Software Agents: A Literature Survey

10.6.2005

Helsinki University of Technology
Systems Analysis Laboratory

Tomi Myllylä
tomi.myllyla@hut.fi
55289W
# Table of Contents

1. Introduction .................................................................................................................. 3
2. Definitions .................................................................................................................... 5
   2.1 Software Agent ........................................................................................................... 5
   2.2 Artificial Intelligence (AI) ....................................................................................... 5
   2.3 Distributed Artificial Intelligence (DAI) ................................................................. 5
   2.4 Multi-Agent Systems (MAS) ................................................................................... 5
   2.5 Programming Paradigm .......................................................................................... 5
   2.6 Agent Communication Language (ACL) .................................................................. 6
3. What is a Software Agent? ............................................................................................ 6
4. Programming Paradigms and Software Agents: A Brief Historical Overview .......... 7
5. Agent Communication ................................................................................................. 10
   5.1 Introduction ............................................................................................................... 10
   5.2 Agent Communication Languages (ACL) ............................................................. 11
      5.2.1 KQML ............................................................................................................... 11
      5.2.2 FIPA ACL ....................................................................................................... 14
   5.3 Content Languages ................................................................................................. 17
      5.3.1 Knowledge Interchange Format (KIF) .............................................................. 17
      5.3.2 FIPA Semantic Language (FIPA SL) .............................................................. 18
   5.4 Ontology (Nucleus) ............................................................................................... 19
   5.5 Message Transport ................................................................................................. 19
6. The Actual Construction: Platforms and Tools (KQML, FIPA) .................................... 20
7. Conclusion ................................................................................................................... 22
8. References ..................................................................................................................... 24
9. APPENDICES ............................................................................................................. 26
   9.1 APPENDIX A: Example Software Agent Application Created With JADE [17] ....... 26
“An interrogator is connected to one person and one machine via a terminal. Her task is to find out which of the two candidates the machine is, and which is human only by asking them questions. If the interrogator cannot make a decision within a certain time the machine is considered to be intelligent.”

(The Turing Test)

1. INTRODUCTION

Software agents are autonomous pieces of software that conduct several tasks delegated to them. In the era of endless information flows, benefits can be achieved by authorizing certain kind of tasks to be done automatically by small independent software programs. Software agents are continuously running, personalized and semi-autonomous, and this makes them useful for a wide variety of information and process management tasks [1].

In the wider context, software agents represent an area of Artificial Intelligence (AI) combined to computer science [11]. Distributed Artificial Intelligence (DAI) is a subfield of AI that is concerned with solving problems in distributed manner and one of its research areas is agent-based computing. This is illustrated in Figure 1 [11].
Game theory has been widely used in order to understand the behavioral framework of software agents. Game theory suits well for this purpose since agents can be viewed as independent decision-makers that try to achieve their personal objectives and maximize their utilities. Of course, these can be different depending on agents. In most of the cases agents need to negotiate and collaborate with other agents in order to achieve their objectives.

This paper concentrates on software agent technologies. The main purpose is to go beyond the mathematical foundation of how agents should work, and rather answer the question of how software agents are actually constructed. What are the main technologies used to build software agents? How do agents communicate? What are the roles of traditional programming paradigms in constructing software agents? Finally, numerous terms and protocols exist in software agent world and this paper tries to identify and explain the most important ones in order for the reader to gain a grasp of the subject. In the end an example is provided about building a software agent.
2. DEFINITIONS

2.1 Software Agent

Numerous definitions exist for software agent and there is no single commonly accepted one. Brawshaw [3] states the following definition: Software agent is a software entity that functions continuously and autonomously in a particular environment, which may contain another agents and processes.

2.2 Artificial Intelligence (AI)

Wikipedia [13] defines artificial intelligence as “intelligence exhibited by any manufactured system”.

2.3 Distributed Artificial Intelligence (DAI)

Distributed Artificial Intelligence is a subfield of AI research dedicated to the development of solutions for complex problems that are not easily solveable with classic algorithmic programs. There are three main streams in DAI research: Parallel program solving, Distributed program solving, and Agent-based problem solving [13].

2.4 Multi-Agent Systems (MAS)

Multi-Agent Systems (MAS) are systems composed of multiple agents [11].

2.5 Programming Paradigm

Programming paradigm represents the programming style. Object-oriented programming and functional programming are examples of different programming paradigms.
2.6 Agent Communication Language (ACL)

Dogac and Singil [6] provide the following definition for Agent Communication Languages: ACL is a language that provides a set of application-independent primitives that allow an agent to state its intention in an attempt to communicate with other agents.

3. WHAT IS A SOFTWARE AGENT?

In the definition we saw that a software agent is a piece of software that is able to act autonomously in particular environment. Figure 2 from Wooldridge [14] illustrates how agent interacts with its environment.

![Figure 2: Software agent and its environment [14]](image)

However, we also saw that there is no commonly accepted definition either. Generally, we can state that an agent is a software entity that is able to conduct information-related tasks without human supervision [6], which can be viewed as an autonomic property. Nevertheless, autonomy alone usually isn’t considered adequate feature for software agents [11]. Chira [11] reminds that software agents should also possess following features:

- **Reactivity**

  Agent should be able to perceive its environment and respond to changes that occur in it (reactive behavior).

- **Proactivity**
An agent should also have the ability to take the initiative and not only react to external signals. This helps agent to pursue its individual goals (goal-directed behavior)

- **Cooperation**

An agent should be able to interact with other agents. This can be arranged via agent-communication language

There are also several other requirements stated by many individual researches but the previously mentioned are the most commonly accepted ones. Software agents can also be classified to different categories depending of their properties. Examples of these can be collaborative agents that emphasize cooperation with other agents, mobile agents that are capable of roaming wide area networks and information agents that are designed to manage, manipulate or collate the vast amount of information available for many distributed sources [11].

### 4. PROGRAMMING PARADIGMS AND SOFTWARE AGENTS: A BRIEF HISTORICAL OVERVIEW

Table 1 illustrates the simplistic evolution of programming languages used by Parunak and Van Dyke [15]. When programming with Top-down programming methods (Monolithic Programming) programmers have full control of programs they are writing. In this programming paradigm source code is generally not reusable or modular, and each program is a separate entity used only for certain purpose. The programmer is responsible of the state of the program and system operator determines the invocation [4]. Modular Paradigm brings in the procedural approach and reusable components of source code. Furthermore, Object-Oriented Programming Paradigm introduces the state of the program as internal states of individual objects created by programmer. In addition to individual states objects also maintain own actions defined as methods. Object-Oriented Paradigm has been associated to the development of software agents since there are many intuitive similarities between objects and software agents [4]. That is, software agents are independent and they have their own rules according to which they act. However, Odell [4] also brings out interesting difference between objects and software agents, that is,
software agents are also autonomous: they are empowered to act, in other words, they take *initiative* to achieve their goals. Objects in traditional OOP paradigm are considered passive since their actions have to be invoked by caller. Second, Odell [4] proposes that agents may respond to interaction any way they choose: they can accept or refuse the proposed action. In other words, software agents can behave unexpectedly. Objects in OOP tend towards more predictable actions. Therefore, Object-Oriented Paradigm does not fully respond to the agent programming needs from every point of view, although most of the agent actions can still be implemented with Object-Oriented programming languages [4]. The difference is more philosophical than practical. In addition, one interesting detail is that, in the agent world, other interactors may not know the behaviour of the software agent in the system. In OOP the interactor has to have some knowledge about the object: it must be known what methods to invoke. More about comparison between the OOP paradigm and software agents can be found from [4].

Table 1: Evolution of Programming Approaches. Reproduced from [15].

<table>
<thead>
<tr>
<th></th>
<th>Monolithic Programming</th>
<th>Modular Programming</th>
<th>Object-Oriented Programming</th>
<th>Agent Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unit Behavior</strong></td>
<td>Nonmodular</td>
<td>Modular</td>
<td>Modular</td>
<td>Modular</td>
</tr>
<tr>
<td><strong>Unit State</strong></td>
<td>External</td>
<td>External</td>
<td>Internal</td>
<td>Internal</td>
</tr>
<tr>
<td><strong>Unit Invocation</strong></td>
<td>External (CALLed)</td>
<td>External (message)</td>
<td>Internal (rules, goals)</td>
<td></td>
</tr>
</tbody>
</table>

Shoham [16] proposed a new programming paradigm called Agent-Oriented Programming (AOP) in the early 1990s. AOP is a specialization of Object-Oriented Programming paradigm and it allows programming of agents in terms of their mental states. This includes components like beliefs, decisions, capabilities and obligations [11]. In this paradigm agent programs control agents and communications primitives such as informing, requesting and offering can be used in interaction [11]. Table 2 presents the relation of AOP and OOP programming paradigms. Shoham also developed a prototype
AOP programming language called AGENT0 [11]. In addition, some other agent programming languages also exist and lot of research is made in that area. However, despite the AOP research traditional programming languages are still mostly used in constructing software agent applications [11]. Currently the most popular language for software agent construction seems to be Java and many agent building toolkits and platforms exist for this popular language [11]. More information about these toolkits is provided in chapter 6.

Table 2: The relation between OOP and AOP Paradigms [16]

<table>
<thead>
<tr>
<th></th>
<th>OOP</th>
<th>AOP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Unit</strong></td>
<td>Object</td>
<td>Agent</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Defining State of Basic Unit</strong></td>
<td>Unconstrained</td>
<td>Beliefs, commitments, capabilities, choices…</td>
</tr>
<tr>
<td><strong>Process of Computation</strong></td>
<td>Message passing and response methods</td>
<td>Message passing and response methods</td>
</tr>
<tr>
<td><strong>Types of Messages</strong></td>
<td>Unconstrained</td>
<td>Inform, request, offer, promise, decline…</td>
</tr>
<tr>
<td><strong>Constraints on Methods</strong></td>
<td>none</td>
<td>Honesty, consistency</td>
</tr>
</tbody>
</table>
5. AGENT COMMUNICATION

5.1 Introduction

Agents may have numerous objectives and goals, and although they are individual and autonomous they still often have to collaborate and communicate with other agents in order to achieve their objectives [6]. Dogac and Singil [6] state that in order to collaborate with others, agents are required to:

- Discover the existence, network addresses, capabilities and/or roles of other agents;
- Communicate with other agents through an agent-independent, that is, a standard agent communication language.

Nameservers and facilitators are provided by Multi-Agent Systems to support agent discovery. Agents register their addresses to a name server and their features and abilities to a facilitator. Agents can then use those nameservers and facilitators as a reference to find out abilities and addresses of other agents [6].

After the agents have discovered each other they need to communicate in order to achieve their goals. Communication can be divided into two fundamental parts [8]. First, agents need to agree on a common agent communication language (ACL) that provides the basis for stating intentions to other party [8]. Second, mere common language is not enough but agents must also have common vocabularies for representing shared domain concepts and application-dependent content [6]. This includes both a shared ontology and the content that is defined by a Content Interchange Format (CIF) [6]. Therefore, agent communication languages basically consist of these three layers that are shown in Figure 6.
5.2 Agent Communication Languages (ACL)

Speech act theory is the basis of most popular agent communication languages. Speech act theory was originally developed by linguists in an attempt to understand how humans use language in everyday situations [5]. In speech act theory, human expressions are viewed as actions, in the same way as actions performed in the everyday physical world (e.g. picking up a block from table) [5]. As a result of this, speech act theory uses the term *performative* to identify the intention behind spoken communication. Examples include verbs like request, tell or inform. Performatives are used as constraints of semantics in communication between agents, and they simplify how agents should react to messages they receive [7]. Two most successful agent communication languages so far have been Knowledge Query and Manipulation Language (KQML) and FIPA-ACL.

5.2.1 KQML

KQML was one of the first initiatives to specify how to support communication of agents using a protocol based on speech acts theory [7]. However, currently no single consensus or true de facto standard exist on KQML specification. As a result, many different KQML “dialects” exist and different agent systems may speak slightly different language and not
fully understand each other [7]. According to Dogac and Singil [6] KQML has following key features:

- KQML messages are opaque to the content they carry. KQML messages do not merely communicate sentences in some language but rather communicate an attitude about the content (assertion, request, query, basic response, etc.).

- The language primitives are called performatives (as stated earlier).

- At the agent level, the communication appears as point-to-point message passing.

- Special agents, called facilitators, may exits in KQML environment. Facilitators provide agents additional networking-related functions such as association of physical addresses with symbolic names, registration of agents and/or services offered and sought by agents, enhanced communication services such as forwarding, brokering and broadcasting.

Poslad, Bourne, Hayzelden and Buckle [7] state that communication between agents involves three aspects: the method of message passing, the format or syntax, of the information being transferred, and the meaning, or semantics of the information (and message). Any KQML message has the following syntax [7]:

(KQML-performative
  :sender <word>
  :receiver <word>
  :language <word>
  :ontology <word>
  :content <expression>
  ...
)

Example of KQML message could be [5]:

(tell
  :content "cost(bt, service-4, £5677)"
  :language standard prolog
  :ontology bt-services-domain
  :in-reply-to quote service-4
  :receiver customer-2
  :sender bt-customer-services)
The KQML performative of this message is tell, and the agent seeks to inform some customer, customer-2, of a quote for performing some service, service-4. The message is a reply to earlier request from customer-2 to bt-customer services. The content of the message is written in standard Prolog, and the ontology for BT’s services domain is used here [5].

A KQML message may have many different performatives. It depends of performative of the message what parameters, such as “sender”, “language” etc, are introduced in the actual KQML message. Table 3 introduces KQML performatives divided into several categories [5].

<table>
<thead>
<tr>
<th>Category</th>
<th>Reserved performative names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic informational performatives</td>
<td>tell, deny, untell, cancel,</td>
</tr>
<tr>
<td>Basic query performatives</td>
<td>evaluate, reply, ask-if, ask-about, ask-one, ask-all, sorry</td>
</tr>
<tr>
<td>Multi-response query performatives</td>
<td>stream-about, stream-all</td>
</tr>
<tr>
<td>Basic effector performatives</td>
<td>achieve, ınachieved</td>
</tr>
<tr>
<td>Generator performatives</td>
<td>standby, ready, next, rest, discard, generator</td>
</tr>
<tr>
<td>Capability definition performatives</td>
<td>advertise</td>
</tr>
<tr>
<td>Notification performatives</td>
<td>subscribe, monitor</td>
</tr>
<tr>
<td>Networking performatives</td>
<td>register, unregister, forward, broadcast, pipe, break</td>
</tr>
<tr>
<td>Facilitation performatives</td>
<td>broker-one, broker-all, recommend-one, recommend-all, recruit-one, recruit-all</td>
</tr>
</tbody>
</table>

In addition to the lack of agreed specification there are also some other problems related to KQML language. According to Poslad, Bourne, Hayzelden and Buckle [7], one of these problems is the lack of well-defined semantics. The use of performatives alone is insufficient to guarantee that other agents will interpret messages correctly.
5.2.2 FIPA ACL

The problems of KQML are the driving forces behind the FIPA specification of its own agent communication language FIPA ACL. The purpose of FIPA ACL is to provide a standard way to package messages in such a way that the meaning of the messages is clear to other compliant agents [7]. FIPA ACL tries to achieve this goal by reducing the number of performatives. Although there are huge amount of verbs in English corresponding to performatives, the FIPA-ACL defines only the minimal set of these verbs for agent communication (it consists of approximately 20 performatives) [6, 7]. This way more flexible approach for agent communication is achieved. Poslad, Bourne, Hayzelden and Buckle [7] state that FIPA ACL provides some benefits including:

- Dynamic introduction and removal of services.
- Customized services can be introduced without a requirement to recompile the code of the clients at run-time.
- Allowance for more decentralized peer-to-peer realization of software.
- A universal message-based language approach providing a consistent speech-act based interface throughout software (flat hierarchy of interfaces).
- Asynchronous message-based interaction between entities.

Table 4 introduces the FIPA ACL performatives, called communicative acts. As can be seen from the Table 4, FIPA ACL has 22 communicative acts, which is considerably less than KQML has. FIPA ACL communicative acts can be divided to primitive and composite communicative acts [8]. Composite communicative acts can be composed of primitive communicative acts either by substitution or sequencing. An agent only has to implement the communicative acts it needs and the “not-understood” act [8]. However, performative is only one part of FIPA-ACL message, and the message contains also many other elements shown in Table 5.
<table>
<thead>
<tr>
<th>Action</th>
<th>Performative</th>
<th>Action</th>
<th>Performative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept proposal</td>
<td>Inform</td>
<td>Reject proposal</td>
<td></td>
</tr>
<tr>
<td>Agree</td>
<td>Inform Ref</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td>Not Understood</td>
<td></td>
<td>Request</td>
</tr>
<tr>
<td>Call for Proposal</td>
<td>Propagate</td>
<td>Request When</td>
<td></td>
</tr>
<tr>
<td>Confirm</td>
<td>Propose</td>
<td>Request Whenever</td>
<td></td>
</tr>
<tr>
<td>Disconfirm</td>
<td>Proxy</td>
<td>Subscribe</td>
<td></td>
</tr>
<tr>
<td>Failure</td>
<td>Query If</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inform</td>
<td>Query Ref</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5: FIPA-ACL message elements. Reproduced from [6].

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performative</td>
<td>Denotes the type of the communicative act of the ACL message</td>
</tr>
<tr>
<td>Sender</td>
<td>Denotes the identity of the sender (the name of the agent) of the message</td>
</tr>
<tr>
<td>Receiver</td>
<td>Denotes the identity of the intended recipient of the message</td>
</tr>
<tr>
<td>Reply-To</td>
<td>This element indicates that subsequent messages in this conversation thread are to be directed to the agent named in the reply-to-element, instead of the agent named in the sender element</td>
</tr>
<tr>
<td>Content</td>
<td>Denotes the content of the message</td>
</tr>
<tr>
<td>Language</td>
<td>Denotes the language in which the content element is expressed</td>
</tr>
<tr>
<td>Encoding</td>
<td>Denotes the specific encoding of the content language expression</td>
</tr>
<tr>
<td>Ontology</td>
<td>Denotes the ontology(s) used to give a meaning to the symbols in the content expression</td>
</tr>
<tr>
<td>Protocol</td>
<td>Denotes the interaction protocol that the sending agent is employing with this ACL message</td>
</tr>
<tr>
<td>Conversation-id</td>
<td>Introduces an expression (a conversation identifier) which is used to identify the ongoing sequence of communicative acts that together form a conversation</td>
</tr>
<tr>
<td>Reply-with</td>
<td>Introduces an expression that will be used by the responding agent to identify this message</td>
</tr>
<tr>
<td>In-reply-to</td>
<td>Denotes an expression that references an earlier action to which this message is a reply</td>
</tr>
<tr>
<td>Reply-By</td>
<td>Denotes a time and/or date expression which indicates the latest time by which the sending agent would like to have received a reply</td>
</tr>
</tbody>
</table>

Dogac and Cingil [6] provide an example of a FIPA ACL message:

(cfp
  :sender (agent-identifier :name BuyerAgent)
  :receiver (set (agent-identifier :name SellerAgent))
  :content
    ((action (agent-identifier :name SellerAgent)
      (sell :movie Gladiator))
     (any ?x (and (= ?x (price Gladiator)) (< ?x 20))))
  :ontology movie-ontology
  :language FIPA-SL)
In this example BuyerAgent asks SellerAgent to sell the movie Gladiator for less than $20. The message structure begins with a word identifying the communicative act, which is here “cfp” meaning “call for proposal”. The rest of the message contains parameters, beginning with a colon following a keyword identifying the parameter name. In this example, sender (BuyerAgent) and receiver (SellerAgent) are specified in the parameters as well as the maximum price offered ($20). The remaining parameters specify the message content, content language and the used ontology [6]. Ontology means that agents must have a “common vocabulary” about the subject.

5.3 Content Languages

As we have seen earlier, both the KQML and FIPA-ACL messages contain parts that are marked with content parameter. This content part of the message defines the actual information about the matter agents are trying to communicate. Agent communication languages typically define the outer language (ACL) and an inner language (CIF), as was shown in Figure 3. The inner language, representing the content part of the message, allows an agent to express its actual application-dependent content to other agents [6]. In the previous examples this was the cost of the service or the price offer for the movie Gladiator. Agents must understand each other in this content language level to be able to successfully interact with each other. Two content languages among the most popular ones [8] are presented here. They are Knowledge Interchange Format (KIF) and FIPA Semantic Language (FIPA SL). The syntax of both these languages is inherited from Lisp programming language [8].

5.3.1 Knowledge Interchange Format (KIF)

The Knowledge Interchange Format (KIF) provides a high-level intermediate language for communication between agents [8]. KIF provides for the encoding of simple data, constraints, rules, quantified expressions, metalevel information and procedures [6]. KIF also provides the capability to define objects, functions, and relations related to knowledge
representation [6]. Dogac and Cingil [6] provide some introductory examples for representing simple data, called facts, with Knowledge Interchange Format:

(wage John 80 10)
(wage George 56 20)

These examples state that “John has worked for 80 hours with an hourly wage of 10 Dollars” and “George has worked for 56 hours with a hourly wage of 20 Dollars” [6].

Second example enhances the use of more complex ways of representing knowledge and relations of information:

(>(* (hours_worked Employee1) (hourly_wage Employee1))
 (* (hours_worked Employee2) (hourly_wage Employee2))

This example states that total wage of Employee1, which is calculated by multiplying the hours_worked with his hourly wage is greater than the total wage of Employee2 [6]:

Dogac and Cingil [6] also state that there are some tools used to manipulate KIF encoded information. These include Prologic and Epilog, which are libraries of Lisp subroutines. In addition, there are also many Java-based KIF parsers available [6].

5.3.2 FIPA Semantic Language (FIPA SL)

FIPA Semantic Language (FIPA SL) is a formal language developed to define the content of the FIPA-ACL. FIPA SL can be used to express objects, propositions and actions [8]. Object expression is used to declare variables and make assertions [10]. Action expressions describe some action that is either already performed, intended to be performed in the future or is currently being performed. Propositions are used to represent the behavioral aspects of agents like goals, intents, beliefs and uncertainty [10]. For example, agents may have persistent goals stated in the form

(PG < agent > < expression >)

This states that an agent holds a persistent goal that expression becomes true but will not necessarily possess a plan to achieve this [6].
The FIPA SL specification has three subsets: FIPA-SL0, FIPA-SL1 and FIPA-SL2. FIPA-SL0 is the most simple and FIPA-SL2 the most complex related to available logical features for representing information [8].

5.4 Ontology (Nucleus)

Even if agents speak the same language, they require some common understanding of the meaning of the message content. This is provided by specialized knowledge component called ontology that specifies the objects, concepts and relationships in a given domain [7]. In other words, agents have to use similar vocabularies for representing shared concepts in a domain, and agents are also able to extract information from various sources when these sources share the same underlying ontology [6].

According to Dogac and Cingil [6], ontologies are usually built using schema definition or a knowledge presentation language. Dogac and Cingil [6] also state that creating and expressing ontologies of any size is difficult and time-consuming work and many tools have been developed for analyzing and developing ontologies. One of these tools is called Ontolingua.

5.5 Message Transport

Now that we have defined how software agents communicate in agent-level we will take a look at the wider picture in order to understand how these agent messages actually move in the network. This includes linking these agent interaction protocols to network data transfer mechanisms.

Multiple layered agent messages are finally encoded into transport messages that, among with other information, include the information about message receiver and sender. These transport messages are sent through the network using various transport mechanisms. There are many transport services available that are able to transfer messages from one agent to another in the network. Higher-level protocols such as CORBA, DCOM and Java
RMI allow the use of remote procedure calls between distributed objects that hide the lower level complexity of the transfer mechanisms. Here we don’t dig further to the world of network and transport protocols, but following list summarizes some of the most common ones of these protocols:

- CORBA
- DCOM
- Remote Method Invocation using Java RMI
- Web Services
- HTTP
- Wireless Access Protocol

These are network-level protocols, and TCP/IP is a common example of a lower-level data-transfer protocol when transferring data through network.

6. THE ACTUAL CONSTRUCTION: PLATFORMS AND TOOLS (KQML, FIPA)

Now that we have provided the theory for agent interaction, communication and general agent architectures, we still haven’t answered the question how to actually construct software agents? Agent systems are difficult to build from scratch. Chira [11] states that numerous languages and platforms have been created by different research groups and companies to support the development of agent-based applications. However, traditional languages are still usually used to construct agent applications. Currently the most used programming language for developing agent applications is Java. This is mainly because of its rich library of functions related to concurrency, security, support for object-oriented programming techniques and multithreading.

Agent implementors don’t have to start building agents from scratch since numerous platforms and toolkits have been introduced for building agent applications, and they are widely available for third-party developers, application developers and end-users under
various licensing arrangements [7]. Agent toolkits are defined as sets of components from which to build agent systems and sets of tools to help operate agent systems [7]. Most agent toolkits are based on Java programming languages, and many implementations exist both on FIPA and KQML agent communication architectures. Some of these toolkits are presented in Table 6.

<table>
<thead>
<tr>
<th>Product Name (Company)</th>
<th>Mobile / Stationary</th>
<th>Language</th>
<th>Standards</th>
<th>Availability / Licensing</th>
<th>Example applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>JatLite (Stanford Uni.)</td>
<td>Stationary</td>
<td>Java</td>
<td>KQML</td>
<td>Open Source</td>
<td>Design Decision</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tracking, constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mgt, enterprise</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>control.</td>
</tr>
<tr>
<td>ZEUS (BT UK)</td>
<td>Stationary</td>
<td>Java</td>
<td>FIPA KQML</td>
<td>Open Source</td>
<td>Supply Chain mgt,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>service provisioning,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>network resource mgt.</td>
</tr>
<tr>
<td>FIPA-OS (Nortel Networks)</td>
<td>Stationary</td>
<td>Java</td>
<td>FIPA</td>
<td>Open Source</td>
<td>Personalized services,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VPN, VHE, meeting</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>scheduler, etc.</td>
</tr>
<tr>
<td>JADE (CSELT)</td>
<td>Stationary</td>
<td>Java</td>
<td>FIPA</td>
<td>Open Source</td>
<td>Travel Assistant,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>audio-visual</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>entertainment,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>meeting scheduler,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>etc.</td>
</tr>
</tbody>
</table>

An example implementation of agent application built with JADE is provided in appendix A.
7. CONCLUSION

As we have seen, the world of software agents is quite complex and fragmented. Numerous protocols, standards and tools exist in this area, and there is no single standard to use when implementing software agents. In this paper I tried to uncover the most important standards and protocols of this wide area, and also take a practical view on the subject how to actually implement software agents. Therefore, numerous platforms and tools were presented for building software agents and also an example was provided how to actually construct an agent application.

Figure 4 from Dogac and Cingil [6] summarizes the journey we went through. At the beginning we took a look at the agent communication. We saw that agents communicate through Agent Communication Languages (ACL), and these languages are typically divided into three layers: common agent communication language, content language and ontology. Agent communication language provides the basis for agent communication through communication primitives, and most of the popular agent communication languages today enhance speech act theory, developed by linguists, for stating agent intentions. Two most successful agent communication languages so far have been Knowledge Query and Manipulation Language (KQML) and FIPA-ACL. In addition to common language for stating intentions agents also need to have common vocabularies and common meaning for content that is being exchanged between them. The two inner layers of ACL languages provide methods to address this, and important standards such as Knowledge Interchange Format (KIF) were presented.

Furthermore, agent communication standards present only the agent-level communication framework and they have to be linked to network level communication mechanisms in order to fully understand how software agents work and how they are constructed. Figure 4 [6] presents standards used below the semantic layers of agent communication. This includes standards for data representation, such as XML and CORBA IDL, and actual transfer protocols at the network and transport levels.

Finally, we ended our journey by presenting numerous tools to build software agents, and also provided an example software agent with JADE toolkit using Java programming language.
Figure 4: Software agent technology stack [6].
8. REFERENCES


9. APPENDICES

9.1 APPENDIX A: Example Software Agent Application Created With JADE [17]

An example of software agent application created with JADE toolkit is provided here. The example is from Gerstner Laboratory’s JADE tutorial [17] and it demonstrates nicely the message passing between agents.

We will develop two agents. The first one sends to the second one simple message. The agents use Agent Communication Language (ACL) for the communication, and each message is represented by the class jade.lang.acl.ACLMessage.

At the beginning, the agent SenderAgent sends the message to the agent with the name jack. The ReceiverAgent is waiting for the messages and, if some message arrives, it prints the message on the console.

The class SenderAgent is very simple. It creates new instance of the ACLMessage and fills the receiver and the content attribute. At the end it uses the method send(ACLMessage m).
```java
package tutorial.simplemessage;

import jade.core.AID;
import jade.core.Agent;
import jade.lang.acl.ACLMessage;

public class SenderAgent extends Agent {
    protected void setup() {
        System.out.println("Hello. My name is "+this.getLocalName());
        sendMessage();
    }

    private void sendMessage() {
        ACLMessage aclMessage = new ACLMessage(ACLMessage.REQUEST);
        aclMessage.addReceiver(new AID("jack"));
        aclMessage.setContent("Hello! How are you?");
        this.send(aclMessage);
    }
}
```

Receiving messages is little more complicated than sending messages. You must write some behaviour that is responsible for processing the incoming messages. You have to add the behaviour in the `setup()` method to the list of behaviours.
package tutorial simplesmessage;

import jade.core.Agent;

public class ReceiverAgent extends Agent {

  protected void setup() {
    System.out.println("Hello. My name is "+this.getLocalName());
    addBehaviour(new ResponderBahaviour(this));
  }
}

The class ResponderBahaviour is extended from the SimpleBehaviour class, that is an abstract class. Therefore you must override its methods action() and done(). In the method action() you process incoming messages. The method done() returns true only if the behaviour is finished. In our case this method returns always false. You must also specify, which messages may be processed by this behaviour. In our case, this behaviour processes only messages with performative REQUEST. This constrain is specified in the class MessageTemplate.
package tutorial.simplemessage;

import jade.core.Agent;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.core.behaviours.SimpleBehaviour;

public class ResponderBahaviour extends SimpleBehaviour {

    private static final MessageTemplate mt =
            MessageTemplate.MatchPerformative(ACLMessage.REQUEST);

    public ResponderBahaviour(Agent agent) {
        super(agent);
    }

    public void action() {
        while (true) {
            ACLMessage aclMessage = myAgent.receive(mt);
            if (aclMessage!=null) {
                System.out.println(myAgent.getLocalName()+" : I receive message."
                                    +aclMessage);
            } else {
                this.block();
            }
        }
    }

    public boolean done() {
        return false;
    }
}
You can run this example as:

```java
>java jade.Boot john:tutorial.simplemessage.SenderAgent
jack:tutorial.simplemessage.ReceiverAgent
```

As a result you can see something like:

Hello. My name is john
Hello. My name is jack
jack: I receive message.

(REQUEST
 :sender (agent-identifier :name john@bubik-small:1099/JADE)
 :receiver (set (agent-identifier :name jack@bubik-small:1099/JADE))
 :content "Hello! How are you?"
)