Deploying BDI agents in open, insecure environments

Rogier C. van het Schip, Martijn Warnier, Frances M.T. Brazier

Section Systems Engineering,
Faculty of Technology, Policy and Management,
Delft University of Technology,
The Netherlands
{m.e.warnier, f.m.frances}@tudelft.nl

Abstract. Secure deployment of agents in open, insecure environments is a challenge, for which a number of multi-agent system (MAS) frameworks have been designed. Secure deployment of BDI-based MAS in such environments, however, has yet to be addressed. This paper proposes an architecture to securely support large-scale, heterogeneous, BDI-based multi-agent systems, using Jason and AgentScape to illustrate the approach. An example scenario in which BDI agents negotiate the price of electricity in an open energy market sets the stage.

1 Introduction

One of the most predominant modeling paradigms within multi-agent design, is the Beliefs-Desires-Intention (BDI) [5, 12] paradigm. BDI agents explicitly reason about other agents and their dynamic environments. To date, however, most BDI agents do not often reason about their own security in open, insecure environments.

There is no fundamental reason why BDI languages cannot be supported by agent middleware designed to this purpose [8]. This paper proposes an approach to support a BDI system with an agent middleware platform that provides secure support for large-scale, heterogeneous, multi-agent systems.

This paper addresses the integration of a BDI system with an agent platform, Jason [3, 4] with AgentScape [19], thereby allowing Jason agents to run on AgentScape middleware. The new integrated system extends Jason's functionality with Web service access [21], agent identity management, secure and anonymous communication [28] and access to configuration [20] and directory [27] services using an agent platform built for security [24]. Jason adds BDI-like agent programming to AgentScape.

Most BDI agent systems such as Jadex [23] and 2APL [7] are supported by the JADE [1] agent platform. The Jason system is also be supported by JADE, however JADE has some limitations which make secure deployment in large open environments difficult. See Section 2 for more on this issue.

The next section discusses the scientific and engineering contributions of integrating both systems, before describing both the Jason and AgentScape systems

in Section 3. The following section details the design of the integrated system and AgentScape's expanded functionality is illustrated in Section 5, using a scenario of Jason agents negotiating the price of electricity in a large-scale, open environment. This paper ends with conclusions and suggestions for future work.

2 Contributions

Autonomous agents are widely regarded as the next step in Grid resource negotiations or autonomous Web services [10, 22]. Such autonomous agents can be designed to reason using cognitive structures, such as goals, beliefs and plans. To enable the use of these cognitive agents in such open, large-scale and, inherently, insecure settings, platforms are needed that support these agents in their operation.

To support this operation, the integrated agent platform must meet several requirements:

- The platform must be scalable, in order to support tens of thousands of agents.
- Agents running on the platform must be secured from other, possibly malicious, agents.
- The cognitive reasoning of such agents must be supported.
- Because of the openness of the platform, agents must be allowed to operate anonymously in order to, for example, be protected against *buyer profiling*. This also provides additional security against malicious agents.

To enable the use of such platforms, this paper discusses an engineering approach to this problem. A system is described that supports secure deployment of BDI agents in open, large-scale environments, which by supporting Jason BDI agents on the AgentScape middleware platform.

The existing integration of Jason into JADE supports several aspects of secure deployment of multi-agent systems. However, this paper argues that this integrated system is not yet suitable for the deployment of BDI agents in open, large-scale systems: JADE features a global lookup service [2]. Local caches of this service are used to cover most requests for information, but the global lookup cache must be consulted for unknown information. Given enough requests, this global service becomes a bottleneck to the operation of the agent platform. AgentScape's Foncation [20] lookup service operates distributed, eliminating this bottleneck to allow multi-agent systems to grow to truly large scale.

In addition, JADE features an Agent Management System (AMS) that manages the operation of a (JADE) agent platform, such as creation and deletion of agents. It also controls use of and access to the platform and maintains a list of agents running on the platform and agents must request administrative actions from the AMS [2]. Such a single AMS forms another bottleneck for large scale deployment of the JADE system.

Finally, by default JADE has no support for security. And while it's possible to use extensions to JADE, such as JADE-S [17] and SAgent [14], these extensions are outdated and are no longer deployed in practice. They also lack certain security features, in particular agent anonymity.

The most important functional requirements for integration of the Jason system with the AgentScape platform are:

- Jason agents must be able to perform all operations to which they are accustomed, in the same way as usual. Backwards compatibility with existing applications must be maintained.
- New operations for Jason agents should be implemented as extensions to the current framework.
- Jason agents are provided the same support as all other AgentScape agents: secure communication with other (non-Jason) agents, access to internal services (e.g. web service access gateway, anonymity service). This also means all message sending and Web service access must be done through the platform, to allow the platform to monitor agents and their resource consumption.

Jason agents should be able to call the same (Jason) operations as before. Custom internal actions, which allow Jason agents access to (non-Jason) program code, are used to extend the functionality provided to Jason agents. Access to the webservice gateway, for example, should be (and are) implemented as a custom internal action, using familiar syntax. The Agent API should not be (and has not been) adapted.

3 Background

This section briefly describes the Jason and AgentScape systems.

3.1 Jason

Jason [3, 4] is an interpreter for an extended version of AgentSpeak(L). AgentSpeak(L) is a high-level programming language for the design of cognitive agents and was developed by Rao in 1996 [25]. It is based on the Beliefs-Desires-Intentions (BDI) theories of Bratman [5] and the Procedural Reasoning System [12], designed by Georgeff. Jason adds several features to AgentSpeak(L), such as plan failure handling, inter-agent communication and an extensible set of *internal actions*, which allow an agent access to Java programmed code.

The Jason system is modular and extensible, and runs on different *middleware platforms*. All of Jason's BDI reasoning is performed in a Jason agent's own *reasoning cycle*, as part of an agent's *BDI engine*. The BDI engine passes calls to middleware-dependent methods to Jason's second major component, the *infrastructure*. The infrastructure maps calls to the underlying middleware and back.

3.2 AgentScape

AgentScape [29] is an agent middleware platform designed for scalability, security, heterogeneity and interoperability. AgentScape supports multiple operating systems, e.g. Windows, Linux, Solaris and Mac OS X, and various programming languages, e.g. Java, Python and C. AgentScape provides FIPA compliant secure inter-agent communication [6] and weak agent mobility [11].

Agents run in *locations*. A location is a group of computers (*hosts*) that belong to a single administrative domain, coordinated by a *location manager*. Each host has its own *host manager*. An example of a number of AgentScape locations is depicted in Figure 1. As depicted, the hosts need not be homogeneous.

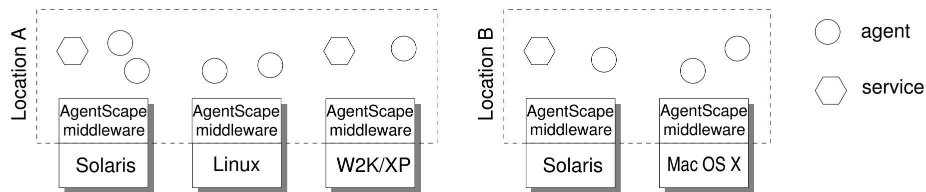

Fig. 1. AgentScape locations host agents and services. Each location can consist of multiple, possibly heterogeneous, hosts

AgentScape runs several internal middleware services, a number of which are depicted in Figure 2: location and host managers, agent servers and a web service gateway [21]. Note that in practice a host often runs several *agent servers*: at least one for each of the languages it supports. Internal middleware services include, for example, an internal lookup service, an anonymity service and a fault tolerance service In addition to internal middleware services, AgentScape provides external services such as distributed directory [20] services and configuration [27] services.

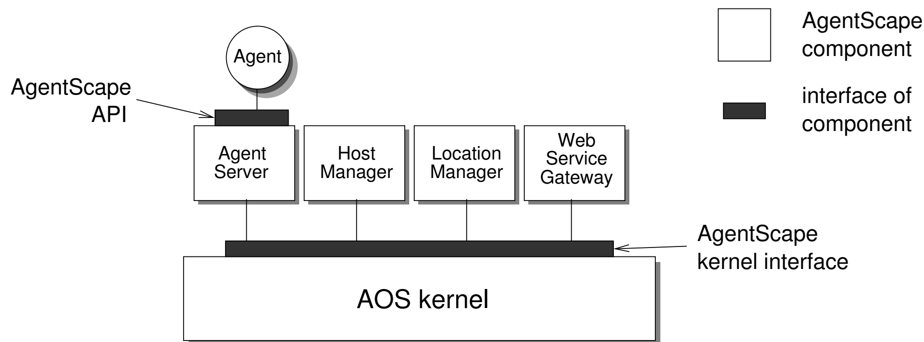

Fig. 2. The architecture of an AgentScape platform

Agents access AgentScape through the AgentScape API. This is basically the only requirement: there are no further requirements with respect to their design.

4 Architecture overview

As stated above, calls from a Jason agent's BDI engine that need to be handled by the underlying middleware are forwarded to an agent's infrastructure. For AgentScape to support Jason agents, this layer thus needs to convert Jason calls to AgentScape calls. The Jason infrastructure is, in fact, a partial infrastructure, as only two classes of a regular Jason infrastructure are needed in the AgentScape infrastructure:

– `AgentScapeAgArch`, the Jason agent architecture
– `AgentScapeRuntimeService`, supporting Jason runtime services

In effect, the entire Jason BDI engine operates as a complex agent inside AgentScape, using the infrastructure to convert its actions and decisions into AgentScape API calls, also see [26]. This result can be seen in Figure 3: the two classes of the infrastructure reside on top of the AgentScape API and provide AgentScape's functionality to the Jason BDI engine.

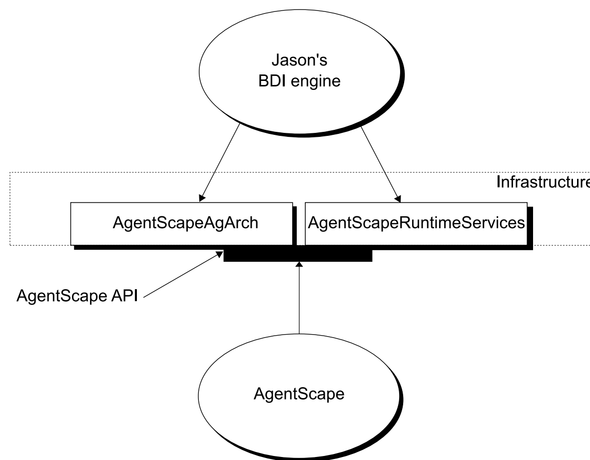

Fig. 3. The design of the integrated system

This design achieves a clear separation between AgentScape and Jason. An update or change to either system does not require major changes to the infrastructure[1].

[1] The system was designed for use with the Jason 1.1 version. During design, the 1.1.2 version of Jason was released. No changes to the infrastructure were needed.

4.1 Agent Architecture

In the infrastructure, one class represents a Jason agent architecture. It is known as the `AgentScapeAgArch`. This class implements the interface of a Jason agent, allowing it to pass messages to the BDI engine and send messages on behalf of the agent. To make calls to the AgentScape API, this class also extends AgentScape's Agent class. This provides the class with access to the AgentScape API and therefore to all of AgentScape's functionality.

The `AgentScapeAgArch` handles all functionality of a Jason agent architecture. As it is also registered as an AgentScape agent, it has access to all functionality offered by the AgentScape API. The `AgentScapeAgArch` starts and stops a Jason agent and handles inter-agent communication. For each internal action that requires middleware access, the `AgentScapeAgArch` provides a function. For example, to send a message, an agent's reasoning cycle calls its internal action `send`, which resides inside the BDI engine. In turn, this internal action calls the relevant method in the infrastructure, which is programmed to call the relevant method of the middleware used. The middleware handles the message, delivering it to the other agent, albeit a Jason agent or another agent.

`AgentScapeAgArch` also allows an agent access to new internal actions provided by the AgentScape middleware. For example, an AgentScape agent can request the name of its current (AgentScape) location, access AgentScape's Web service gateway, or access AgentScape's anonymity service, using custom internal action and a special function in the `AgentScapeAgArch`. For more details on access to the web service gateway, see Section 4.4.

4.2 Runtime Services

The second component of the infrastructure is known as the `AgentScapeRuntimeServices`. This class supports the `AgentScapeAgArch`. It provides runtime services to Jason agents. For example to start and stop agents, whenever required by the Jason BDI engine. As in the `AgentScapeAgArch`, the Jason BDI engine accesses these functions through internal actions. Despite not being an AgentScape agent itself, the `AgentScapeRuntimeServices` part has direct access to the AgentScape API. It works directly on the API and does not need to pass its calls through the agent architecture.

4.3 Agent communication

In both AgentScape and Jason, agents have names. AgentScape's naming system allows mapping names to unique agent handles, which is used by Jason agents to map their Jason name to one of their handles on agent startup. This allows Jason agents to send and receive messages to both their Jason agent names and AgentScape handles.

Jason supports broadcasting to all Jason agents. To this end, each Jason agent stores its handle along with a reserved string in the AgentScape lookup

service, identifying itself as a Jason agent. A special function in `AgentScapeAgArch` finds all entries of the reserved string and sends an AgentScape multicast to all AgentScape handles stored along with it, effectively broadcasting a message to all Jason agents in the system. It is also possible to broadcast messages to all agents, i.e., Jason and AgentScape agents. Sending messages between Jason agents and other (regular) AgentScape agents is also supported.

4.4 Web Service Access

In addition to its other functionality, AgentScape also supports Web service access to its agents. The `AgentScapeAgArch` has a `callWebService` method for calling Web services, which needs to be given the locations of the WSDL file and method stubs. The reasoning cycle can access this function by using a custom internal action `RunWebService`, which in turn calls the `callWebService` method in the `AgentScapeAgArch`.

An alternative design would have been to allow agents to directly use a Java Web service framework, such as Axis. However, it was decided to have agents contact Web services through the platform, for security purposes, as it allows monitoring of agents and their resource consumption. In AgentScape, agents are sandboxed. Providing unrestricted and unmonitored Web service access violates this design principle.

To use a Web service in Jason:

1. A WSDL file is used to create Web service stubs which can be called by agents. These stubs are created by a custom parser known as the `WS-stubber`.
2. As a Jason agent runs, the Jason BDI engine executes the code for web service execution. A sample AgentSpeak(L) plan can be seen in Figure 4. Once this plan is executed, the custom internal action `RunWebService` for Web service access is called.
3. This internal action passes the call on to the infrastructure; in Jason, calls from an internal action to the middleware are always given to the infrastructure to be executed. In this example, the `AgentScapeAgArch` has a `callWebService` function for handling Web services. This function is called next.
4. The `AgentScapeAgArch` accesses the stubs which were created at compile-time. This is possible because the `AgentScapeAgArch` is registered with AgentScape as an agent. It binds to these stubs, to be ready to call the service.
5. Next, the `AgentScapeAgArch` makes the call to the Web service. The called stub transports the call to AgentScape's Web service gateway, which functions as a proxy. This allows an AgentScape administrator control and monitor all agent Web service use.
6. The Web service gateway receives the results of its call and returns them to the stub, which hands them back to the `AgentScapeAgArch`.
7. Finally, the `AgentScapeAgArch` returns the results to the internal action. These are returned to Jason's reasoning engine, without additional overhead for Jason.

Note that the syntax for the use of the Web service internal action is no different for Jason developers than the use of regular internal actions. The calls necessary to use a service translate from Jason to AgentScape and back. In the body of a plan, the internal action is called. This, in turn, calls a function in the `AgentScapeAgArch`. The call is handed off and when the results return, they are parsed into Jason variables for use in the BDI reasoning engine.

```
ownCurrency("EUR").
foreignCurrency("USD").

+!convertCurrency :
   ownCurrency(Currency1) & foreignCurrency(
       Currency2)
   <-
   .println(
       "Calling currency convertor service.");
   as.RunWebService(
     CurrencyConvertor.wsdl,
     [Currency1, Currency2Ratio],
     Ratio
   );
   .println("Call successful. Conversion ratio:");
   .println(Ratio).
```

Fig. 4. This Jason plan checks the ratio between two currencies, using a web service

5 Scenario

This section details a scenario for negotiating Jason agents in an (semi-)open, insecure environment. Managing large-scale, decentralized systems in the energy market is a challenge [16]. Electricity is generated by many sources, including non-traditional sources such as private solar cells or wind turbines. Predictability of output to the power grid is becoming more difficult, possibly jeopardizing a reliable power supply, if not taken into account. A potential solution is decentralized markets in which autonomous agents regulate the power used by electrical appliances, small power sources or groups of the previous. The agents operate without human intervention, on behalf of their human owners. These human owners are both consumers and producers: selling power if their solar cells and wind turbines produce more power than their appliances use, or buying extra power if their usage is greater than their production. To enable continuous automatic negotiation, agents from both power consumers and providers run on an

AgentScape location, allowing them to either buy or sell the power their owner needs. The micro payments for these transactions are performed through a Web service.

As any party, consumer or provider, can send agents to an AgentScape location, the environment is completely open. In this setting, security is a major issue: once agents are exploitable and vulnerable to attacks, the affordable power supply of their owners cannot be guaranteed. In general, autonomous agents face two major threats: malicious hosts and malicious agents [9]. Malicious agents attempt to damage the operation of the platform or the agents running on it. As the discussed system is open to all parties, malicious agents are a realistic threat and must be countered. By default, AgentScape sandboxes all agents [24], preventing them from taking certain, possibly dangerous, actions. Although this limits the actions available to all agents, this is not considered a problem: the system is designed for inter-agent communication through negotiation. For example, sandboxed agents are denied access to the local file system. However, as the main goal of these agents is inter-agent negotiation, which is not limited by sandboxing, limiting the agents in this way does not damage this goal.

The second threat faced by agents is that of malicious hosts. In this scenario, the AgentScape middleware is considered to be trusted. This means the malicious host problem is not applicable.

To deal with the openness and insecurity of the environment, agent communication is anonymized. To perform their tasks agents need access to remote services. Several negotiation services are hosted by the platform. These aspects are discussed in greater detail below.

5.1 Agent anonymity

Agents, equipped with the usage profile of their owners, derive predicted energy needs of their owners and negotiate with other agents to get the resources they need. This communication must be done anonymously, for various reasons.

Primarily, anonymous communication protects the privacy of an agent's owner, to ensure his identity remains hidden. In addition, it prevents power suppliers from forming cartels and performing price fixing against a certain consumer: if the identity of a consumer is not known and (s)he uses a different identity for each negotiation with suppliers, they cannot form a cartel against him. Simply changing agent will not achieve this, a more complex anonymity plan is needed.

Finally, buyer profiling becomes impossible: using different identities for each negotiation prevents providers from creating a profile about a consumer. Having two different negotiations with a single supplier, using two different identities, means the supplier cannot create a profile about the consumer as it is not known if both negotiations were with the same agent. For these three reasons, communication within the open, insecure location must be done anonymously.

In the system, this anonymity is achieved by AgentScape's anonymity service [28]. It hides both sender and receiver identity, while also hiding the link between these agents, as this might also provide privacy information to outside

observers. This is achieved by using AgentScape handles, which agents can request as many as needed and cannot be traced back to the original agent: an agent sends a message to the anonymity service, together with the handle of the intended recipient. The anonymity service creates a new handle for the sender and forwards the message with the new handle as sender. If a response arrives, the handle is mapped to its original handle and sent to the correct recipient.

As the sender does not know which agent is behind the receiving handle, both sender and receiver identity are hidden. In addition, as these handles cannot be traced back to their original agent, they also achieve link anonymity as it cannot be observed which two agents communicated.

5.2 Web service access

Another central concept in this scenario is that of Web service access. Making micro payments requires access to the external world, which is achieved using an external Web service. These micro payments are used to pay for the resource transactions.

Using AgentScape's Web service gateway for this purpose ensures payments meet the platform's security policy, as they have to pass a check before being accepted and forwarded to the Web service. It also allows monitoring of the content and number of transactions by the platform [21].

5.3 Hosted services

To provide support for negotiation, the platform hosts various mediator [18] and auction [13] services to connected agents. Any agent that trusts the platform can accept these services as trusted third parties to mediate in their negotiations or to auction resources on their behalf.

6 Conclusions and Future Work

This paper presents a mechanism for secure deployment of BDI-based multi agent systems in open, insecure, dynamic, heterogeneous environments. In particular, agents written in the BDI system Jason can be deployed in such environments using the AgentScape multi-agent middleware. The resulting integrated system has been illustrated for BDI agents in a (semi-)open, insecure system, negotiating for resources on behalf of their owners in the context of a virtual energy market.

The functionality described in this paper is thus available to Jason agents running on AgentScape middleware. It includes Web service access and anonymized communications for agents. Also, the platform offers services for use by agents, such as directory, configuration, auction and negotiation services.

This extends the functionality provided by the other middleware platforms that currently support Jason. Heterogeneity and anonymity in inter-agent communication, are needed in many large scale open environments as illustrated above for management of energy markets.

Current work focuses on (1) further extension of Jason agents' functionality including additional functionality provided by AgentScape, in particular agent mobility and simplified agent startup, and (2) support for other BDI languages, in particular GOAL [15].

Acknowledgments

This project is partially supported by the NLnet Foundation `http://www.nlnet.nl`, partially supported by the ACCESS project, `http://www.iids.org/access`, part of the NWO TOKEN program and partially supported by the VU-star project.

References

1. F. Bellifemine, A. Poggi, and G. Rimassa. JADE – A FIPA-compliant agent framework. In *Proceedings of the 4th International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'99)*, pages 97–108, London, UK, Apr. 1999.
2. F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, NJ, Apr. 2007.
3. R. H. Bordini, J. F. Hübner, and R. Vieira. Jason and the golden fleece of agent-oriented programming. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 3–37. Springer, 2005.
4. R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. John Wiley & Sons, 2007.
5. M. Bratman. *Intention, plans, and practical reason*. Harvard University Press Cambridge, Mass, 1987.
6. J. Cucurull, B. J. Overeinder, M. A. Oey, J. Borrell, and F. M. T. Brazier. Abstract software migration architecture towards agent middleware interoperability. In *Proceedings of the 2nd Int'l Multiconference on Computer Science and Information Technology (IMCSIT)*, volume 2, pages 27–37, October 2007. ISSN 1896-7094.
7. M. Dastani and J.-J. C. Meyer. A Practical Agent Programming Language. *Proceedings of the fifth International Workshop on Programming Multi-agent Systems (ProMAS 07)*, 2007.
8. E. E.-D. El-Akehal and J. Padget. Pan-supplier stock control in a virtual warehouse. In N. e. Berger, Burg, editor, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)- Industry and Applications Track*, pages 11–18, 2008.
9. W. M. Farmer, J. D. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. In *Proc. 19th NIST-NCSC National Information Systems Security Conference*, pages 591–597, 1996.
10. I. T. Foster, N. R. Jennings, and C. Kesselman. Brain meets brawn: Why grid and agents need each other. In *AAMAS*, pages 8–15. IEEE Computer Society, 2004.
11. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998.

12. M. Georgeff and F. Ingrand. Decision-making in an embedded reasoning system. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 972–978, 1989.
13. P. Gradwell, M. A. Oey, R. J. Timmer, F. M. T. Brazier, and J. Padget. Engineering large-scale distributed auctions. In *Proceedings of the Seventh Int. Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. ACM, May 2008.
14. V. Gunupudi and S. Tate. SAgent: A security framework for JADE. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1116–1118. ACM New York, NY, USA, 2006.
15. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent programming with declarative goals. In C. Castelfranchi and Y. Lespérance, editors, *ATAL*, volume 1986 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2000.
16. G. James, W. Peng, and K. Deng. Managing Household Wind-Energy Generation. *IEEE Intelligent Systems*, 23(5):9–12, 2008.
17. N. Lhuillier, M. Tomaiuolo, and G. Vitaglione. Security in Multi-Agent Systems: JADE-S goes Distributed. *Special issue on JADE of the TILAB Journal EXP-in search of innovation*, 2003.
18. D. G. A. Mobach. *Agent-Based Mediated Service Negotiation*. PhD thesis, Computer Science Department, Vrije Universiteit Amsterdam, May 2007.
19. B. J. Overeinder and F. M. T. Brazier. Scalable middleware environment for agent-based Internet applications. In *Applied Parallel Computing*, volume 3732 of *Lecture Notes in Computer Science*, pages 675–679. Springer, Berlin, 2006.
20. B. J. Overeinder, M. A. Oey, R. J. Timmer, R. van Schouwen, E. Rozendaal, and F. M. T. Brazier. Design of a secure and decentralized location service for agent platforms. In *Proceedings of the Sixth International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2007)*, May 2007.
21. B. J. Overeinder, P. D. Verkaik, and F. M. T. Brazier. Web service access management for integration with agent systems. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC)*. ACM, March 2008.
22. S. Paurobally, V. A. M. Tamma, and M. Wooldridge. A framework for web service negotiation. *TAAS*, 2(4), 2007.
23. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. *Multi-Agent Programming: Languages, Platforms and Applications. Springer, Berlin*, 2005.
24. T. B. Quillinan, M. Warnier, M. A. Oey, R. J. Timmer, and F. M. T. Brazier. Enforcing security in the agentscape middleware. In *Proceedings of the 1st International Workshop on Middleware Security (MidSec)*. ACM, December 2008.
25. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. *Agents Breaking Away (LNAI 1038)*, 1996.
26. R. C. van het Schip. Integrating Jason into AgentScape - Joining BDI-theory with Agent Technology practise. Master's thesis, Vrije Universiteit Amsterdam, October 2008. MSc thesis report, Supervisors: dr. Martijn Warnier and prof. dr. Frances Brazier.
27. S. van Splunter, F. M. T. Brazier, J. Padget, and O. Rana. Dynamic service reconfiguration and enactment using an open matching architecure. In *Proceedings of the International Conference on Agents and Artificial Intelligence, Porto, Portugal*, January 2009.
28. M. Warnier and F. M. T. Brazier. Organized anonymous agents. In *the Proceedings of The Third International Symposium on Information Assurance and Security (IAS'07)*. IEEE, August 2007.

29. N. J. E. Wijngaards, B. J. Overeinder, M. van Steen, and F. M. T. Brazier. Supporting Internet-scale multi-agent systems. *Data & Knowledge Engineering*, 41(2-3):229–245, 2002.