to exchange messages, they require a common way of formulating messages. This is something that agent infrastructure should provide through the definition of an appropriate agent communication language. ZEUS uses the FIPA agent communication standard.

Ontology In addition to a common language for formulating messages agents also need common methods for describing their application domain. Exactly which ontologies are used in any situation is an application specific issue. ZEUS aids by providing tools for defining ontologies.

Coordination Although it could be argued that coordination is clearly an application-specific task, ZEUS provides some of the most widely used coordination mechanisms. These can significantly aid the development process if the provided coordination mechanisms are applicable to the application at hand.

Integration with legacy software Agent-based systems are often proposed as ideal solutions for integration of new systems with legacy software. Agents can act as the interlocutor between legacy software and new systems. ZEUS addresses this issue by providing a means for ZEUS agents to interface with external programs.

Beyond these issues, the ZEUS design follows a set of basic guidelines: a clear separation between domain-specific problems and agent-level functionality; a friendly graphical interface for development; an open and extensible design; and strong support for standards and standardized technologies as evident by its compliance with the FIPA standards.

1.3.2 Agents

According to the ZEUS perspective, agents are deliberative, so they reason explicitly about which goals to select and which actions to perform. They are goal-directed, so any action performed is in support of a specific goal. They are versatile, so they can perform a number of goals and engage in more than one task. They are truthful, so when dealing with other agents they always state the true facts. Finally, agents are temporally continuous, so they have a notion of time and can synchronize based on a clock.

Based on this approach, the ZEUS toolkit provides a set of components that represent specific agent functionalities such as planning and scheduling algorithms, agent communication language capabilities (using the FIPA ACL) and communication protocol implementations, ontology support and coordination.

The assembly of these components readily leads to the construction of what is termed a *generic ZEUS agent*, illustrated in Figure 1.2. Agents can send and receive messages, through *Mailbox* and *Message Handler* components. A *Resource Database* component has a list of the resources available

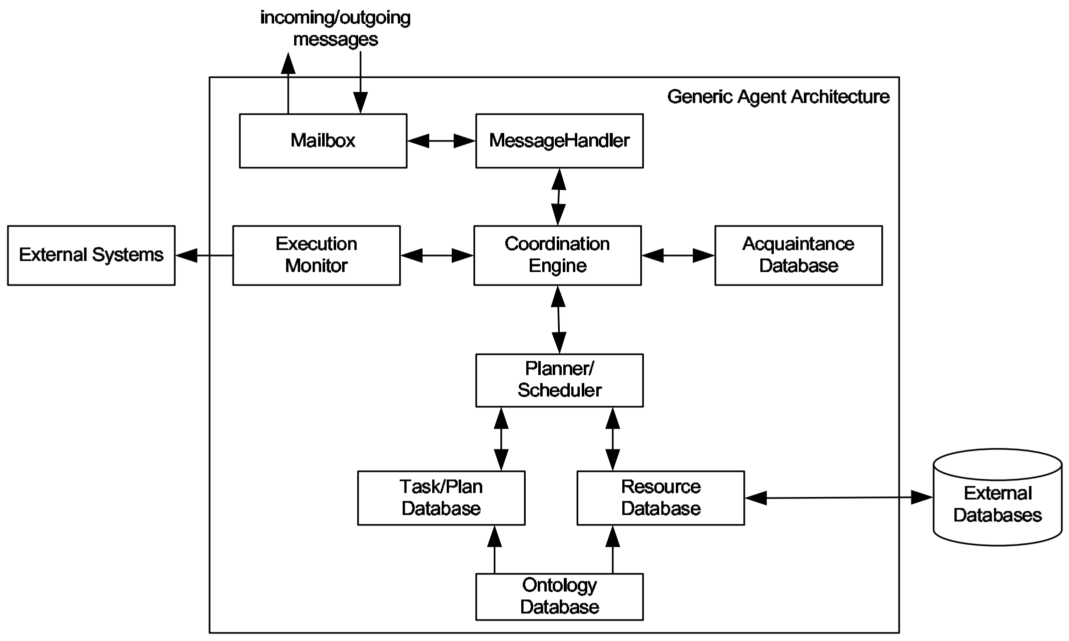


Figure 1.2 Generic Zeus Agent Architecture

to the agent, with the possibility to directly interface with external databases. Through the *Execution Monitor* component, agents can interface with external systems such as legacy systems and also keep track of actions. The *Coordination Engine* component handles the agent's goals, deciding which to follow or abandon. It also handles interaction with other agents, based on the available interaction protocols. Information about other agents, such as name and abilities, is kept in an *Acquaintance Database* component. Finally, the *Planner/Scheduler* component has the task of producing plans and the timings for when actions defined in the plans should be performed in reference to specific goals as requested by the *Coordination Engine*.

This generic agent has all the rudimentary tools necessary to form the base of an agent functioning in a variety of domains. Although it is possible to provide different implementations for these buildings blocks and therefore obtain different types of generic agents, it does not seem possible to deviate significantly from the organizational structure of the inter-component relationships. Nevertheless, since the code for each of these components is provided as part of the overall Zeus package, it is possible to configure them in any manner desired or add or replace existing components. Crucially, the ZEUS development environment assumes this particular configuration for enabling development via a graphical interface and without direct interaction with these components at the code level.

1.3.3 Multi-Agent Systems

1.3.3.1 Low-level

All communication in ZEUS is based on message exchange using the TCP/IP protocol and ASCII messages. This is done to allow for maximum portability of agents. As a result, all services are high-level services that depend on the FIPA ACL and ontologies.

1.3.3.2 High-level Services

Infrastructure support for a multi-agent system in ZEUS revolves around *utility agents*. The term utility agent is used to differentiate between those agents that provide supporting infrastructure and those that perform the actual application tasks, which are called task agents. There are two types of such utility agents, as follows.

The *Agent Name Server* (ANS) maintains a registry of all known agents (or White Pages) and provides a mapping between an agent.s name and its logical network location. It is necessary to have at least one ANS since without it no agent would be able to communicate with another. In larger applications it may be necessary to have a number of ANS agents in order to support all the agents. However, there is always a root ANS agent to bootstrap the system, and other utility or task agents must be provided with the network address of this agent. This root ANS agent also provides a system-wide clock that other agents refer to so that they can synchronize on registration with the ANS. The

Facilitator agent maintains a list of abilities for those agents registered with it (or Yellow Pages). The Facilitator agent is necessary in order to deal with dynamic changes in the capabilities of agents.

The operation of a multi-agent system starts with the registration of each agent with the ANS. Subsequently, agents can retrieve the network address of other agents they wish to communicate with from the ANS. This implies that the agents have prior knowledge of other agents. names and abilities and just need the actual network address. Alternatively, if they do not have this prior knowledge, the application requires a Facilitator, which maintains its information by querying the ANS about registered agents and then queries each agent in turn about its abilities. This approach for discovery of other agents restricts the flexibility of the system to the need for a root ANS agent, so some prior knowledge will always be required. Furthermore, the design of the Facilitator is rudimentary and does not allow for more sophisticated behaviour like the dynamic registration and de-registration of agent capabilities. These are issues that must be handled at the level of the application design, according to the needs of each application.

Agent communication in ZEUS is based on the exchange of FIPA ACL messages. This is supported through specific implementations of the Mailbox and MessageHandler components that can parse such messages and handle the protocols relating to their receipt and transmission. The content of messages is formulated according to the ontologies describing the domain of operation of the agent. Ontologies are supported through the Ontology Database component, which allows developers to equip agents with ontologies that are then used in the formulation of plans and goals and the description of resources.

Coordination is supported through a variety of approaches. The central approach is based on variations of the contract-net protocol [**?**], where there is a Call For Proposals by an Initiator agent followed by replies from Responding agents, and a negotiation phase that can proceed based on a number of strategies. Furthermore, ZEUS allows for the definition of roles, such as peer, subordinate, and superior. Through the definition of roles, multi-agent systems can be given an organizational structure that can aid coordination between agents. Finally, ZEUS allows for multi-agent planning by enabling each agent to factor into its planning responses tasks that depend on other agents.

1.3.4 Agent-building software

Perhaps the main strength of ZEUS is the availability of a graphical interface that allows for the development of an entire multi-agent system application with almost no need to code anything except the interfaces to external systems. Furthermore, this development environment also suggests a certain method for the development of applications.

Development begins with the definition or import of the ontologies that are to be used in the application. An *Ontology Editor* is provided for this. Then, through the *ZEUS Agent Editor*, each task agent is configured by defining planning parameters, tasks, available resources, acquaintances, roles and interaction protocols. Agents are then linked to external programs or resources, such as databases and legacy software. Finally, the utility agents are configured. At this stage, code generation for each agent can take place and the agents can be distributed on the platforms from which they will operate.

1.3.5 Management Services

ZEUS enables the monitoring and control of a multi-agent system through a variety of perspectives, using utility agents that interrogate other agents about their operation and then collate and present the information in an appropriate manner. The Society Tool provides visual information about the exchange of messages between agents, the Report Tool shows the progress on the main tasks and execution state of each sub-task, the Agent Viewer allows the monitoring of the internal state of each agent, the Control Tool allows this state to be altered and the Statistic Tools collects statistics on individual agents and the society as a whole.

The sum of these services provides a powerful tool for the debugging of applications. However, by its nature, it creates a significant amount of traffic within a system and places resource demands on each individual agent. Furthermore, certain types of information, such as the internal state of each agent, may not be available at all in an environment in which agents come from, or represent, different organisations. These services, therefore, should be considered as viable in settings where the multi-agent system is relatively closed, where security concerns are low and where the number of agents is not too large.

1.4 RETSINA

1.4.1 Background

RETSINA (Reusable Environment for Task Structured Intelligent Network Agents) is a multi-agent systems toolkit developed over a period of years, and at least since 1995, at the Intelligent Software Agents laboratory of Carnegie Mellon University's Robotic Institute. RETSINA has been used extensively in a range of applications, such as financial portfolio management, e-commerce, and mobile communications. The toolkit, available as the RETSINA Agent Foundation Classes, can operate in Windows, Unix and mobile platforms, and uses a variety of languages (Java, C++, C, Python, Lisp, Perl) that are tailored to the specific environments. However, the main infrastructural components are written in Java. A limited version of RETSINA is freely available for non-commercial use, under license by Carnegie Mellon University [2].

The design of RETSINA is based on two central assumptions about agent applications development [41]. Firstly, multi-agent systems infrastructure should support complex social interactions between agents through the provision of services that are based on predefined conventions on how social interaction will take place. These predefined conventions refer, mainly, to the use of a common communication language, protocols and ontologies. From the perspective of the multi-agent system infrastructure, agents are seen as black boxes, but they are expected to be able to participate in social

2 http://www-2.cs.cmu.edu/ softagents/

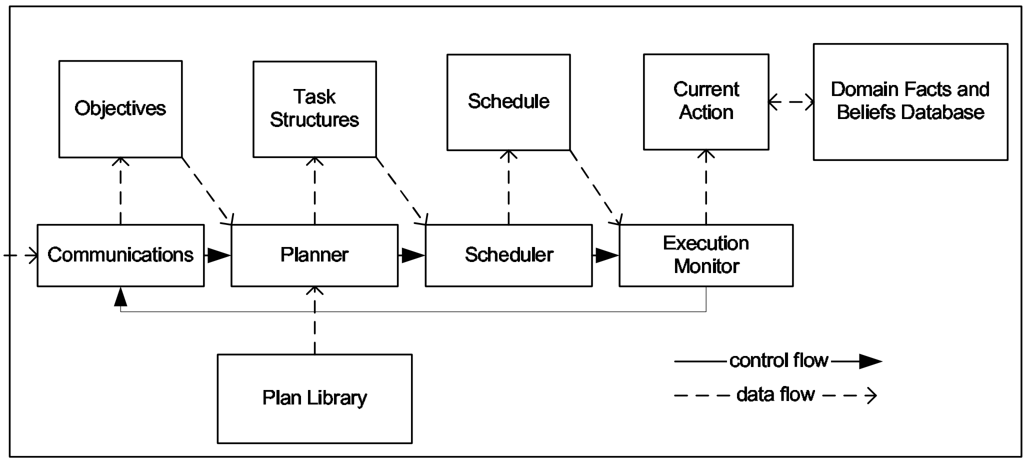


Figure 1.3 Retsina Agent Architecture

interactions based on these conventions. Secondly, agents in a multi-agent system engage in peer-to-peer relationships. Any societal structures, such as hierarchies, should emerge through these peer-to-peer interactions, and should not be imposed by a centralized approach. This is in recognition of the need to sever any ties from centralized control, and allow for truly distributed structures to emerge. These assumptions for multi-agent systems development lead to a very clear separation between individual agents and the supporting infrastructure.

1.4.2 Agents

An agent in RETSINA is understood, in abstract terms, as a standalone survivable piece of code with communicative and intelligent behaviour. In real terms, it is understood as any piece of software that is able to interact with other agents, and with the RETSINA multi-agent system infrastructure, following the conventions defined in RETSINA.

All agents are derived from a *BasicAgent* class, which provides the main functions required for operation in a RETSINA multi-agent system, such as message-handling, logging, visualization, and discovery of other agents. This agent-specific functionality is separated from operation within specific operating environments by placing agents in an *AgentShell*, which provides the necessary interfaces for interaction with the underlying operating system. Furthermore, the AgentShell provides basic management functionalities such as starting up or shutting down the agent and a timer module.

The reasoning and planning for agents is handled by the RETSINA Agent architecture, shown in Figure 1.4. It is based around the interactions between a *Communication* module that handles messages from other agents, a *Planner* that derives plans base on a provided set of goals and a plan library, a *Scheduler* that uses the output from the Planner to schedule when tasks will be performed, and an *Execution Monitor* that handles the actual performance of actions. These modules are supported by appropriate knowledge and beliefs, which are divided into *Objectives*, *Task Structures*, *Schedules*, *Current Actions* and a *Domain Facts and Beliefs Database*.

RETSINA divides agent functionality into four main classes that are built on top of the BasicAgent and represent specializations of the basic architecture to deal with different types of functionalities.

- *Interface Agents* interact with users by receiving inputs and displaying results.
- *Task Agents* carry out the main problem-solving activities by formulating plans and executing them by coordinating and exchanging information with other agents.
- *Information Agents* interact with information sources such as databases or web pages. The Task agents provide the queries, and the information agents are specialized in retrieving the required information by interfacing with databases, the web, and so on.
- *Middle Agents* provide the infrastructural support for the discovery of services between agents.

Although it is possible for developers to provide their own extensions of the BasicAgent class, it is suggested that application development begins from the specialized extensions already provided.

1.4.3 Multi-Agent Systems

1.4.3.1 Low-Level Services

Communication in RETSINA is facilitated by two types of low-level services. Firstly, the RETSINA Communicator module in individual agents enables agent-to-agent communication and abstracts beyond the underlying physical transmission layer and network type. This allows developers to focus on communication at the agent level. Secondly, dynamic discovery of high-level infrastructure services is enabled via the use of a multicast protocol. Once agents enter a multi-agent system application, they multicast their presence and can be detected by high-level infrastructural services that then communicate directly with them. This multicast discovery is based on the Simple Service Discovery Protocol, which was developed as part of the Universal Plug-n-Play [3] ad-hoc networking effort. It is a lightweight protocol that is intended to be used by service providers to announce the availability (or otherwise) of a service, and by service requesters to query for specific services. In RETSINA, a reply to a multicast query is a TCP/IP address and port number that can be used for communication with the discovered service.

The RETSINA multi-agent system provides some basic security services for the authentication of agents, and for the protection of communication between agents. A Certificate Authority system is used for identity protection, through which each agent is guaranteed by a trusted authority.

3 http://www.upnp.org/

Communication between agents is protected through the use of a public/private key system and support for the SSL protocol.

1.4.3.2 High-Level Services

RETSINA views infrastructure as something that should be cleanly separated from multi-agent system applications and individual agent behaviour. As a result there is no support for specific coordination mechanisms, organisational structures or any regulatory policies as this are deemed to be application specific issues. However, although there is no support for specific coordination mechanisms, protocol specification and interpretation are supported through a protocol engine and language that is based on finite I/O automata.

Agents exchange messages that are divided into two parts. Firstly, an envelope defines the sender, receiver, thread of conversation, ontology and ACL used. Within this envelope, content could be specified using any ACL and appropriate ontologies. RETSINA directly supports the KQML ACL, by enabling agents to parse KQML messages, and an ontology derived from the Wordnet Ontology [19]. This functionality is implemented in the BasicAgent.

The basic high-level infrastructural support is provided through *Agent Name Servers*. An ANS maps agent identifiers to logical network addresses. There is also support for multiple name servers and redundant name servers in order to provide robustness and fault redundancy. Each agent is provided, through the BasicAgent class, with an ANS component that enables registration, de-registration and lookup for name servers. Agent name servers can be discovered dynamically through multicast requests. As a result, a multi-agent system can survive without the presence of an ANS, and without the need for prior knowledge of an ANS.

Middle agents provide the second level of infrastructure support. The main type of middle agent is the *Matchmaker*, which provides a mapping between agents and services. This mapping is created through advertisements that matchmakers receive from service provider agents. The Matchmaker then matches a request to service providers and leaves them to handle all subsequent interactions. Both the advertisements of service availability, and the requests for services, are described using a specialized language, called LARKS (Language for Advertisement and Request for Knowledge Sharing) [40]. LARKS is required to provide a standardized description of each service, such as input and output, pre and post-conditions, the context, and a textual description of the service. The result is a KQML message that contains a LARKS advertisement, which uses the appropriate application ontology to describe the available service. The RETSINA toolkit also provides Broker and Blackboard middle agents. Brokers completely hide service providers from the service requestor by mediating all interactions. Blackboard agents simply provide a basic blackboard service where requests are posted for everyone to see, but capabilities are only known by the service providers who can then choose to reply to service requestors directly.

1.4.4 Agent Building Software

The RETSINA Agent Foundation Classes are integrated within the Microsoft VisualStudio development environment. A RETSINA Agent AppWizard is available that provides some basic support for agent development, but there is no step-by-step guidance and the bulk of development involves direct interaction with code.

For debugging, RETSINA provides a useful graphical tool that enables developers to receive compose and send KQML messages to agents in order to test their ability to respond to messages.

1.4.5 Management Services

RETSINA considers management as an issue that should be actively supported through the multi-agent system infrastructure. For this purposes it provides three types of management. The *Logger* is a service that is able to record the main state transitions between agents for inspection by developers. Agents provide this information through the Logger module that is implemented in the BasicAgent class. This logging service can be connected to an *ActivityVisualizer*, which provides a graphical representation of the activity in a RETSINA application. Finally, a *Launcher* service is provided that can coordinate the configuration and start-up of infrastructural components and agents on diverse machines, platforms and operating systems from a single control point.

A graphical tool is available specifically for managing Agent Name Servers, which allows the direct inspection of the information currently registered with an ANS and the configuration of the ANS itself.

As mentioned earlier, these tools are only effective in what are very controlled situations, where all agents fall under the same organizational domain, and where there are no issues concerning the misuse of information on the state of agents.

1.5 IMPACT

1.5.1 Background

IMPACT (Interactive Maryland Platform for Agents Acting Together) is a joint research project between the University of Maryland in the USA, Bar-Ilan University in Israel, the University of Koblenz-Landau in Germany, the University of Vienna in Austria, and the University of Milan in Italy. IMPACT has been used extensively in military applications, such as in the visualization and analysis of army logistics operations, the simulation of combat complex combat situations and the provision of support for controlled flight. The development environment and the core of the infrastructural components are written in Java. At the time of writing IMPACT was not available for use outside the project developers, however more information , including user manuals, can be found online [4].

4 http://www.cs.umd.edu/projects/impact/

The view of what constitutes appropriate infrastructure support and software agent development is illustrated through 10 desiderata that the IMPACT project aims to meet [39].

- It should always be possible to agentize non-agent programs.
- The methods in which data is stored should be versatile in recognition of the current diversity in data storage mechanisms.
- The theory of agents should be independent from the specific actions any agent may perform. Such actions are a parameter of the agent.
- The decision-making mechanisms of each agent should be clearly articulated in order to enable modification at any point of an agent's life.
- It should be possible to reason about beliefs, uncertainty and time.
- Security mechanisms are critical to protect the infrastructure from malicious agents, and to protect agents from other agents assuming false identities.
- There should be some method of providing guarantees as to the performance of agents.
- A theory of agents needs to be accompanied by an efficient implementation and should be such as to allow for an efficient implementation.
- Infrastructure reliability is paramount.
- Testing a theory through practical applications is essential.

1.5.2 Agents

Agents in IMPACT are divided into two parts:

- the software code, which consists of data types and functions that can manipulate those data types; and
- the wrapper, which provides the actual intelligent agent functionality.

The software code could be any software program, and represents the actual interface to the environment through which the agent effects change in it. The wrapper represents the actual agent functionality that is able to manipulate the software code according to the behaviour dictated by the wrapper's programming. This division is the IMPACT solution to the requirement for being able to agentify any program through a wrapper.

The wrapper is further divided into a set of basic components that come together to provide the IMPACT agent architecture, illustrated in Figure 1.4. All actions are regulated by the *Agent Program* that specifies which actions an agent should or should not perform in specific situations; the Agent Program defines what IMPACT terms the agent's *Operating Principles*. The Agent Program itself is

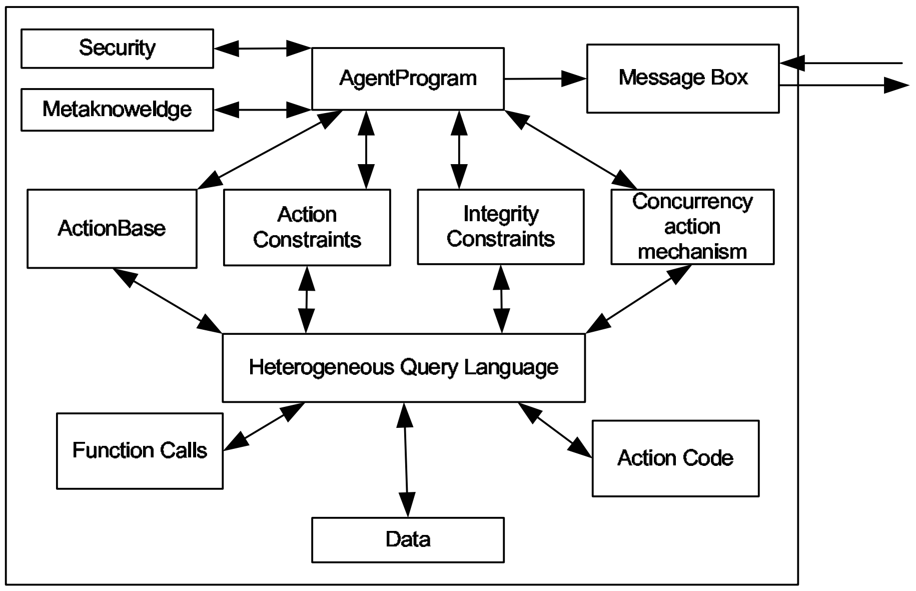


Figure 1.4 IMPACT Agent Architecture

defined according to an *Agent Program Language* that allows for a wide set of regulatory modalities (Do, Obliged, Forbidden, Waived and Permitted). An *Action Base* component holds descriptions of all the actions an agent can perform along with the preconditions for the execution of actions.

It is important to stress that IMPACT takes a wider view of what represents an action than many others. Everything an agent does, including tasks that are traditionally taken for granted or considered an integral part of the architecture, such as planning or timing, are considered actions that must be explicitly defined within the Action Base. Actions can be performed concurrently, and are regulated by a *Concurrent Action Mechanism* component that decides, based on the current agent state and desired actions, whether a composite action can be defined that will achieve the desired actions. Concurrency is also regulated by a set of *Action Constraints* that explicitly define when certain actions cannot be performed concurrently. A set of *Integrity Constraints* specify which agent states are legal in a given context and ensure that the agent does not perform any actions that may violate these constraints. A *Heterogeneous Query Language* component provides the interface with the software code part of the agent. Finally, an agent is equipped with *Metaknowledge* that includes descriptions of what services the agent is able to provide, and beliefs about other agents, and a *Message Box* component that handles communication with other agents.

The most interesting feature of the IMPACT agent architecture, which clearly distinguishes it from other architectures, is the emphasis on ensuring that the agent operates within very well defined parameters. The agent architecture clearly stipulates what actions are allowed, integrity constraints, action constraints, and so on. This provides a multi-layered solution to the problem of being able to guarantee “correct” behaviour. Furthermore, the development process of agents in IMPACT also includes several consistency checks that ensure there are no conflicting rules, such as both forbidding and permitting an agent to do something. We will not elaborate the details of these consistency checks here, but the interested reader can refer to the extensive articles on IMPACT elsewhere (eg. [18, 16, 17]).

1.5.3 Multi-Agent Systems

1.5.3.1 Low-level Services

Agents in IMPACT operate within a dedicated platform, called an Agent Roost, which provides network connectivity and manages the agents operating within it. It is written in Java and uses Java Remote Method Invocation (RMI) to communicate with other Agent Roosts, so IMPACT agents can operate from any platform that can handle Java RMI. The Agent Roost handles the incoming and outgoing messages for each agent within it and can *wake* agents when a message arrives so that they can process it.

Communication with systems outside the Agent Roosts is achieved through a generic Connections module, which is then specialized to enable connections to specific systems, such as Oracle servers.

1.5.3.2 High-level Services

Infrastructure support is based on IMPACT Servers, which provide Yellow Pages services, a type service, a thesaurus service and a synchronization module. All agents providing services must register with the IMPACT Server. Services are described based on a standardized HTML-like language. The service specification requires a service name in terms of a verb-noun expression (e.g. $rent : car[Japanese]$), input and output variables, and service attributes (e.g. cost, response time, etc). Only authorized developers can introduce new agents in the system and the process is semi-automatic, since the developer can use a graphical interface to describe the services provided by the agent at the moment of its introduction into the system.

The Yellow Pages service is a matchmaking service that matches service requests to service providers. This matchmaking service is enhanced through a similarity matching algorithm that is able to match a service request to a service provider even if the service request is not defined in the precise terms with which the service provision has been defined. For example, a request for a $car\_purchase$ can be matched to a car_seller service provider. This is achieved by maintaining two term hierarchies within the IMPACT Server, one for nouns and one for verbs, and an agent table. The term hierarchies contain sets of synonyms that can be used to compute the similarity between two terms. The agent table contains, for each service provider, a noun term, a verb term and the agent name. If there is no direct match between the service request and an entry in the agent table, the term hierarchies are used to discover if there is another service that is sufficiently similar to the service request. The term hierarchies can be updated each time a new service type is registered. This approach provides a more robust service to agents since it can anticipate inconsistencies between service descriptions and service requests, and deal with them.

The type and thesaurus services are, in essence, services in support of the Yellow Pages service. The type service allows developers to define relationships between types that can then be used to aid the service discovery process. For example, a $japanese\_car$ type can be defined as a sub-type of car. The thesaurus service allows the matchmaking algorithms to discover that the term car and $automobile$ are synonyms and update the relevant term hierarchy.

The issue of reliable infrastructure is tackled by mirroring IMPACT Servers, so as to ensure that if one server is not available, others can provide essential services. The synchronization module has the task of ensuring that updates in one server are mirrored to other servers.

The issue of communication between agents in IMPACT is not considered as something that should be stadardized at the infrastructure level. The Message Box is intended to parse any message and allow the rest of the agent architecture to handle the message in a standardized way. As a result, it is up to the application developer to provide an appropriate implementation of the Message Box component. There is also no specific support for coordination between agents.

1.5.4 Agent Building Software

The IMPACT toolkit provides an agent development environment, called AgentDE, that allows developers to define every aspect of the agent that forms part of the agent wrapper. The AgentDE can maintain a library of actions, agent programs, service descriptions, and other definitions used during development so that they can be quickly re-called and reused. Various connections to external databases are also defined using the AgentDE. Once an agent has been defined, the AgentDE can perform a number of checks to ensure that the agent fulfils a number of requirements for consistency and safety. It then produces a binary file, called the agent metadata file (a serialized set of Java objects), which must then be copied to the target Agent Roost initialization directory. There it is deserialized by the Agent Roost and placed into operation. A more automated process, where the AgentDE can directly communicate with active Agent Roosts and transfer the agent metadata file over a network connection is under development.

1.5.5 Management Software

Management in IMPACT revolves around managing Agent Roosts. It is possible to access both through graphical interfaces. The Agent Roost interface allows developers to monitor the state of individual agents in the Roost and the incoming and outgoing messages in the Roost.

1.6 JADE/LEAP

1.6.1 Background

The JADE (Java Agent Development Environment) toolkit provides a FIPA-compliant agent platform and a package to develop Java agents. It is an open-source project distributed by TILab (Telecom Italia Labs) that has been under development since 1999 at TILab and through contributions by its numerous users. At the time of writing, version 3.01b1 is available, and it implements the FIPA2000 specifications. The platform has undergone successful interoperability tests for compliance with the FIPA specifications.

LEAP (Lightweight Extensible Agent Platform) is the result of a research project funded by the European Union and undertaken by a consortium of organisations coordinated by Motorola and including Broadcom, BT, TILab, Siemens, ADAC, and the University of Parma. The aim of the project was to provide an agent platform that is suitable for limited capability devices, such as PDAs and mobile phones.

The relationship between the two projects is that LEAP is a lightweight implementation of the core functionalities of the JADE FIPA platform, and can be used in conjunction with the JADE libraries for agent development. The latest release of JADE integrates LEAP so as to provide a unique toolkit that enables the development of FIPA-compliant agent applications on devices ranging from limited capability mobile devices to desktop computers.

The JADE toolkit has been widely adopted throughout the world, and there is an active community that contributes to its development and offers additional tools. Some examples of applications involving JADE are the development a multi-agent information system supporting the consultation of a corporate memory based on XML technology, communicating agents for dynamic user profiling, collective information dissemination and memory management, and agent-based health care services.

In a relatively recent development a Jade Board has been established, governed by Telecom Italia Labs and Motorola, which is open to all companies and organisations that have an interest in using and further developing JADE. More information about the Jade Board, additional documentation, and links to downloading the JADE toolkit can be found online [5].

1.6.2 Agents

The JADE toolkit facilitates the development of agents that can participate in FIPA-compliant multi-agent systems. It does not define any specific agent architectures but provides a basic set of functionalities that are regarded as essential for an autonomous agent architecture [6]. These are derived by interpreting the minimum concrete programming requirements for satisfying the characteristics of autonomy and sociality. Autonomy is interpreted as an implementation of agents as active objects (i.e. with their own thread of operation). The requirement for sociality leads to enabling

5 http://sharon.cselt.it/projects/jade/

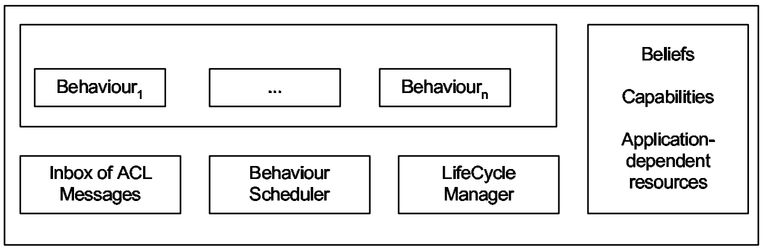


Figure 1.5 Jade Agent Components

agents to hold multiple conversations on a peer-to-peer basis through an asynchronous messaging protocol.

This basic single agent infrastructure is provided through an Agent class, which developers then extend to provide their own implementations of agents. Programs extending the Agent class operate within JADE containers that manage the agent lifecycle. Agents can be started, stopped, removed, suspended and copied. Furthermore, each agent has access to a private message queue, where messages are stored until the agent chooses to retrieve them, and access to a set of APIs that allows the formulation of FIPA ACL messages. An outline of the main aspects of the agent class are illustrated in Figure 1.5.

Specific agent actions take place through a concurrent task model. Each task, or behaviour as it is termed in JADE, is an extension of the Behaviour class of the JADE toolkit. Each agent has a behaviour task list, and the Agent class provides methods for adding or removing behaviours. Once an agent is placed within a container and set into operation, behaviours are executed based on a round-robin non-pre-emptive scheduling policy. Of course, complex tasks require a more sophisticated scheduling of behaviours as well as the conditional execution of behaviours. JADE provides models that are divided along the lines of Simple behaviours, to address tasks not composed of sub-tasks, and Composite behaviours, to address tasks made up through the composition of several other tasks. There are also cyclic and one shot implementations of Simple behaviours, and parallel, sequential and finite state machine implementations for Composite behaviours. Development is further aided by the provision of specific implementations of Behaviour to handle basic tasks such as receiving or sending messages, and support for the set of interaction protocols defined by FIPA.

The LEAP core for JADE offers a lightweight version of the JADE container that can operate on PDAs. LEAP agents use a device-specific Communicator module, which handles the specific connectivity protocols of the device and network at hand. Agents for limited devices use the same task-based model as JADE agents, within the limitations of the device at hand.

1.6.3 Multi-Agent Systems

1.6.3.1 Low-level Services

Multi-agent systems in JADE can be divided into three first-order components. A JADE Platform is made up of a number of Containers that operate on individual machines. Each Container can have a number of Agents within it. A Platform can be thought of as defining a common application domain, and agents within this platform have access to the same infrastructural services. Containers handle the communication between agents and access to Platform services. Communication within JADE platforms is based on JAVA RMI.

Communication between platforms is based on the FIPA-defined Message Transport Protocol (MTP) over which ACL messages can be sent. The actual implementation of the MTP can vary, and FIPA provides specifications for a number of different technologies. As a result, JADE provides a pluggable MTP framework along with concrete implementations that include an Internet Inter-Orb Protocol (IIOP) and an HHTP implementation.

Agents on mobile devices can communicate within the JADE platform through a gateway workstation that provides a translation of messages coming from limited devices into either Internet InterOrb Protocol (IIOP) or Java RMI.

1.6.3.2 High-level Services

The high-level services offered by JADE follow the FIPA specifications, so we will avoid a long description here as the specifications are covered in another chapter. Each JADE platform has access to an Agent Management System, which manages the platform and supervises access to it as well as providing White Pages services. Yellow Pages services are offered by Directory Facilitators and several can exist within a FIPA platform. JADE provides implementations of the SL-0 content language and Agent Management Ontology that is used by the AMS and DF services to communicate. Finally, FIPA-defined interaction protocols are also supported.

JADE supports the development of user ontologies through a Java package that offers a set of classes providing the common high-level terms for any ontology, such as Action, Result, TruePreposition, etc.

Agents in JADE can take advantage of support for mobility to move between containers. At the time of writing, only inter-platform mobility is supported. Agents can be completely removed from one container and placed in another, or they can be cloned across many containers. Mobility introduces the notion of location and other related issues, so JADE provides a mobility ontology that allows agents to describe such concepts.

1.6.4 Agent building software

JADE is a set of APIs that can be use to deploy an agent platform and develop agents. No software is provided to guide this process. However, there is extensive documentation of the APIs, a detailed Programmers Guide, and a wealth of examples.

1.6.5 Management Services

A significant number of utilities are provided for managing and monitoring the activity of an agent platform. A Remote Monitoring Agent (RMA) provides control of the platform lifecycle and all the registered agents within the platform. It provides a GUI which, amongst other functions, allows access and control to individual agents, such as starting and stopping them and sending custom ACL messages to them. Through the RMA, a separate GUI that allows management of Directory Facilitators can also be launched. A Dummy Agent utility is a graphical tool that enables developers to perform all the main activities any agent can perform (i.e. behave like an agent in the platform). As such, it is a useful debugging tool that can help indicate where communication between agents is not developing in the desired manner. A Sniffer Agent allows the monitoring of messages exchanged between a group of agents. Finally, an Introspector Agent provides information about and control of the lifecycle of a single agent.

1.7 JACK

1.7.1 Background

JACK is an agent development environment produced by the Agent Oriented Software Group, which has its headquarters in Melbourne, Australia. JACK was first released in 1998 and is currently at Version 4.0. It has a wide user base, both in commercial and academic environments. It is commercially available, with special licenses for research purposes. A demonstration version is freely available.

There are two guiding principles underpinning the development of JACK. Firstly, agent-oriented development can be thought of as an extension of object-oriented development. As a result, JACK operates on top of the Java programming language, acting as an extension that provides agent-related concepts. JACK developers compare it to the relationship between C and C++, where the latter is an extension of the former for providing object-oriented concepts. Secondly, agents in JACK are intelligent agents in that they are based on the Belief-Desire-Intention architecture. JACK is also supportive of agent standards can FIPA-compliant systems can be produced using JACK.

The JACK development environment can be divided into three main components. The JACK Agent Language is a superset of the Java language, and introduces new semantical and syntactical features, new base classes, interfaces, and methods to deal with agent-oriented concepts. The JACK

Compiler compiles the JACK Agent language down to pure Java, so that the resulting agents can operate on any Java platform. Finally, the JACK Agent Kernel is the run-time program within which JACK agents operate, and provides the underlying agent functionality that is defined within the JACK Agent Language.

More information about JACK including documentation and access to the toolkit itself can be found online [6].

1.7.2 Agents

Although JACK can support a wide variety of agent architectures, the default architecture, and the one that is clearly supported through appropriate concepts in the JACK Agent Language, is the BDI architecture.

The Agent base class is the central artefact of the JACK Agent language. Through it, developers define beliefs, plans, external and internal events and capabilities. This class is intended to be extended to implement application specific agents. Agents schedule actions, including concurrent actions, using the TaskManager. A timer can provide different notions of time, such as a real-time clock (through the system clock) and a dilated clock that can be fast-forwarded, slowed down or even stopped. Finally, the Agent class provides support for sending and receiving messages.

Beliefs represent the knowledge that an agent possesses about the world. A BeliefSet is a database of beliefs that represents beliefs through a first-order, tuple-based relational model. Although agents can store information outside a BeliefSet, it is recommended that BeliefSets are used, since they can provide logical consistency, automatic update of beliefs based on events, and allow powerful queries on beliefs.

Plans are sequences of actions that agents execute on recording an event. Each plan in JACK corresponds to a single event, and multiple plans can be declared to handle the same event. Reasoning capabilities are provided to aid in outlining the required decision-making for deciding which plan to perform when an event occurs. This reasoning is based on the plan.s relevance to a given situation and permitted context based on the agent’s beliefs.

Events within the agent architecture are divided into: external events (e.g. messages from other agents); internal events, initiated by the agent itself; and motivations. Motivations are described as goals that the agent wants to achieve. Events kick-start action in JACK by activating the required plans that may, in turn, raise other internal events or cause external events.

Capabilities provide means for structuring a set of reasoning elements into a coherent cluster that can be plugged into agents. This enables the creation of libraries of capabilities that the developer can use to provide agents with particular functionality. Capabilities can contain within them the relevant plans, beliefs and events, as well as the final code that will implement the actions required by the plans. Through this notion, JACK promotes a high level of code re-use and the incremental development of agents.

6 http://www.agent-software.com

1.7.3 Multi-Agent Systems

1.7.3.1 Low-level Services

Networking capabilities in JACK are based on UDP over IP, with a thin layer of management on top of that to provide reliable peer-to-peer communications.

1.7.3.2 High-level Services

Agent communication between agents is handled by the JACK Kernel. Agents can exchange messages by specifying the name of the agents they wish to communicate with, assigned at the time of creation, and the JACK Kernel takes care of routing the message to the appropriate agent. If an agent resides on a remote host then, along with the agent name, a portal name must be specified, to indicate to the JACK Kernel the logical network address of the remote host. Finally, a rudimentary Agent Name Server is provided that can provide the required portal name in case it is not known. As mentioned earlier JACK supports interoperability with other FIPA-compliant agent systems, and to this ends the FIPA ACL is supported. However, there is relative flexibility to change communications languages since the MessageEvent objects simply define the message as a string.

JACK provides support for coordination between agents based on Team Oriented Programming. This coordination mechanism views a group of agents as a whole and assigns goals to a team of agents, which must then coordinate their activity to achieve the team goal. In order to enable this, JACK offers a plug-in to the main JACK development environment called SimpleTeam. It does not specify specific team management techniques (e.g. hierarchical) but allows developers to assign roles, specify concurrency constraints and define team plans.

1.7.4 Agent-building software

JACK provides a comprehensive, graphical agent development environment. A high-level design tools allows a multi-agent system application to be designed by defining the agents and relationships between them, in a notation similar to UML. Details of individual agents can also be specified at this level. This design can then be used to generate code outlines. A plan-editor allows plans to be specified as decision diagrams. Along with these high-level tools, there is a component browser that allows developers to specify the actual agent code using the JACK Agent Language. Finally, a plan tracing tool and an agent interaction tool allow developers to visualize the monitoring of an application.

1.7.5 Management Services

An application can be monitored through an Agent Tracing Controller. This graphical tool allows a developer to choose which agents to trace and provides a visual representation of the agents stepping through their plans.

1.8 LIVING MARKETS

1.8.1 Background

The *living markets* toolkit is produced by Living Systems AG, which have their headquarters in Donaueschingen. The company has been developing agent-based solutions since 1996 and their toolkit is being used in a variety of settings including complex trading processes, logistics and distribution, and voice and bandwidth trading and settlement. They have a wide client base and have won several awards including Leading Technology Pioneer as recognised by the World Economic Forum and Best German Internet Company.

The *living markets* toolkit is divided into a base agent server, which handles the application domain independent issues relating to agent-based development, and specific solutions for specific markets (ranging from transportation to intra-enterprise production and deal flow optimisation) are built on top of the agent server. Similarly to JACK, agent-based development in *living markets* is considered as a natural progression from object-oriented techniques to role and goal-oriented programming techniques. As such, the adoption of an agent-based approach to building systems represents a *paradigm shift* in dynamic systems development, rather than simply an alternative pattern of object-oriented development. The base agent server is programmed using Java and the default communication is done through Remote Method Invocation (RMI). However, there is also support for a range of industry standards such as XML, Secure Sockets Layer (SSL), Hypertext Transfer Protocol (HTTP), and the Common Object Request Broker Architecture (CORBA).

Living Systems is primarily a solutions company, so they adapt their toolkit to specific customer needs as opposed to marketing the toolkit directly. As a result it is harder to identify specific features, as we have done with other toolkits as there is a range of features adapted to and developed for specific markets. However, it still represents as interesting show case of what is a *pragmatic* agent system. More information about the company and the toolkit can be found online [7].

1.8.2 Agents

From an abstract level point of view agents in *living markets* are understood as proactive, goal-directed entities able to perform actions and perceive the environment. They have specific domain expertise and may adopt roles. Agents, similarly to RETSINA, are specialised into four generic types according to functionality.

Application agents These are domain specific agents and represent the main core functionality of the system.

Integration agents These agents are dedicated to integrating the rest of the system with existing systems outside of the *living markets* environment.

7 http://www.living-systems.com

Interface agents Interaction with the system by people is handled through the interface agents.

System Agents These are the agents that handle the management of the *living markets* system itself performing tasks such as performance monitoring and load balancing.

At the practical, implementation level agents generally operate within the agent base server, called LARS (living agents runtime system), and communicate by exchanging XML messages. Within LARS servers agents occupy their own thread of operation so multiple agents can operate concurrently. *Remote agents* that operate outside a LARS server (e.g. on a mobile client) are also supported, albeit within the limitations of the environment within which they operate. In reflection of the application of *living markets* in financial domains there is strong support for message encryption using the RSA encryption algorithm [8] and the Blowfish key algorithm [9]. Once decrypted the XML messages are stored in a message box that can then be processed by the agent based on a set of logic rules, the agent's *standard logic* that defines its basic behaviour. This handles application domain independent activities such as requests on the status of the agent or requests for moving to another LARS server. The *living market* agents also have a set of business rules, which define the *business logic*. This is where the logic for dealing with specific business processes in encoded. The business logic interfaces with a *persistence layer* that can allow agents to store or retrieve information from the file systems or databases.

Beyond the distinction between business and standard logic there is relative freedom in developing agent architectures within the *living markets*. The business rules can be coded to access a set of *capabilities* made available by the server that they can use to achieve their specific tasks. Such capabilities can also include components that allow the interface with external applications.

1.8.3 Multi-Agent Systems

1.8.3.1 Low-level Services

The low-level services provided by the *living markets* toolkit are primarily concerned with enabling access to external systems and communication between agents. The LARS servers provide a dedicated communication channel that enables communication between agents within a single server as well as a special message router agent, which is able to route messages to other message routers residing on other LARS servers. When agents reside on the same server messages are Java objects passed by reference between the agents. When communicating externally the default communication method is Java RMI, alhtough a variety of alternative channels, including strings over basic sockets, can be supported.

There is strong support for integration of agents with external systems, either through file transfer or HTTP messages or through application programming interfaces that can be connected

8 RSA (named after its inventors Ron Rivest, Adi Shamir and Leonard Adleman) allows a person to encrypt a message using a *public key* than can only be decrypted by the holder of a *private key*.

9 Blowfish is a fast symmetric block cipher.

to agents. The support for integration extends to Enterprise Java Beans (EJBs) servers through customised beans that link EJB servers to LARS servers.

The *living markets* systems attempts to address the issue of scaling agent systems to deal with potentially hundreds or thousands of agents interacting (a very realistic expectation in a financial environment). To this end, LARS servers are designed so as to take advantage of multi-processor environments and can also be arranged into clusters. In addition, since there is support for mobility of agents between servers, agents can be moved automatically to the right servers to improve performance.

1.8.3.2 High-level Services

The *living markets* toolkit offers a wide range of high-level services for agent applications reflecting the range of application environments it has been used in. For the purposes of this review we focus on the support offered for business-to-business applications, although several of these issues apply to other domains as well.

The toolkit divides the required services into four tiers based on functionality. Firstly, agents need to be able to search for partners in deals, for products or for services. The toolkit supports means for describing this information and making it available to agents. Secondly, service providers and service requests need to be matched. The *living markets* toolkits support a method they term *softmatching*. With this method results on searchers can be returned based on their similarity to the actual request. The level of similarity required can be specified by the agent. Thirdly, the toolkit supports a range of dynamic pricing mechanisms that allow agents to decide on the price for service provision. This mechanisms include English, Dutch, Reverse and Vickrey auctions [10], as well as bilateral and multilateral negotiations. Finally, the last tier deals with the clearing and settlement of deals supporting physical and financial settlements.

Agent communication is based on the FIPA ACL, packaged within XML messages. The message channel within the LARS servers and the message router take care of delivering the message to the appropriate recipient. Finally, there is support for transaction management across platforms and databases.

1.8.4 Agent Building Software

Agent development is supported by an integrated graphical agent development environment, the *living markets* Development Suite. This software allows application developers to visually design *agent scenarios*, which are representations of the main agents in the system and the communication flows between them. For each agent the developed can provide a description of the agent, a list of the main goals and their relative importance and the services the agent is meant to provide. Based on these scenarios agent can be created and business logic defined in detail.

10 A description of different auction mechanisms in the context of agent-mediated eCommerce can be found here [25].

1.8.5 Management Software

Management in a *living markets* system is divided between day-to-day management of entire systems and more detailed management of the agents and the servers.

General management capabilities are provided through a *living markets* Management Console. This application provides a web-based interface that is meant to allow the day-to-day administration of the application. More detail management and control of individual agent is provided through a control centre that allows detailed access to each LARS server and the agents residing on the server. Individual messages can be scrutinized and settings relating to communication infrastructure can be controlled.

1.9 OTHER TOOLKITS

This chapter focuses on and analyses in some detail six significant toolkits for agent-based development. This six can be considered as representative of the range of ideas currently prevalent. However, they are by no means the only ones. In this section we very briefly describe a few more toolkits that have had significant use so as to provide a more comprehensive view of the wide range of propositions available.

agentTool agentTool [30] is a toolkit developed at Kansas State University in direct support of the Mutliagent Systems Engineering methodology [15], also developed at KSU. The methodology specifies seven stages starting from identifying the system goals then applying use cases and deriving roles based on them. Subsequently, agent classes are created, conversations constructed and agent classes assembled. Finally, the overall system deployment takes place. The agentTool software supports the construction and assembly of agent classes and conversations, through graphical tools, that lead to the generation of the actual agent code. The architecture of the multi-agent system and individual agents is supported through a notion of *concurrent tasks*, where each task defines a certain decision-making capability. Task are designed graphically as finite state automata and tasks can integrate both intra-agent and inter-agent relationships.

Agent Factory The Agent Factory [12] is developed at the Practice and Research in Intelligent Systems and Media (PRISM) Laboratory of the University of Dublin. It provides extensive support for development through a graphical environment and a distributed run-time platform that scales from workstations to limited-capability PDAs. There are some FIPA-compliant aspects such Directory Facilitators and FIPA management agents. Development is supported by a structured methodology that leads to the implementation of BDI-type agents, The definition of agents is done through an interpreted programming language based on a formal logic model.

BOND BOND is a FIPA-compliant multi-agent system developed at the University of Central Florida. The main motivating concept behind the BOND agent infrastructure system is the view of agents as active mobile objects with some level of intelligence [7, 8]. Another significant design decision is to enable the dynamic reconfiguration of agents [9], to answer to the dynamically changing requirements placed on agent applications. Agents are build using BOND objects. These objects represent an extension of conventional Java objects through the addition of a unique identifier, dynamic properties, communication support, registration with a local directory, serialization and cloning, multiple inheritance and support for editing via a graphical interface. A BOND agent is viewed as a finite state machine with an agenda to follow (i.e. goals to achieve) based on strategies that are made available to agent. There are two possibilities for the creation of a BOND agent. They can be created statically based on the BOND agent framework APIs or dynamically using what is called the BOND *Blueprint* language. Through this language the various components of a BOND agent can be described and are assembled dynamically via a *bondAgentFactory*. BOND agents can also be serialized back to Blueprint for persistent storage or transfer to other hosts where they can resume operation.

CoABS The CoABS [11] (Co-operating Agent Based Systems) project is funded by DARPA (Defense Advanced Research Projects Agency) and the goal is to build enabling infrastructure that will allow the integration of agent-based systems developed with other toolkits. In order to achieve this it makes use of Jini middleware technology, and offers wrappers for each agent that provide basic infrastructure service in a Jini context such as subscription, security, visualisation and logging.

DECAF The DECAF (Distributed, Environment-Centred Agent Framework) toolkit is developed at the University of Delaware [22]. DECAF focuses on the individual agent architectures rather than the underlying distributed infrastructure, although basic Agent Name Server services are provided. Agents in DECAF can be programmed used a purpose-made DECAF language that allows developers to program agents using coarse grain concepts such as agent actions that abstract away from the more fine-grained JAVA programming languages method calls that implement the functionality. DECAF agents also benefit from carefully thought out planning (based on TAEMS [44] and execution scheduling facilities [21].

Open Agent Architecture The Open Agent Architecture (OOA) [32] is developed at the Artificial Intelligence Center of SRI International. Agent communication and coordination is handled by specialised *Facilitator* agents that can handle the distribution of tasks to other agents, enabling the execution of complex goals, as well as act as global data stores for the agents. Other agents in the system are divided into application agents, interface agents and meta-agents. Meta-agents are similar to the facilitator in that they offer coordination support but whereas facilitators are domain independent meta-agents are domain dependent. Communication between agents is based on an OOA-specific Interagent communication language. Finally, the toolkit comes with a range of useful agents already configured such as a Google agent, that implements the Google APIs for programmatic access and a Wordnet agent.

11 http://coabs.globalinfotek.com/

Sensible Agents Sensible Agents [3] is a toolkit is being developed by the Laboratory of Intelligent Processes and Systems at the University of Texas at Austin. It provides a distributed environment for agent operation and communication. The main focus is on developing agents that are able to aid in decision making in environment with limited resources. There is extensive support for modelling capabilities to provide the appropriate knowledge for agents to reason about the environment and powerful planning capabilities. Agents are able to plan and make decisions on goals and actions priorities and the level of control an agent is allowed to take decision can be adjusted at a global level through appropriate restraining structures ranging from command-driven slave-master relationships to each agent being locally fully autonomous.

SoFAR The Intelligence, Agents and Mutlimedia Group at Southampton University has developed the SoFAR (Southampton Framework for Agent Research) toolkit [33]. The focus of SoFAR is on providing a reliable infrastructure that supports agent communication and discovery in the domain of distributed information management (e.g. handling metadata streams synchronously with multimedia streams over a wide-area network [13]). The main contributions are a robust communications layer that abstracts away the low-level details from the actual agents, extensive support for managing and integrating ontologies into the agent-based system, and registration and subscription of agents to services based on contracts. Agents and ontologies can be specified in XML files, facilitating reuse and abstracting away from programming language issues, which are then processed by tools provided with the framework to generate the appropriate code for agents and supporting ontologies.

The chapter, so far, has not touched on toolkits focused on providing infrastructure for mobile agent systems. There are several such toolkits (e.g. D'Agents [23], Aglets [27], Mole [4], SOMA [5]), but their focus is on enabling mobility and dealing with the inevitable security issues [11, 26, 43] rather than on the wider issues of agent-based systems development. The main aim of mobile agent research is to automate the process of moving code from one computer to another. In traditional software, the decision of whether to migrate is made externally to the code that will eventually migrate while in mobile agents the decision of whether to migrate is contained within the mobile code unit that may eventually migrate. Much has been written about the merits of mobile agents (e.g. [28, 24]), with the most important advantages being locality of reference, a high degree of adaptability and fault-tolerance. Such systems, however, have also faced significant resistance due to the security concerns and doubts about their true advantages. Despite such fears the increase in mobile users is making the issues mobile agent research deals with very relevant. For example, a mobile phone or PDA entering a new office and seeking services for its user is very similar to a mobile agent moving to a new host. Even more complicated is the situation where a user travels to an area where service provision is provided by a different organisation (*roaming*). The concepts of agents and mobility are now coming together to provide solutions for such problems (e.g. [38, 29]. As such it can be expected that future agent-based systems toolkits will have to exploit the lessons learned from mobile agent research and related them to the issues dealt so far.

1.10 DISCUSSION

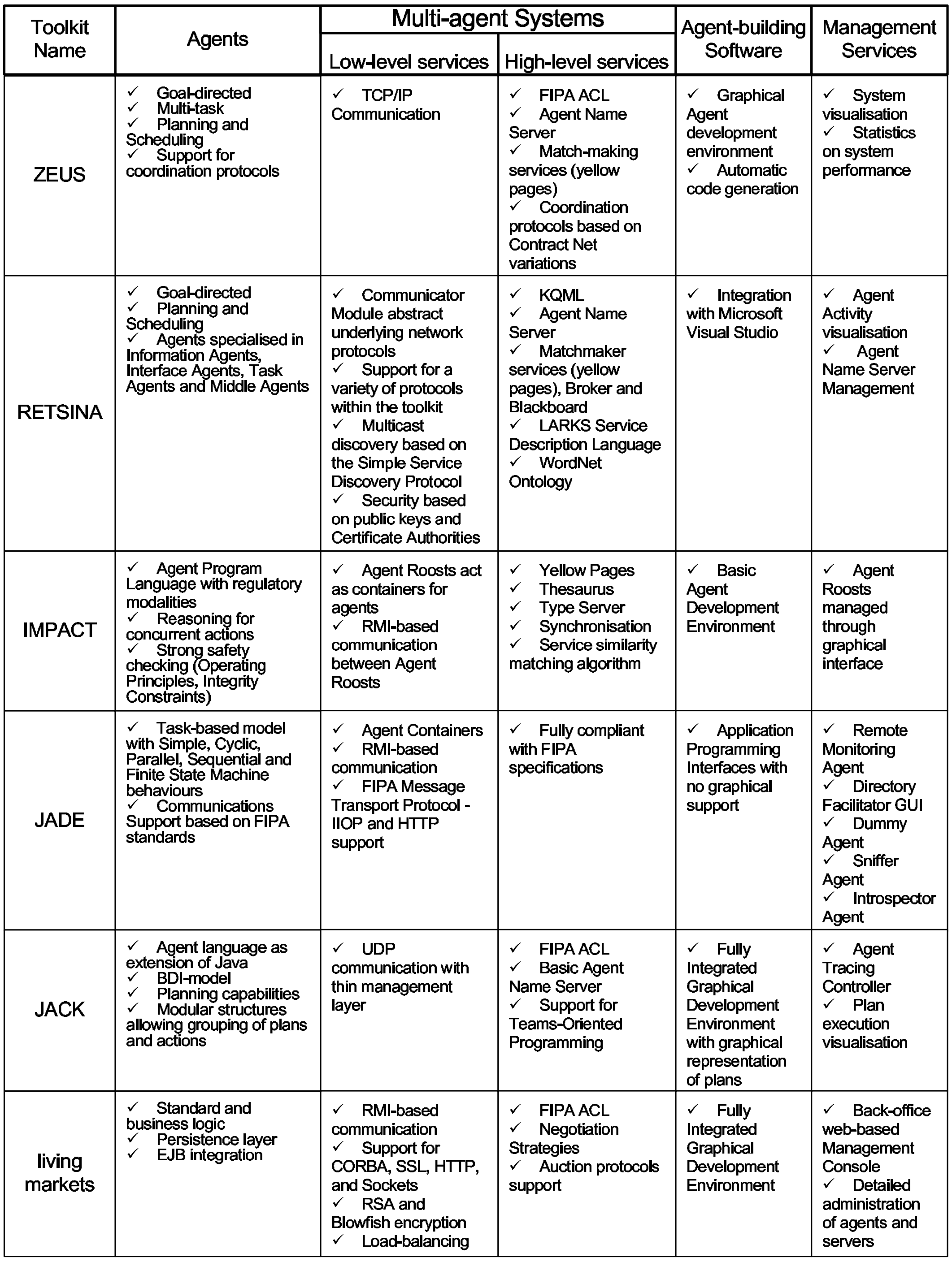
The six toolkits outlined in this chapter form a representative sample of the current state of the art in the field. All have a history of at least three years of development and have been used in a significant number of applications that have proved that agent-based system development can bring true benefits to the appropriate application domains. As such, they can all be described as *successful* attempts at providing an agent toolkit. Nevertheless, few guides exist to aid in ascertaining which of these is better. For example, there is no clear answer to the question of whether a standardized ACL is better than an application specific one in the case where the application being developed is not envisaged to operate in an open environment which any agent can enter. This specific issue is further complicated by the fact that even those toolkits that use a generic ACL, such as the FIPA ACL, may still not effectively support truly open agent systems since other aspects of the infrastructure constrain agents.

This section attempts to draw some links between the six main toolkits reviewed in this chapter, while a summary of their features is provided in Figure 1.6. The different approaches that each toolkit takes for each of the aspects considered are discussed along with some of the relative advantages and disadvantages they offer.

1.10.1 Agents

Of the six toolkits the only one which does not provide significant structure for an agent architecture is JADE. However, the task-based model it supports provides an effective framework upon which to build agent architectures, and it provides some basic pieces of functionality such as handling messages and participating in coordination protocols. Of the other five architectures, three (ZEUS, RETSINA, JACK) are essentially variations of the Belief-Desire-Intention approach, with JACK offering the most “faithful” interpretation of the architecture. It is difficult to argue for which of the three offers the best implementation, since there are relative advantages for each. ZEUS provides a more consistent separation of issues of resources and models of other agents; JACK offers an effective and efficient system for managing beliefs through the BeliefSet; and, finally, RETSINA provides significant supporting infrastructure through the Logger module and powerful scheduling and monitoring capabilities. To a large extent, the choice might depends on how well specific application requirements can translate to the exact definitions provided by each. IMPACT departs from the traditional approaches and places a lot of emphasis on the ability to guarantee that agent behaviour will follow certain constraints. As such, IMPACT can be considered a pioneer from this point of view. At the same time, some analysis would be necessary to verify whether the added complexity introduced in the architecture is worth the guarantees offered for behaviour. The *living markets* approach is perhaps the closest to current industry practise in enterprise application systems since the notion of *business logic* and *standard logic* will resonate with similar notions in more standard technologies such as Enterprise Java Beans.

The only real conclusions that can be drawn is that in order to choose between them a developer should carefully weight the advantages and disadvantages in relationship to the specific application



Toolkit Name	Agents	Multi-agent Systems		Agent-building Software	Management Services
		Low-level services	High-level services		
ZEUS	✓ Goal-directed ✓ Multi-task ✓ Planning and Scheduling ✓ Support for coordination protocols	✓ TCP/IP Communication	✓ FIPA ACL ✓ Agent Name Server ✓ Match-making services (yellow pages) ✓ Coordination protocols based on Contract Net variations	✓ Graphical Agent development environment ✓ Automatic code generation	✓ System visualisation ✓ Statistics on system performance
RETSINA	✓ Goal-directed ✓ Planning and Scheduling ✓ Agents specialised in Information Agents, Interface Agents, Task Agents and Middle Agents	✓ Communicator Module abstract underlying network protocols ✓ Support for a variety of protocols within the toolkit ✓ Multicast discovery based on the Simple Service Discovery Protocol ✓ Security based on public keys and Certificate Authorities	✓ KQML ✓ Agent Name Server ✓ Matchmaker services (yellow pages), Broker and Blackboard ✓ LARKS Service Description Language ✓ WordNet Ontology	✓ Integration with Microsoft Visual Studio	✓ Agent Activity visualisation ✓ Agent Name Server Management
IMPACT	✓ Agent Program Language with regulatory modalities ✓ Reasoning for concurrent actions ✓ Strong safety checking (Operating Principles, Integrity Constraints)	✓ Agent Roosts act as containers for agents ✓ RMI-based communication between Agent Roosts	✓ Yellow Pages ✓ Thesaurus ✓ Type Server ✓ Synchronisation ✓ Service similarity matching algorithm	✓ Basic Agent Development Environment	✓ Agent Roosts managed through graphical interface
JADE	✓ Task-based model with Simple, Cyclic, Parallel, Sequential and Finite State Machine behaviours ✓ Communications Support based on FIPA standards	✓ Agent Containers ✓ RMI-based communication ✓ FIPA Message Transport Protocol - IIOP and HTTP support	✓ Fully compliant with FIPA specifications	✓ Application Programming Interfaces with no graphical support	✓ Remote Monitoring Agent ✓ Directory Facilitator GUI ✓ Dummy Agent ✓ Sniffer Agent ✓ Introspector Agent
JACK	✓ Agent language as extension of Java ✓ BDI-model ✓ Planning capabilities ✓ Modular structures allowing grouping of plans and actions	✓ UDP communication with thin management layer	✓ FIPA ACL ✓ Basic Agent Name Server ✓ Support for Teams-Oriented Programming	✓ Fully Integrated Graphical Development Environment with graphical representation of plans	✓ Agent Tracing Controller ✓ Plan execution visualisation
living markets	✓ Standard and business logic ✓ Persistence layer ✓ EJB integration	✓ RMI-based communication ✓ Support for CORBA, SSL, HTTP, and Sockets ✓ RSA and Blowfish encryption ✓ Load-balancing	✓ FIPA ACL ✓ Negotiation Strategies ✓ Auction protocols support	✓ Fully Integrated Graphical Development Environment	✓ Back-office web-based Management Console ✓ Detailed administration of agents and servers

Figure 1.6 Review of features for toolkits

being developed and the expertise available to the developer. For example, if one is not familiar with the BDI approach or artificial intelligence planning techniques the effort in learning that may not be worth the more direct results achievable through a different approach. It is also worth noting that in domains where there are limited capability devices the more lightweight architectures of JADE and *living markets* could prove much more suitable.

1.10.2 Multi-Agent Systems

1.10.2.1 Low-Level Services

Of the systems reviewed, ZEUS and JACK offer the most lightweight solutions, using the TCP and UDP protocols respectively over IP networks. This provides flexibility at the cost of a lack of features, and forces the agent toolkit developers to provide the required functionality. The IMPACT, *living markets* and JADE toolkits use the more heavyweight JAVA RMI [12]. To a certain extent, this is a limiting factor, since RMI consumes resources and can complicate the deployment process. In general it can be argued that remote method invocation techniques run counter to the philosophy of multi-agent systems, where agents should call on other agents to perform tasks using high-level communication language. RMI is perhaps more suitable for traditional distributed systems where the purpose is to abstract network issues and allow the remote invocation of methods in a manner that appears similar to local method invocation.

RETSINA and *living markets* provide a more balanced use of low-level services. RETSINA makes interesting use of the relatively lightweight SSDP discovery protocol for dynamic infrastructure discovery, and allows direct communication between agents. The RETSINA approach indicates how new generation middleware technology can be highly effective in providing the low-level services required by multi-agent systems.

RETSINA and ZEUS allows agents to operate as stand-alone programs with all the required functionality to support participation in and communication with other agents contained in each agent. This provides significant flexibility, and allows the developer to choose which infrastructural services an agent needs to participate and, as a consequence, support. JACK, JADE, IMPACT and *living markets* all provide some sort of container from within which agents should operate. On the one hand, the benefits of a container are obvious, since the container can contribute significantly to the required supporting services for all agents, provide a standardized means for handling communications, and seems to be the only way to support mobility for agents. On the other hand, using a container brings with it related costs and limits developers significantly in how much liberty they can have in employing alternative approaches. Perhaps the solution would be for toolkits to support both modes of operation to give developers the best of both worlds, as *living markets* supports to a certain extend for agents operating on limited-capability devices.

12 Alhtough alternatives are possible even within these toolkits RMI is the default and most well-supported approach

1.10.2.2 High-Level Services

The system review has revealed that the single most important high-level service functionality is the discovery of other agents. The solutions revolve around White Pages and Yellow Pages services. ZEUS, RETSINA and JADE provide both, while IMPACT provides only a Yellow Pages service, and JACK only a White Pages service. The *living markets* toolkits provides something similar to a Yellow Pages services along with the notion of *softmatching*, which allows a relaxation of the matching criteria, a feature that is very appropriate in business settings.

IMPACT differentiates itself from the others through an alternative approach for Yellow Pages services by using a simple service description language and a powerful similarity matching algorithm. This acknowledges that in heterogeneous environments, service descriptions and requests may not always be consistent and some means are required to deal with the problem. This approach might become more common in the emerging environment of the Semantic Web and Web Services.

Nevertheless, since IMPACT does not provide a White Pages service, it excludes the possibility that an agent may have prior knowledge of which agent it wants to contact but not the contact address of that agent. Furthermore, it makes it difficult to monitor exactly which agents are registered in a multi-agent system. ZEUS and JADE consider this a vital function and require that all agents register with the White Pages service. JACK seems to assume that in most cases agents will have prior knowledge of which agents they need to contact and may not even require a White Pages service. RETSINA is the most flexible toolkit in this respect, since it provides both White and Yellow Pages services, but does not require that either is available to bootstrap the system since infrastructure services can be dynamically discovered.

Agent communication languages are clearly supported by all the systems (ZEUS-FIPA, RETSINA-KQML, JADE-FIPA, Jack-FIPA and *living markets*-FIPA). IMPACT supports message exchange between agents but does not define a specific agent communication language. The benefits of a standardized communication language, such as FIPA, are obvious in open environments but are not as clear in more specialized domains where a simpler approach may provide the required functionality.

Some form of ontology support is available in all the systems, with ZEUS, JADE and JACK providing a more open approach where support is provided but specific ontologies are considered to be an application domain issue. RETSINA provides support for large, generic ontologies that can cover a wide range of domains. IMPACT allows for ontologies to *develop* as new agents enter the system, by allowing developers to relate the new concepts introduced to existing ones. Again, IMPACT's approach may indicate that this is a possible path for large-scale heterogeneous systems where consistency is difficult to achieve. Nevertheless, the process may become unwieldy if too many new concepts are introduced, and relationships between them must be made by developers.

Although all the systems concerned can claim to support multi-agent systems in heterogeneous environments, the claim for support for *open* heterogeneous systems is much harder to maintain. Open heterogeneous systems require standardization, such as that provided by FIPA, but at the same time need to acknowledge that flexible infrastructure support like RETSINA's is vital, and systems for dealing with inconsistencies such as IMPACT's will play a very important role.

1.10.3 Agent-building software

All the toolkits, with the exception of JADE, provide some agent building software, with JACK's and *living markets* being the most refined, a clear reflection of their commercial background. It could be argued that what is required is a development environment like the one JACK or *living markets* provides, combined with the infrastructural services provided by the other toolkits. Although agent technologies have progressed significantly, research projects cannot expend the resources required to develop sophisticated development environments. This is clearly an issue that must be dealt with through the take-up and support of such technologies by industry.

Another important issue in relation to agent-building software is the methodology used to develop a system. A refined agent development environment could also act as a guide through the methodology for an agent-based development. JACK, ZEUS and *living markets* all allow for these through different routes. However, similarly to infrastructure issues there is as yet no clearly agreed common methodologies.

1.10.4 Management Services

All of the systems provide some sort of management, but this is clearly an area that still requires development, and the way in which to proceed is not clear. ZEUS adopts an approach whereby each agent is interrogated about its actions by specialized agents to provide a visualization of the whole system. This method is clearly costly. RETSINA provides support for visualization of the system through the Logger module, bypassing the usual agent communication channels. This is still expensive, but can be more effective than the ZEUS approach. JADE allows monitoring of activity by interrogating containers about the agents operating within them. Finally, JACK provides a comprehensive approach for visualization that is integrated with the development environment. However, it is unclear how these methods scale, and how large-scale open agent systems could be managed by these means.

References

[1] Ronald Ashri, Michael Luck, and Mark d'Inverno. Infrastructre Support for Agent-based Development. In M. d'Inverno, M. Luck, M. Fisher, and C. Preist, editors, *Foundations and Applications of Multi-Agent Systems*, volume 2403 of *LNAI*, pages 73–88. Springer, 2002.

[2] Ronald Ashri, Iyad Rahwan, and Michael Luck. Architectures for Negotiating Agents. In V. Marik, J. Muller, and M. Pechoucek, editors, *Mutli-Agent Systems and Applications III*, volume 2691 of *LNAI*, pages 136–146. Springer, 2003.

[3] K.S. Barber, R. McKay, M. MacMahon, C.E. Martin, D.N. Lam, A. Goel, D.C. Han, and J. Kim. Sensible Agents: An Implemented Multi-Agent System and Testbed. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 92–99, 2001.

[4] Joachim Baumann, Fritz Hohl, Kurt Rothermel, Markus Strasser, and Wolfgan Theilmann. MOLE: A mobile agent system. *Software - Practice and Experience*, 32(6):575–603, 2002.

[5] P. Bellavista, A. Corradi, and C. Stefanelli. A secure and open mobile agent programming environment. In *Proccedings of the Fourth International Symposium on Autonomous Decentralized Systems*, pages 238–245. IEEE Computer Society Press, 1999.

[6] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing Multi-agent Systems with JADE. In C. Castelfranchi and Y. Lesperance, editors, *Intelligent Agents VII: Agent Theories Architectures and Languages*, volume 1986, pages 89–103. Springer, 2000.

[7] Ladislau Boloni, Kyungkoo Jun, Krzysztof Palacz, Radu Sion, and Dan C. Marinescu. The Bond Agent System and Applications. In D. Kotz and F. Mattern, editors, *Agent Systems, Mobile Agents, and Applications*, volume 1882 of *LNCS*. Springer, 2000.

[8] Ladislau Boloni and Dan C. Marinescu. An object-oriented framework for building collaborative network agents. In A. Kandle, K Hoffman, D. Mlynek, and N.H. Teodorescu, editors, *Intelligent Systems and Interfaces*, pages 31–64. Kluwer Publishing, 2000.

[9] Ladislay Boloni and Dan C. Marinescu. Agent Surgey: The Case for Mutable Agents. In José D. P. Rolim, editor, *Parallel and Distributed Processing, 15 IPDPS 2000 Workshops*, volume 1800 of *LNCS*, pages 578–585, 2000.

[10] J. M. Bradshaw, M. Greaves, H. Holmack, T. Karygiannis, W. Jansen, B. G. Silverman, N. Suri, and A. Wong. Agents for the Masses. *IEEE Intelligent Systems*, 14(2):53–63, 1999.

[11] Joris Claessens, Bart Preneel, and Joos Vandewallw. (How) can mobile agents do secure electronic transactions on untrusted hosts? A survey of the seecurity issues and the current solutions, volume=3, year=2003. *ACM Transactions on Internet Technology*, (1):28–48.

[12] Rem Collier, Gregory O'Hoare, Terry Lowen, and Colm Rooney. Beyond Prototyping in the Factory of Agents. In Vladimir Marik, Jorg Muller, and Michal Pechoucek, editors, *Multi-Agent Systems and Applications III*, volume 2691 of *LNAI*, pages 383–393, 2003.

[13] Don Cruischank, Luc Moreau, and David De Roure. Architectural Design of a Multi-Agent System for Handling Metadata Streams. In *The 5th ACM International Conference on Autonomous Agents*, pages 505–512, 2001.

[14] K. Decker, K. Sycara, and M. Williamson. Middle-Agents for the Internet. In *Proceedings of the 15th Joint Conference on Artificial Intelligence*, pages 578–573. Morgan Kaufmann, 1997.

[15] Scott A. DelOach, Eric T. Matson, and Yonghua Li. Multiagent Systems Engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 11(3), 2001.

[16] Thomas Eiter and V.S. Subrahmanian. Heterogeneous Active Agents, II: Algorithms and Complexity. *Artificial Intelligence*, 108(1–2):257–307, 1999.

[17] Thomas Eiter, V.S. Subrahmanian, and George Pick. Heterogeneous Active Agents, I:Semantics. *Artificial Intelligence*, 108(1–2):179–255, 1999.

[18] Thomas Eiter, V.S. Subtahmanian, and Timothy Rogers. Heterogeneous Active Agents, III:Polunomially implementable agents. *Artificial Intelligence*, 117(1):107–167, 2000.

[19] C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[20] J. Ferber and O. Gutknetch. A meta-model for the analysis of organisations in multi-agent systems. In *Proceedings of the Third International Conference on Multi-Agent Systems*, pages 128–135, 1998.

[21] John Graham. *Real-Time Scheduling in Distributed Multi-AGent Systems*. PhD thesis, University of Delaware, 2001.

[22] John Graham and Keith Decker. Towards a Distributed Environment-Centered Agent Framework. In N.R. Jennings and Y. Lesperance, editors, *Intelligent Agents VI Agent Theories, Architectures, and Languages*, volume 1757 of *LNCS*. Springer, 1999.

[23] Robert Gray, David Kotz, George Cybenko, and Daniela Rus. D'Agents: Security in a multiple-language, mobile agent system. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 154–187. Springer-Verlag, 1998.

[24] Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. Mobile agents: Motivations and State of the Art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2001.

[25] M. He, N.R. Jennings, and H. Leung. On agent-mediated electronic commerce. *IEEE Trans on Knowledge and Data Engineering*, 15(4), 2003.

[26] Yang Kun, Guo Xin, and Liu Dayou. Security in mobile agent system: problems and approaches. *ACM SIGOPS Operating Systems Review*, 34(1):21–28, 2000.

[27] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java(tm) Mobile Agents with Aglets(tm)*. Addisson-Wesley, 1998.

[28] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[29] Chung Fan Liu and Chyi Nan Chen. A sliding-agent-group communication model for constructing a robust roaming environment over internet. *Mobile Networks and Applications*, 8(1):61–74, 2003.

[30] Scott A. De Loach and Mark Wood. Developing Multiagent Systems with agentTool. In C. Castelfranchi and Y. Lesperance, editors, *Intelligent Agents VII - Agent Theories, Architectures and Languages*, volume 1986 of *LNCS*, pages 46–60, 2001.

[31] A. R. Lomuscio, M. Wooldridge, and N.R. Jennings. A classification scheme for negotiation in electronic commerce. *International Journal of Group Decision and Negotiation*, 12(1):31–56, 2003.

[32] David L. Martin, Adam J. Cheyer, and Douglas B.Moran. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, 1999.

[33] Luc Moreau, Norlizza Mohamad Zaini, Don Cruishanck, and David De Roure.

[34] H. Nwana, D. Ndumu, L. Lee, and J. Collis. ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems. *Applied Artifical Intelligence*, 13(1):129–186, 1999.

[35] P.D. O'Brien and M.E. Wiegand. *Agents of Change in Business Process Management*, volume 1198 of *LNAI*, pages 132–145. Springer, 1997.

[36] David V. Pynadath and Milind Tambe. The Communicative Multiagent Team Decision Problem: Analyzing Teamwork Theories and Models. *Journal of Artificial Intelligence and Research*, 16:389–423, 2002.

[37] Juan Antonio Rodriguez-Aguilar and Carles Sierra. Enabling Open Agent Institutions. In Kerstin Dautenhahn, Alan H. Bond, Lola Canamero, and Bruce Edmonds, editors, *Socially Intelligent Agents: Creating relationships with computers and robots*. Kluwer, 2002.

[38] George Samaras and Christoforos Panayiotou. Personalised portals for the wireless user based on mobile agents. In *Procerdings of the second international workshop on Mobile commerce*, pages 70–74. ACM Press, 2002.

[39] V.S. Subrahmanian, Piero Bonatti, Jurgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.

[40] K. Sycara, S. Widoff, M. Klusch, and J. Lu. LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. *Autonomous Agents and Multi-Agent Systems*, (5):173–203, 2002.

[41] Katia Sycara, Massimo Paolucci, Martin van Velsen, and Joseph Giampapa. The RETSINA MAS Infrastructure. *Autonomous Agents and MAS*, 7(1–2), 2003.

[42] R. Titmuss, I.B. Crabtree, and C.S. Winter. *Agents, Mobility and Multimedia Information*, volume 1198 of *LNAI*, pages 146–159. Springer-Verlag, 1997.

[43] Christian F. Tschudin. Mobile agent security. In Matthias Klusch, editor, *Intelligent Information Agents*, pages 431–446. Springer-Verlag, 1999.

[44] Tom Wagner, Bryan Horling, Victor Lesser, John Phelps, and Valerie Guralnik. The Struggle for Reuse: Pros and Cons of Generalization in Taems and its Impact on Technology Transition. In *Proceedings of the ISCA 12th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2003)*, 2003.

[45] M. Winikoff, L. Padgham, and J. Harland. Simplifying the Development of Intelligent Agents. In *AI2001: Advances in Artificial Intelligence. 14th Australian*, pages 557–568, 2001.

[46] F. Zambonelli, N.R. Jennings, and M. Wooldridge. Organisational rules as an abstraction for the analysis and design of multi-agent systems. *INternational Journal of Software Engineering and Knowledge Engineering*, 11(3):303–328, 2001.