

Supporting Flexible Construction of Agents through an Explicit Agent Model

Bart Vanhaute *Bart De Decker*

Report CW 321, April 2001



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Supporting Flexible Construction of Agents through an Explicit Agent Model

Bart Vanhaute Bart De Decker

Report CW 321, April 2001

Department of Computer Science, K.U.Leuven

Abstract

Abstract Programming agents is a hard task, mainly because of the diverse aspects of an agent that need to be considered. A developer has to decide how the functionality of the agent is implemented, in what form the agent will store its information and how it will handle the sometimes complex interactions with other agents. Additionally, there are the many security requirements to take into account. Access to agent services needs to be controlled, network communication should be protected, accounting must be performed, agent data need to be protected, etc. The current agent platforms do not really support these aspects very well. As a result, the agent programs become very tangled and thus are hard to maintain, evolve and reuse. In this paper, we present an open agent platform, targeted at experimentation, both in application structure and security. The platform starts from a more explicit model of an agent and its platform. We argue that this model allows more reuse and flexibility in the construction of agent applications.

Keywords : agent model, composition, flexibility, separation of concerns
CR Subject Classification : D.2.11 Domain Specific Architecture, D.2.13 Reusable Software, I.2.11 Multiagent Systems

Supporting Flexible Construction of Agents through an Explicit Agent Model

Bart Vanhaute

DistriNet, Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Leuven, BELGIUM
Tel. (+32)16327069, Fax (+32)16327996
Bart.Vanhaute@cs.kuleuven.ac.be

Bart De Decker

DistriNet, Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Leuven, BELGIUM
Tel. (+32)16327633, Fax (+32)16327996
Bart.DeDecker@cs.kuleuven.ac.be

Keywords : agent model, composition, flexibility, separation of concerns

Abstract

Programming agents is a hard task, mainly because of the diverse aspects of an agent that need to be considered. A developer has to decide how the functionality of the agent is implemented, in what form the agent will store its information and how it will handle the sometimes complex interactions with other agents. Additionally, there are the many security requirements to take into account. Access to agent services needs to be controlled, network communication should be protected, accounting must be performed, agent data need to be protected, etc. The current agent platforms do not really support these aspects very well. As a result, the agent programs become very tangled and thus are hard to maintain, evolve and reuse. In this paper, we present an open agent platform, targeted at experimentation, both in application structure and security. The platform starts from a more explicit model of an agent and its platform. We argue that this model allows more reuse and flexibility in the construction of agent applications.

1 Introduction

Currently, the agent programming model is very ad-hoc, and does not appear to scale very well. Programming complex agent-applications becomes tedious and error-prone. In agent systems based on object-oriented programming languages [6, 7], one typically has to derive from an abstract base class that represents the generic structure of an agent. This inheritance relation constrains the composability of agents from a number of units of basic functionality. The units need to be tied together manually, with delegation from the general agent structure to its parts and vice versa. End-users with no real programming skills will therefore be unable to construct an agent by simple composition and configuration. What is needed is a clearer structure of an agent such that the agent functionality can be expressed more naturally and composition is supported more

directly.

Communication between agents is offered either through direct method invocations [6] on (proxies of) other agent objects or through a simple messaging service [10]. Building complex interaction patterns (such as contract negotiation and auction protocols) based on those simple constructs can be hard. Design patterns like OBSERVER and STATE are used to untangle the code [7], but result in an agent structure that is only more complex to deal with. Furthermore, the case of API-like specifications for direct communication thwarts software evolution. Maintaining upward and downward compatibility between different versions of an API is tedious. An ideal communication mechanism allows an easy implementation of complex interaction patterns, without constraining it to a fixed, non-evolvable model.

In addition to the basic functionality, there is the variable set of security requirements [5] to take into account. Agent platforms are open environments, and as such require more flexible security implementations. For instance, a rigid access control infrastructure, such as an access control matrix based on identities and operations, is not feasible if both the identities and the set of possible operations change over time. The only really workable approach starts from the idea that every entity in the system has to take care of its own security requirements. Although a considerable part of the security implementation can be shared between all parties, specific policies and custom mechanisms will always come up. Separation and reuse of these parts of an agent implementation is at least as important as reuse of its functionality. To conclude, a flexible solution should support an easy integration of security mechanisms into the functionality of agents and agent platforms.

The rest of the paper is structured as follows. First we lay out the requirements for the agent system. Then, we will present our model for an agent system and explain in what way it can achieve our goals. The following section deals with some implementation details, after which

we briefly describe how the model supports transparent addition of security. We conclude with a proposal for further developer support.

2 Requirements

Our overall goal is to develop an open agent platform targeted at experimentation. The experiments take place in two research fields. First, we want to be able to build complex agent applications in a way that promotes reuse and modularisation. Second, we want to treat security requirements as much as possible independently from these applications. More specifically, we have the following goals for the four major parts of the agent system.

2.1 The Agent

The model of the platform should allow a natural description of the agent application structure. We view an agent as an autonomous, self-contained entity, trying to work in an open environment. Agent applications are mainly centred around two main activities. Firstly, an agent has a goal to fulfil, that is not directly related to any other agent. Typically an agent has one or more roles, where each role describes a more specific part of the general task that it has to perform. Secondly, in its task it needs to communicate with other agents, respond to queries and process results. This is commonly summarised as the active and reactive nature of an agent. Internally, an agent encapsulates a state that is not accessible to agents. It might for instance want to record some of the information it has gathered or has some beliefs that are based on previous interactions. The model should support this description very naturally, such that an agent developer does not have to work around the limitation or a lack of proper support.

Furthermore, the agent model should have composition support. Expert agent programmers write modular units that implement typical functionality for an agent in a specific role, defined in a certain application domain. For instance one programmer could write a module to do price negotiation, while another programmer writes modules for product discovery and payment. End-users can compose agents from these units according to their needs, without the need for real programming skills.

2.2 The Agent Platform

We strive for a minimal agent platform specification. The more of the platform is (strictly) defined, the less it is useful under varied circumstances. The platform should be based on a small number of concepts. Any advanced feature should be optional, in order not to burden the basic design. At the same time, it should be possible to add features later on, without breaking the base model. This means the basic concepts must be generic enough to be extensible to new requirements. This also means any later extension to

the platform should not break the code of already existing agents.

The model for an agent platform will be implemented and installed on a variety of hardware. On embedded systems, the platform would only provide a few very simple services, whereas an implementation for a high capacity server could have many services, some of which are rather complex and demanding. If all implementations are based on the same model, it is possible to implement a (simple) agent that can run on any of those platform implementations. Obviously, an agent that requires one or more of the complex services will not be able to do anything on a platform that does not offer these. The agent should however be able to check what services are offered by the platform it is running on, and perhaps move to a more suitable platform if its requirements are not met.

2.3 The Communication Infrastructure

We want to provide the correct level of genericity in the agent communication facilities. As explained before, both direct method invocations and basic messaging services are too low-level to be directly usable by an agent programmer. Typical agent interactions often take the form of conversations, built up from an ordered sequence of messages sent back and forth between the agents. The communication infrastructure should support this relation between successive messages as well as the relation between incoming and outgoing messages of a concrete conversation.

A lot of research is being done on other forms of communication, like generative communication using tuple spaces [2], infobusses and event services that offer a higher level of uncoupling. Which of these is most useful depends on the actual agent application. Choosing one of them as the basic communication infrastructure will make it difficult to support the other. A full implementation of an agent system could support these through services that are built on top of its more simple communication implementation.

2.4 The Security Implementation

Security requirements in a agent system can vary enormously under different circumstances¹. Therefore, a fixed security implementation is unworkable. As every agent can have its own unique security requirements, there will potentially be a new implementation of these requirements in every other agent, even if the real functionality of the agent is the same. Vice versa, it is very likely that agents with varying functionality still have similar security requirements.

This situation calls for a separate description and implementation of those difficult security requirements. From a software engineering point of view, this separation of concerns is highly desirable. It enables separate verification of the implementation and reuse in many agents. Together

¹Agent servers with different levels of trust, closed intranet environment versus internet applications, information gathering agents versus stock exchange agents, etc.

Figure 1: Model of an agent

with the composition support in the agent model, different modules for the agent functionality and the security implementation can be combined, such that a tailored construction of agents according to specific user requirements is possible.

3 The Model

Our proposal for an agent system consists of three main parts.

- The model of an agent. This defines the internal structure of an agent. Its purpose is to support the construction of agents by end-users, by composition of reusable parts.
- The model for the agent platform. The specification is kept minimal, providing only basic services like agent creation, arrival and departure, and discovery.
- The model for the communication infrastructure. This model is based on explicit communication channels and agent references as reified entities.

The model of an agent contains the following parts, as illustrated in figure 1. A number of *tasks* implement the activity of an agent. Each task will typically implement one functional part or role of the global activity of an agent, although there could also be tasks implementing non-functional activities such as authentication. Each task has its own thread of control, so multiple task can be active at the same time within one agent. A number of primitives can be provided to synchronize tasks, if needed. The composition support of the model is partly the result of this division of functionality into tasks.

The reactive nature of an agent is supported through the *port* concept. Ports are used in the communication between agents. The basic operations on a port are `accept`, to wait for an incoming message or to read an already delivered message, and `send`, to send a message to another agent. As such, the ports define the interface to the communication infrastructure of the agent system. How the messages are actually transmitted is explained below. A port can be used in two situations. Firstly, a port is used when the agent wants to offer a service to other agents. For this purpose, the port is registered under a name within the agent. Secondly, a port is created when an agent wants to

communicate with another agent. In this case, the port is the result of requesting a particular service of a particular agent. Agents can refer to each other only through special agent references.

When tasks want to store information, they will use the *storage* part of an agent. The information is referenced using string based indexes. This separation between the processing part of an agent and the storage of more or less permanent data, makes it possible to share data between different tasks within one agent. It also enables extra processing over the data that is stored. We will come back to this in section 5.

Another important part of an agent is its *identity*. This is a fixed information structure that serves to (uniquely) identify the agent, across multiple agent platforms. The identity does not directly relate to its owner, creator, originator or any other principals involved. For this purpose an extra query can be added to agents, though this is not mandatory. Only when security requirements such as access control and accounting would call for this information, it needs to be included in the implementation.

Finally, every agent has a pointer to a *context*, the place it is executing in. Through this reference, an agent can use the services of the agent platform.

The agent platform provides an implementation of all the concepts used by an agent (such as identities and agent references) as well as the global agent structure itself. To create an agent, the agent user only has to give the platform an implementation for the tasks the agent will perform, and the initial data that is to be used. The platform will then set up the global agent structure that conforms to its implementation, and start the tasks.

The platform basically offers only one important service to the agents, namely the way to *communicate* with another agent. All other services are implemented as (stationary) agents, and can thus be contacted and modelled just like normal agents. For bootstrapping purposes, the agent platform has a reference to a basic locator service. By querying this service, an agent can find other services, like a more advanced discovery service, a database service, etc.

The implementation of agent communication is based on *channels*. A channel is a two-way connection between two agents over which messages flow. The model is comparable to that of network communication with sockets, although at a higher abstraction level : the unit of communication here is a message. The ports of an agent can be connected to a channel. Performing a send operation on a port means transmitting it over its attached channel. The delivery of the message happens asynchronous. It is however possible to wait for an answer over the channel, simple by performing an `accept` operation on the port. Because messages always arrive or are send through a specific channel, the notion of conversations is more directly present in the model. A complete conversation between two agents, regarding a particular service, is described by all the the messages that have been send back and forth over that channel. The relation between successive mes-

sages is also immediately clear.

How the communication channels actually behave is up to the agent platform. One channel could for instance only provide local communication whereas another channel would implement location transparency. The platform can select between the different implementations on the basis of the agent reference that is provided in a channel setup request. The platform has complete control over what the agent references are, from the viewpoint of an agent, they are opaque. This makes it possible to set up agents as services that can only be accessed locally, simply by issuing the correct kind of agent reference. When a reference to such a service would leave the agent platform, it is no longer valid : it can no longer be used to set up a connection to the service it is pointing to.

4 Implementation Issues

We have implemented a prototype of this model in Java², as a set of interfaces and abstract classes. Together with concrete implementations and a factory, a real instantiation of a platform is made that has a specific behaviour (e.g. only a local platform, a single hop platform, or a platform supporting general mobility).

In this prototype, basic protection of the agent platform and the individual agents is provided through language protection [12]. The platform and the agents are each loaded into a separate (class loader) name space. In a shared name space, the interfaces are defined through which exchange of objects can happen. As each of the agents only has access to classes in the shared name space, they cannot manipulate the platform or the other agents directly.

Migration of agents is implemented as a service, but needs come cooperation of the platform implementation to for instance manage agent references. The agent implementation is responsible for the serialisation of its state. The migration service receives an already serialised version that is ready to be migrated. Strong migration of agent can be supported by using a byte-code transformer [11]. A local scheduler within the agent is able to stop all the agent's threads, and capture their execution state. The byte-code transforming approach has the benefit that agents that have no need for strong migration, do not suffer the performance loss of its support.

The network communication infrastructure is provided by a gate service on the platform. This service can hide all details of (network) protocols and location of remote agent platforms. As all remote communication passes through this service, network security (such as integrity, confidentiality, ...) can be implemented and enforced in one place. A framework providing such facilities has already been designed [3]. It should be fairly easy to integrate it into the gate service.

The implementation of agent references that provide location transparency uses this gate service to communicate

to remote agents. However, this use is abstracted away for the agent. Messages to remote agents are transparently forwarded to the gate service. The gate service delivers the message to the gate service of the remote platform that currently hosts the target agent. This gate in turn forwards the message to the target agent. The way the current location of a remote agent is attained is also hidden. Both forwarding and a distributed location service is possible.

5 Adding Security

The model as presented above has been designed in such a way to enable plugging in security mechanisms very easily. Although this paper does not focus on the security issues in agent systems, we will briefly discuss two points where security can be plugged into an agent. As services of a platform are also agents, the same approach can be used to protect the services.

Firstly, we need security in the communication with other agents. As agents communicate through ports and the channels that connect to these ports have an explicit set up phase, the ports are the best place to add communication security. More specifically, an authentication and access control filter can be added around the ports of an agent when a channel is established. Alternatively, access control can be added as a filter over the arriving messages, in case finer grained control is desired. Other security requirements for communication (signing, logging, ...) can equally be implemented as filters over the ports.

A second point that needs security is the data an agent is carrying with. Because in the agent model an explicit storage container is defined, adding security is simplified. A wrapper placed around the storage object takes care of the protection by applying cryptographic algorithms to the data to be stored or read. A security policy for the data protection wrapper is formulated based on (parts of) the index under which the data is stored. Contract information could for instance be signed by the local agent server, such that later changes can be detected.

The above filters are in fact meta-entities, and use a very simple meta programming facility provided in the agent implementation. The model looks very similar to composition filters [1], but is here tailored towards agent communication instead of generic object-orientation. As the filters are described in terms of the entities in the agent model, they are independent of the functionality of an agent and can be used with any agent through composition and configuration. For instance, access control could be enforced on a particular service, simply by declaring that a filter implementing a specific access control mechanism be applied to the port that belongs to the service in question. Or product information gathered by the agent could be integrity protected by stating that a signing and verification filter be applied to all data that is stored under a specific key.

²After a proof of concept implementation in Python

6 Developer Support

Currently, a lot of research is going on in the area of software engineering for agent-oriented applications (see [13]). We do not strive to come up with an alternative methodology. In this section we merely point out that because of the explicitness of the proposed model, some additional support can be build into state of the art tools. To illustrate this, some possibilities are presented that can help a developer in the construction of the internal structure of an agent.

6.1 Agent Programming and Composition

The explicitness of the presented model has the disadvantage that developing tasks for an agent is more elaborate. Instead of simply using the available Java language constructs, the developer has to use the API's of the agent model to do seemingly simple things. For instance, information should not be just stored as data members somewhere in the implementation of a task. Instead, an explicit call to put it the storage container must be made. By promoting the concepts of the model to language constructs and adding these to the Java language (with the help of a preprocessor), the extra complexity can again be hidden for the programmer. In the end, a new agent programming language could be defined based on a number of such constructs. This ultimately could lead to a more direct version of agent-oriented programming, offering real abstractions of the agent model, instead of their translations to basic programming languages.

At a higher level, end-users will want to send out agents that act according to their wishes. The composition support in the agent model can help them create custom agents without the need for programming skills. One example is a buying agent. The user would select a product discovery task, configure it to look for the product she wants, and combine that with a negotiation task, selected from a range of negotiation tasks that each implement a different strategy. Of course, to be really useable for end-users there would have to be some graphical user interface, and also a information base of tasks, ordering tasks in categories. This would for instance prevent illegal combinations of tasks.

6.2 Patterns of Communication

The FIPA agent standardisation effort [4] defines a increasing number of agent applications, based on a description of possible conversations between agents. These descriptions can be seen as a form of extended message sequence charts [9], from the viewpoint of either of the agents involved. The agent sends messages of a specific type, or expects to receive messages of a specific type, and this in some predefined pattern. Such a contract-like description [8] can be used to define a looser way of interaction between agents. The description could for instance be used to describe how an agent in a certain role is allowed to use a specific service.

Eventually, a kind of (loose) type system can be defined, relating roles of agents to allowable interactions. Because of the explicit representation of conversations, we believe the presented agent model can be extended to include such a type system.

The description of interactions can also help a developer implement the interaction with other agents. For instance, unexpected messages can be discarded as a matter of precaution, to guard against malevolent agents. With the proper tool support, such implementations can even be generated automatically: message sequence charts can be transformed into state diagrams, where every state change corresponds to the reception or sending of a message. From the state machine specification, an implementation can be automatically generated, for the target programming language. This implementation can be attached to the port of the agent in question.

7 Conclusion

This paper presented our model of an agent system, which contains the concepts of agent software in a more explicit way. This results in more flexibility when designing and implementing agent applications. Firstly, agents can easily be composed from a number of functional units. Secondly, communication between agents corresponds more to the idea of conversations, or two-way interaction patterns. Thirdly, the agent platform is kept very minimal, providing only basic communication possibilities. The services are defined as agents. Lastly, security requirements for agents can be plugged in without the need to change the implementation of the real functionality of the agent. To conclude, some examples of extra developer support in the form of tools or in the form of model extensions are given.

References

- [1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa. Abstracting Object Interactions Using Composition Filters. *ECOOP'93 Workshop on Object-Based Distributed Programming*. Kaiserslautern, Germany, July 1993.
- [2] G. Cabri, L. Leonardi, F. Zambonelli. MARS: A Programmable Coordination Architecture for Mobile Agents. *IEEE Internet Computing*, 4(4):26–35, July-August 2000.
- [3] B. De Win, J. Van den Bergh, F. Matthijs, B. De Decker, W. Joosen. A security architecture for electronic commerce applications. In *Proc. SEC 2000*, Beijing, China, August 2000.
- [4] Foundations for Intelligent Physical Agents. Web Site at <http://fipa.org>.

- [5] W. Jansen, T. Karygiannis. Mobile Agent Security. NIST Special Publication 800-19, August 1999.
- [6] N. Karnik and A. Tripathi. Agent Server Architecture for the Ajanta Mobile-Agent System. In *Proc. PDPTA '98*, Las Vegas, NV, July 1998.
- [7] D. B. Lange, M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. August 1998, Addison Wesley Longman.
- [8] B. Michiels, B. Wydaeghe. Pattern Contract Systems: Using Patterns for Component Composition. In *Proc. EuroMicro '99*, Milan, Italy, September 1999.
- [9] James Odell, H. Van Dyke Parunak, B. Bauer. Representing Agent Interaction Protocols in UML. In *Agent-Oriented Software Engineering*, Lecture Notes in AI 1957. 2001.
- [10] S. Poslad, P. Buckle, R. Hadingham. The FIPA-OS Agent Platform : Open Source for Open Standards. In *Proc. PAAM 2000*, Manchester, UK, April 2000.
- [11] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proc. ASA/MA 2000*, Zürich, Switzerland, September 2000.
- [12] D. Wallach, D. Balfanz, D. Dean, E. W. Felten. Extensible Security Architectures for Java. In *Proc. 16th SOSP*, Saint-Malo, France, October 1997.
- [13] M. Wooldridge, P. Ciancarini. Agent-Oriented Software Engineering : The State of the Art. In *Agent-Oriented Software Engineering*, Lecture Notes in AI 1957. 2001.