SEWASIE
Semantic Webs and AgentS in Integrated Economies
IST-2001-34825

WP1
Task 1.1
Deliverable 1.6

SEWASIE Architecture Agentization

(Draft 0.2, 26/11/2003)

Abstract –
We describe in this document the basic guidelines for the specification of the SEWASIE architecture in terms of agents technology, and its implementation. We first present general properties of all agents, and the particular design of the different agents that build the system. Then, we identify the necessary agents features for SEWASIE, and evaluate currently available agent platforms in these sense.

Pablo Fillottrani
FUB

Document information

Document ID code	1.6		
Keywords	SEWASIE, architecture, agents		
Classification	**FINAL**	Date of reference	**26/11/2003**
Distribution level	**Sewasie Consortium**		

Editor	Pablo Fillottrani	FUB
Authors	Pablo Fillottrani	FUB
	Jürgen Dix	FUB
Reviewer	Enrico Franconi	FUB
	Sergio Tessaris	FUB
	Sonia Bergamaschi	UNIMO

Version history		
Date	*Version*	*Description*
3/10/2003	DRAFT 0.1	Initial draft
18/12/2003	DRAFT 0.2	Updated: Aachen decisions, available platforms evaluation

Copyright notices

Contents

1 Executive summary **4**

2 Relevant FIPA Standards **5**

3 SEWASIE Agents **9**
3.1 Query Agents 9
3.2 Brokering Agents 12
3.3 SINodes 14
3.4 Monitoring Agents 15
3.5 Communication Agents 16

4 SEWASIE Agent Environment **16**

5 Agent Development Platforms **18**
5.1 ADK 18
5.2 Aisland 19
5.3 April Agent Platform 20
5.4 Comtec Agent Platform 21
5.5 FIPA-OS 21
5.6 Grasshopper 22
5.7 JACK 23
5.8 JADE 24
5.9 Java Agent Services API 25
5.10 Evaluation 26

1 Executive summary

The goal of this document is to present the guidelines for discussing how to include the agents idea into the SEWASIE architecture. It is built upon all previous deliverables describing the global SEWASIE system, and each of its modules. No underlying technology is assumed. Further versions of this documents will incorporate specific definitions of points currently informally described.

The SEWASIE project aims at developing a FIPA[1] compliant trusted agent network [7, section 7.2.1], featuring completely open, scalable and secure-oriented architecture issues with the aim of making available the knowledge as synthesized in semantically enriched nodes of a network.

The SEWASIE architecture is not a general agent framework, and its main objective is not interoperability with other platforms, so only some of FIPA specifications are relevant to the project. In the first section we present these relevant standards. Then, SEWASIE architecture is described in terms of agents, and their necessary characteristics are presented. After that, we introduce the SEWASIE Agent Environment (SEA), the distributed software platform that provides these services for all agents. Finally, in order to provide the basis for agents and SEA implementations, several existing agent platforms are evaluated.

[1] Foundation for Intelligent Physical Agents.

2 Relevant FIPA Standards

We briefly introduce here the most important FIPA specification that are relevant for the SEWASIE architecture: the FIPA Abstract Architecture Specification, and the FIPA Agent Management Specification.

The FIPA Abstract Architecture Specification[13] identifies architectural abstractions, and their relationships, from agent systems. The main objective is to foster interoperability and reusability, but also by describing systems abstractly it becomes clearer how agent systems can be created so that they interoperate. From this set of architectural elements and relations one can derive a broad set of possible concrete architectures. We now present a high level description of the architecture and its main elements, and in the following sections we map the SEWASIE architecture to this elements and relationships. `Typewriter` font is used to identified standard elements of the architecture.

An `agent` is a computational process that implements the autonomous, communicating functionality of an application. An agent is the fundamental actor on any instantiation of the abstract architecture, which combines one or more service capabilities into a unified and integrated execution model. A `message` is an individual unit of communication between two or more agents. Typically, agents communicate using an `agent communication language`. Each agent has an `agent-name`, which is unique and unchangeable. A `service` is a functional coherent set of mechanisms in the agent platform that supports the operations of agents and other services. These services should not be confused by agent-provided services within instantiation of the architecture. Services may be implemented by agents, or as software that is accessed via method invocation. A `services-directory-service` is used to register and locate services within the FIPA infrastructure. Services include, but are not limited to: `message-transport-services`, `agent-directory-services`, gateway services, message buffering services, mobility services, etc (note that the latter services are not mandated in the specification). The message-transport-service is the service that supports the sending and receiving of messages. The agent-directory-service allows an agent to register its description so that other agents can communicate with it.

The FIPA Agent Management Specification[14] provides the normative framework within which FIPA agents exist and operate, through the definition of an Agent Management Reference Model. It establishes the logical reference model for the creation, registration, location, communication, migration and retirement of agents. Figure 1 shows the entities contained in the Agent Management Reference Model.

An `Agent Platform` provides the physical infrastructure in which agents can be deployed. It consists of the machine(s), operating system, agent support software, FIPA agent management components, and agents. The internal design of a platform is an issue for agent system developers, and is not subject to standardization. The concept of an agent platform does not mean that all agents resident on it have to be co-located on the same host computer. It is possible to envisage a fully distributed platform built around proprietary or open middleware standards. FIPA standardization efforts is con-

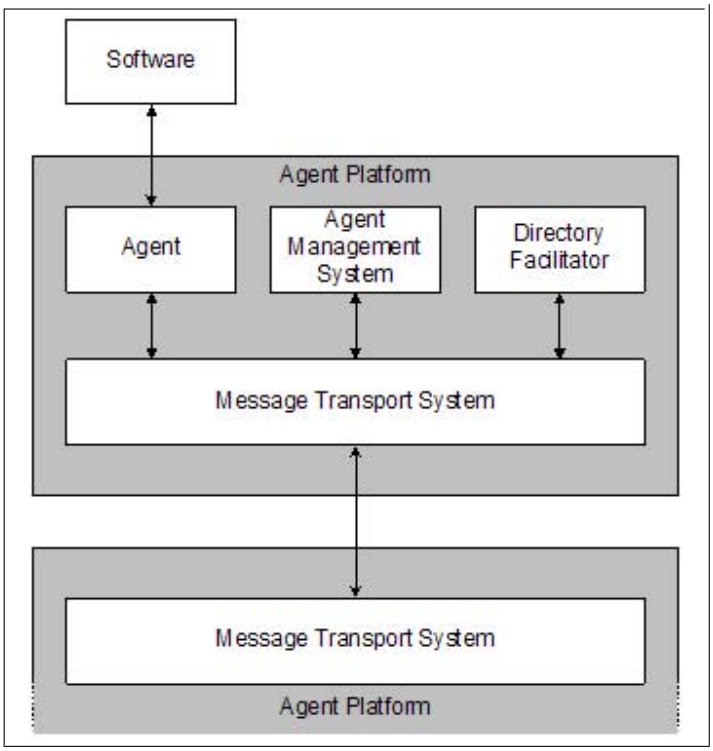

Figure 1: FIPA Agent Management Reference Model.

cerned only with how communication is carried out between agents who are native to the platform and agents outside it. Agents are free to exchange messages directly by any means that they can support.

The `Directory Facilitator` is an optional component of the platform. If present, it must be implemented as a `service`. It provides yellow pages services to other agents. Agents may register their services, or query the it to find out what services are offered by other agents. The `Agent Management System` is a mandatory component that maintains a directory of agent-names registered in the platform. It provides white pages services to other agents. It supports operations for registering, deregistering, modifying, and searching. Additionally, it can instruct the platform to terminate, create, suspend and resume agents. Both components are reification of `agent-directory-service`. The `Message Transport System` is the default communication method between agents on different platforms.

In summary, FIPA agents exist physically on an `agent platform` and realize their funcionalities with the help of the facilities offered in it. In this context, an agent, as a physical software process, has a life cycle that can be used to describe the states which it is believed are necessary, and the responsibilities of the `agent management system` in these states.

The life cycle of a FIPA agent (according to [14]) is shown in figure 2. It is characterized by the following states:

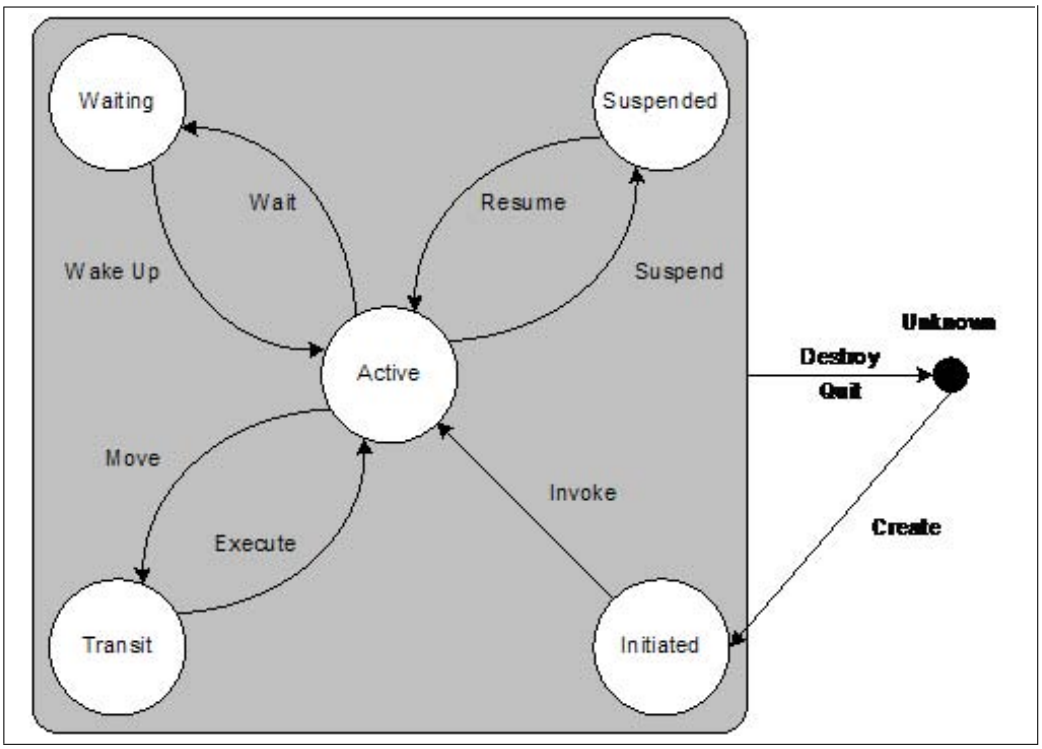

Figure 2: FIPA Agent Life Cycle.

1. `initiated`: the agent has been created, but has not received yet any message invoking its services.
2. `active`: the agent is executing tasks in response to one or more service invocations.
3. `waiting`: the agent has completed all activities involved in all the received service invocations.
4. `suspended`: the agent is waiting for some external conditions in order to continue executing its tasks.
5. `transit`: the agent is in the process of changing its location. Only for mobile agents.

This life cycle has the following characteristics:

- it is **bounded** by the `agent platform`: a static agent is physically managed within a platform, and its life cycle i therefore always bounded to a specific `agent platform`.
- it is **application independent**: the life cycle model is independent from any application system and it defines only the states and the transitions of the agent service.

- it is **instance oriented**: the agent described in the life cycle is assumed to be an instance which has unique name and is executed independently.
- it is **unique**: each agent has only one life cycle at any time, and within only one `agent platform`.

The `agent management system`, on behalf of the `agent platform` has the responsibility of message delivery to every agent in the `active` state, and to buffer messages to agents in the `initiated`, `waiting`, `suspended` and `transit` states. Buffered messages are delivered when the agent returns to the `active` state. In the case of mobile agents this may involve forwarding the messages to a new location.

The transitions between states of an agent can be described as:

- `create`: the creation or installation of an agent. This can only be initiated by the `agent management system`.
- `invoke`: the invocation of a new agent. This can only be initiated by the `agent management system`.
- `destroy`: the forceful termination of an agent. This can only be initiated by the `agent management system` and cannot be ignored by the agent.
- `quit`: the graceful termination of an agent. This can be ignored by the agent.
- `suspend`: puts an agent in a suspended state.
- `resume`: brings the agent from a suspended state. This can only be initiated by the `agent management system`.
- `wait`: puts an agent in the waiting state. This can only be initiated by an agent.
- `wake up`: brings the agent from a waiting state. This can only be initiated by the `agent management system`.
- `move`: puts the agent in a transitory state. This can only be initiated by the agent, and is valid only for mobile agents.
- `execute`: brings the agent from a transitory state. This can only be initiated by the agent management system, and is only valid for mobile agents.

In the cases when there is no specification, the transitions can be initiated either by the agent, or by the `agent management system`. Transitions initiated by the `agent management system` may be internally generated, or executed in response to requirements from other agents in the platform, or from external users of the platform.

3 SEWASIE Agents

In the SEWASIE architecture[7], each agent has a type that determines the set of services the agent provides to other agents, and the set of actions the agent can invoke in response to a service. These actions are capable of changing the data structures managed by the agent, and/or sending messages to other agents requesting services. The action an agent can take, as well as its action policy, must be clearly stated in some declarative language.

The SEWASIE system is a “type closed” agent system, in the sense that it is not expected to incorporate new types of agents, although it is possible to incorporate, and delete, new agents of the pre-defined types. We will informally present each type of agent by introducing first the services it provides to other agents, then describing its life cycle, and finally introducing the set of actions it supports. Some of the described services and actions are not needed in the narrow-deep scenario which is the starting point of the development plan, anyway they are included in order to give a complete picture of the agent structure of the system.

There are four basic types of agents described in the SEWASIE architecture: query agents (QA), brokering agents (BA), monitoring agents (MA), and communication agents (CA). SEWASIE Information Nodes (SINodes) might also be considered as agents, since they exhibit some patterns of interactions with agents of the previous types. SINodes heavily depend on BA, and provide the essential services for the work of QA. Therefore, we also include SINodes in the agent class.

User interfaces might be also considered as agents, since they interact not only with users but also with the other agents. The Query Tool (QT) user interface initiates most agent interactions in the SEWASIE architecture, but provides no services to other agents. A QT generates QAs from user requirements, and after receiving the results it may generate MA or CA. The QT must also interact with BA in order to generate the query. The QT is described in [24]. So the QT is the main client of agent services, but its behavior is only that of a client and not a peer of proper agents. Other user interfaces, like the ontology designer interface, have even simpler behavior. So we have chosen to keep user interfaces outside of the agent infrastructure of SEWASIE.

3.1 Query Agents

Query Agents are in charge of the global query execution strategy, namely addressing the initial BA, receiving SINodes and BA references, actually querying specific SIN Nodes and combining their results. The term “query” is to be interpreted as a general statement in a known intermediate query language, and includes information on the context of the user at the time of the establishment of the query[7].

A QA is the “motion item” of the system, and should be the only carrier of information among the user and the system. The following service is the main objective of a QA:

- **solve-query** to carry a query plus the relevant pieces of the user ontology/profile,

which may help the brokering agents qualify the semantics of the query, define a query plan, doing the query rewriting for specific SINodes and BA, and merge the results. These actions involve processing information from BA, and decide which further BA to contact. In order to fulfill this service the QA needs to know the query and the initial BA. These service should be invoked through an asynchronous messaging protocol.

This service is available only to QT and MA. Query language for the initial query plus the context information, language for results, and communication protocols are not yet defined.

The life cycle of a QA is initiated by an invocation of this only service, and it is finalized when delivering the results. Therefore, the life of a QA is attached to only one query processing. QA are instantiated by users for each request to the system, but also by other agents like monitoring agents. QA creation by users through the user interface is described in [24], and by monitoring agents in [5][2]. Monitoring agents requires querying directly by URI [5].

According to [7] “A typical interaction between a QA and a BA may imply that the BA will provide directions to relevant SINodes and information on SINodes content, or reference the QA to other BA. The QA will then move to such nodes to query them, or may move on to other BA to ask them for directions against.”. The impact of several physically moving QA, carrying partially constructed answers all over the network seems too expensive, and it is a potential overload on the communication protocols. It seems preferable that QA reside in a single Server mode, sending remote messages to BA and SINodes on other Server nodes. In this way, partial result data only returns over the network to the query agent, ensuring only one movement for each partial result.

It might be possible for a query agent to decide that the SEWASIE Server node it is residing in is overloaded, and therefore prefers to move to other Server node in order to improve query answering performance (load balance). It is not clear if this type of mobility would be desirable for the SEWASIE system. In this case, moving policies for the agents should be established. These could be supported and/or implemented by the underlying agent platform. Other adaptive capabilities could be incorporated into QA: missing BA and SINodes, delayed responses, unsatisfactory responses, etc.

Another possibility is to have a network of cooperative QA. Again, in order to improve performance, in the case of several BA yet to be consulted, the QA might decide to delegate part of its duties into other QA. Then, a hierarchical network of QA will be responsible for answering a single query. Data reconciliation in answers should be done when the delegated agent finishes it work, and it is subsumed by its ancestor QA. This option arises the problem of maintaining the consistency of BA and SINodes to be consulted. It is possible to introduce suitable protocols between query agents and their delegated counterparts, for example see [10, 19, 9, 4]. It is important to notice

[2] In [5] it is said that monitoring agents activate query agents through the standard user interface, but this seems to be contrary to the description in [7][section 5.2.6]

that the merging techniques for partial answers should allow this task decomposition. Currently developed techniques for query solving [21] do not support this possibility.

In summary, all states from the life cycle of FIPA agents might be possible for QA. The `create` transition is activated by the QT and MA, followed by an immediate `invoke` transition for the **solve-query** service. Typically, the agent works in solving this requirement, but may enter in the `suspended` or `waiting` states if necessary. No other service requirement should be delivered to the agent. In case of mobile QA, then also the `transit` state is necessary. Once all relevant BA and SINodes have been found and consulted, and the result is delivered, the agent automatically `quit` itself.

The following actions are combined in a QA to respond to a **solve-query** demand:

- **query-BA(BA)** given a query, this action expresses it in terms of the ontology understood by a BA, and presents the updated query to it. As a result, the BA may respond with a set of relevant SINodes, and a set of additional BA to consult. The QA should ask the BA the ontologies relevant to the query, rewrite the query according to these mappings, and send the updated query to the BA. In [20] it is described the response the BA should have in this case. The results obtained should update the internal state of the QA.

- **query-SIN(SINode,BA)** given a query, this action expresses it in terms of the virtual global view associated to a SINode, retrieves the answer, and merges it with previous results. Again, the rewriting of the original query in order to be understood by the SINode, and the merging of partial results are not yet defined. The SINode was informed to the QA by its managing BA, which is responsible also for providing the SINode ontology in order to carry out the rewriting process. The response of a SINode in this case is described in [20, 21]. The merging step could be done in this moment, or in the execution of **query-SIN** action, or in both. This merging strategy has to be decided, according to the types of the results from each SINode.

- **deliver-result** upon execution of this action, the QA decided that no more searching is necessary. If the partial results are merged in each **query-SIN** action, then this action trivially returns what has been previously constructed, and updates the query result manager. Otherwise, it builds the final answer based on the partial results obtained from each SINode, and also updates the query result manager.

All these actions also require knowledge of the initial query and the initial instance of QT.

A clear declarative description of these and all the other actions presented in the other types of agents is expected in future documents. The plan generation that executes a sequence of these actions is the main core of a QA. In order to build this plan, in [22, section 6.2.1] it is expected a cost estimation for the response of a query from each BA and SINode. This has yet to be defined. A trivial plan, assuming all BA and SINodes are always available, which exhausts the list of available BA for a given query, could be implemented for the first prototypes.

Since very many users may issue very many queries, QA should be “lean and mean”, in fact as small and fast as possible [7]. In these sense, query rewriting according to local ontologies (BA, or SINodes) could be delegated to the other agents (in particular BA), which are the actual owners of the ontologies. As an alternative, a query may be formed by a set of pairs ⟨ local-query, agent⟩ (initially consisting of only one pair), and the rewriting is delegating to the brokering agent that informs the new brokering agent or the SINode.

In [7, section 5.1.3.3] it is said that QA may initiate monitoring agents. It is not clear when should it happen, or if the monitoring agent is simply initiated by the QT user interface instead of a QA.

Therefore, a QA has the following properties:

- is attached to solve only one query. This query and the initial BA are provided by the QT.
- interacts with BA and SINodes in order to build the answer for its query.
- must adapt itself to the conditions on the network (missing BA, SINodes, or delayed responses from them).
- has the possibility of being mobile, in the case it is aware of the load in the host computer.
- exhibit no cooperation with other agents of its kind.
- finishes its execution after delivering the result.

3.2 Brokering Agents

Brokering Agents (BA) are responsible for maintaining the meta data about the SEWASIE network. This meta data may consist in the ontologies which are present in the underlying SINodes, and also information about ontologies in other BA. The proposed architecture of brokering agents is described in [22].

There are different roles for BA which depend on the business model of the company which deploys the BA. A company may establish a BA to manage access to its sources which it makes available via SEWASIE. A company specialized on information brokering may establish a BA that combines ontologies provided by several other BA.

In general, the following services should be offered by all BA:

- **manage-SIN(SINode)** the BA is selected to manage a SINode, or it is instructed to update the ontology of an existing SINode. The local ontology of the SINode has to be mapped to the ontology of the BA, and the BA has to send a feedback to the SINode. Also, the new ontology must be broadcasted to other BA. Therefore, this message invoking this service should be asynchronous. The response to this service is described as use case MK1 in [22, section6.1.1.1]. It is not clear who will require this service (the SINodes?, authorized users?).

- **receive-ontology(ontology, BA)** the BA is informed about the ontology of another BA. The received material is incorporated in the current mapping of the BA, and it is evaluated for forwarding to others BA. This forwarding process may take time to complete, so the invoking message of this service should be asynchronous. This service is described as steps 4-6 from use case MK2 in [22, section6.1.1.2]. The requiring BA expects some feedback from the validation of the ontology.

- **get-mappings(query)** a QA issues a query to the BA. The handling of this service is the main task of a BA. It is described as steps 4-6 from use case MP1 in [22, section6.2.1.1]. The BA responds with a set of mapping including other BAs and SINodes. Depending on the response time, this message can be invoked synchronously or asynchronously.

- **get-ontos(query, BA)** a BA is asked about the ontology it knows is residing in other BA. This knowledge is directly available in the requested BA, so the service can be synchronously invoked. This service is needed by QA to rewrite queries before asking the referred BA.

- **get-info-ontology** other SEWASIE agents (CA, QT, MA) request the ontology of the BA. This is described as use case MP2 in [22, section6.2.1.2]. As in the previous case, this knowledge resides in the requested BA, so the service could be synchronicly invoked.

The life cycle of a BA is initiated when an authorized user creates the BA. In the `invoke` transition, knowledge of existing SINodes and other BA should be immediately handled to the newly created BA, in order to build the local ontology. Being in the `active` state, a BA may receive messages for updating its knowledge (the first two services described), or for consulting its knowledge (the last three services described). Serving the former messages the BA it is said to be in *design phase*, while serving the latter the BA is in *query phase*. These two phases need not be strictly separated. So in the `active` state, the BA can accept services in both the design phase and the query phase. That is, requests to update its knowledge, and requests to inform it. In case of simultaneous requirements of different phases, the BA should define a policy for handling them. These agents are expected to be rather large and sophisticated, with the ability to serve requests quickly and to do some semantic relationships mapping [7].

The following actions can be executed in response to the previous services:

- **broadcast-ontology** this action may be taken when the ontology of the BA is updated, i.e. when services **manage-SIN** and **receive-ontology** are requested. It involves deciding which other BA could be interested in the updated ontology, and its packaging and sending. It is described as steps 1-3 in [22, section6.1.1.2].

- **find-relevant-SINodes** this action is taken when a query is submitted by a QA. It is outlined in [20, section 4], although the concrete approach has yet to be decided.

- **find-relevant-BA** this action is also taken when a query is submitted by a QA. It is described in [20, section 4].

- **deliver-answer** collects partial results from the previous two actions, packages them, and delivers the result to the QA. The details of this action are not yet defined.

A BA plan should include strategies to handle all incoming requests, and processing their feedbacks. Some cooperation with other BA could be used in the case of parallel updating of ontologies, although it is not clear how this can be accomplished. No other cooperative task seems appropriate for brokering agents.

In principle, a BA does not have to move from the SEWASIE node it was created, but should be able to cope with network problems[7]. Adaptation to faulty network conditions should be considered for this type of agents. Anyway, the load balance point mentioned in query agents also applies in this case. Moreover, it is necessary to analyze persistence for these agentes, that is how to save their sates when the hosting computer is shutting down. Migrating, or reflecting the state in a database could be done in this case.

3.3 SINodes

The SEWASIE Information Nodes (SINodes) group together several data sources, providing a logical node of information to the network. These nodes may spread over several machines, and have significant resources allocates. SINodes internal structure is described in [7, section5.1.2], including a SINode Query Manager module which is in charge with contacts with other agents.

The following services are available from SINodes to other agents:

- **solve-query(query)** accepts a query expressed in terms of the local ontology, processes it according to the available data sources, builds the answer, and finally delivers it. This service is only required from QA. The general techniques for query reformulation and query processing within one SINode is described in [21]. Depending on the response time, this message can be invoked synchronously or asynchronously.

- **get-info-ontology** similar to the service in BA, informs the global view of the data sources managed by the SINode. This service is required by those BA that are manager of the SINodes upon the establishing of the link, and may be required later when BA are updating their ontologies. The knowledge is directly available in the requested SINode, so this service can be synchronously invoked.

Once SINodes are created, in order to belong to the SEWASIE network, they should be related to one or more BA. This process may be manual, automated, or semi-automated. These details are not yet defined, although it is clear that the BA should

require the SINode ontology. Afterward, a QA that have become aware of the SINode through these BA, can contact it in order to solve its query.

From the point of view of agents, SINodes are not very interesting since they should autonomously reply all requests. In a similar way to BA, SINodes could incorporate some kind of persistence in order not to rebuild its entire ontology the hosting SEWASIE node is not available. Besides this feature, no other agent characteristic like cooperation, adaptation, or mobility seems necessary for SINodes.

3.4 Monitoring Agents

Monitoring Agents (MA) are responsible for monitoring information sources according to user interests which are defined in monitoring profiles. These user interests are defined as monitoring profiles, or are explicitly stated in the user interface. The monitoring agent filters monitoring information with respect to these profiles, and provides contextualized information to be presented to the user by the monitoring interface. To perform this task, MA regularly generates queries in the SEWASIE network. It is not clear if this is done by contacting the same QA for each query or by creating a new QA each time, although the latter solutions seems more adequate to the design of QA. Each monitoring agent contains a fixed internal ontology which is linked to higher level SEWASIE ontologies. The description of the architecture of MA is in [5].

The MA only serves the following request:

- **monitor(query, BA, monitoring profiles)** given a query, a user profile possibly with a local ontology, initiates the process of periodically retrieving data, and visualizing the results. It should be invoked by the corresponding user interface, by an asynchronous message. The execution initiates other user interfaces.

Depending on the business model adopted, a MA may be set up and maintained exclusively by a single customer (i.e. company). Alternatively, one MA can provide shared access for multiple customer interested in the domain. In either case, from the agent point of view, a given MA should interact only with QA, calling for the **solve-query** service. Like in the case of QA, the life cycle of a MA is entirely defined by serving only one of request of its service. The `suspended` state should be used while waiting between generating queries, and also while waiting for query results.

It might be possible for a MA to cooperate with others MA regarding similar queries. This should not be an a-priori agentization decision, but should be established as part of the monitoring strategy. Agent persistency should also be included in this strategy. The monitoring phase might overlap periods when the host computer in not powered, so saving and recovering the agent state is necessary. No mobility seems appropriate for these type of agents.

3.5 Communication Agents

CAs initiate electronic negotiations and conducts their initial steps, based on relevant results they receive from QAs. They are responsible for finding and contacting potential business partner, asking for initial offers, and ranking them. The human negotiator can then decide and choose the best offer to begin negotiating with support from the communication tool. Communication agents are described in [23].

The CA serves the following request:

- **initiate-negotiation(query)** a new CA is created by a QT based upon a query result. The CA compiles the request messages, and sends them to the companies that appear in the query result. In this step, it is possible to create an offer by extracting information about products specification and prices from catalogs. Thus, it is possible access the result manager for that information. Once the replies are received, the CA might check it for consistency, rank them by user-defined criteria, and notify the user.

- **monitor-negotiation** the user sends this request when s(he) wishes to delegate parts of the negotiation to the CA. The agent can be helpful by reminding the user that some information is missing or that a similar state in the negotiation has already been reached before.

Both these service are invoked asynchronously. Each MA life cycle is initiated by an `invoke` transition with an **initiate-negotiation** request. After serving this request, the MA may only serve one **monitor-negotiation** request for the same query before ending its execution.

As has been described, most CA interactions are with the user. Its services are invoked by the QT and other user interfaces, and the user is receiver of the results. Thus, CA are not active components in the SEWASIE agent infrastructure. Persistency should be included in this agents. No adaptation, cooperation with peers, interoperability with other platforms and mobility seems appropriate. The utilitarian behavior of CA should be included in its negotiation policies.

4 SEWASIE Agent Environment

The SEWASIE Agent Environment (SAE) is a FIPA compliant agent platform which mainly hosts the services of an Agent Management System. The whole life cycle of agents, including possibly mobility and persistency, is managed by this component.

Agent creation and invocation may be called by the SEWASIE User Interface component, or by other agents. Agent identifier is assigned by the SAE during creation, and once the agent is active it allows other agents to communicate with it. It contains the agent-name, and contact information. The agent-name should be implemented as a globally unique identifier that can be used as a unique referring expression of the agent.

One of the simplest mechanism is to construct it from the actual name of the agent and its home platform address, separated by the character. The contact information includes the a list of transport addresses where messages to the agent can be delivered. Also, it should be possible to compare agent-names, for example to let a query agent know whether a particular brokering agent has already been queried. Agents in SEWASIE might also be in the suspended or waiting state. Both imply stopping the execution thread of the agent, with the difference that waiting can be only initiated by the agent itself. It is not decided whether suspended agents is necessary in SEWASIE, but a waiting state could be useful for queries waiting only for a SIN Node response. In the case of mobility, it is not clearly defined yet in which way SEWASIE agents should support this feature (see the discussion in the description of the query agents).

Registration services are needed when finding is a given agent is still active. In SEWASIE, this may be useful in case of a query agent receiving from a brokering agent names of other brokering agents that have been deactivated. It might also be useful for a query agent to find the state of the given initial brokering agent Registration services are provided by the `agent management system` in the FIPA standards.

The SAE should also provide security to the agent-level operations. From the agent side, ensuring that agents can protect from its information and processes from external access, and from the system side allowing invocation of agent life cycle operations only to authorized users.

Message transport services are only needed in case of interoperability between platforms is desired. In SEWASIE, this is an essential characteristic of communication agents [23].

There is no need for additional services in SAE. In particular, Directory Facilitator services are not needed. Each SEWASIE agents knows the existence of other agents through the agents to which it is connected. Query agents are created connected to an initial brokering agent[7], and brokering agents know SIN Nodes in the moment these nodes are created, and each other by a P2P connection[20]. Since in SEWASIE the type of each agent is known from the operations that produced its reference, there is no need of agent service description. In this sense, knowledge about services of agents can be built into agent code. This increases the coupling between the code of agents, but it is preferable from the performance point of view. Thesaurus and dictionary services are also not needed since these are roles played by the ontologies within each brokering agent. In this sense, the SEWASIE system constitutes a (although basic) general agent platform providing its services in a distributed fashion, instead of monolithic replicated algorithms. Precisely this distributed organization of the knowledge is one of the objectives of the project.

The SAE should be implemented throughout several network sites, we call SEWASIE nodes. This raises the problem of replicated data, that should be addressed by one of the several techniques known in database community. Since the replicated data in each SEWASIE node only consists of the names of the registered agents, this is not expected to place much burden on the system. It is expected that the selected agent development platform copes with this problem in a transparent way to the agents.

Other agent features like mobility, security and persistence should also be delegated from the agents to the SAE. As it was described in the previous section, the requirements in this sense are not immense. The SAE could implement these features from scratch, or rely on services provided by the agent development platform. The first option makes the features more tailorable to specific needs, whilst takes more developing time due to its complexity. In the second approach we must adapt to the features available in the chosen development platform, but we obtain the benefits from loosely coupled components. Considering the diversity and maturity of existing agent development platforms, we think the latter approach is preferable. In next section we consider several possible agent development platforms for implementing the SAE requirements.

5 Agent Development Platforms

There exists a large number of approaches, toolkits and platforms of different scope and maturity. Most of the platforms considered come from the Publicly Available Implementations of FIPA Specifications [12]. Some other well known platforms have also been included. Evaluation is based exclusively upon the documentation available in each technology website. None of the platforms has been installed and tested empirically.

Each platform is evaluated according to the following criteria:

1. standard compatibility: support for FIPA specifications
2. agent features: like inter-agent messaging, mobility, persistency and security. For security we only consider features for intra-platform communication. Security in inter-platforms communication is not considered since interoperability is not necessary in SEWASIE.
3. availability: whether the technology is publicly available. In this point, we follow the guidelines outlined in [6].
4. documentation and development tools: we consider the availability of programmer and user documentation, and the presence in the product environment of tools that help in programming, debugging, deployment and management of agents.
5. practical applications: known projects that have been built using the platform.

Next we present the available agent platforms, and after that we analyze them for implementing the SAE.

5.1 ADK

The Tryllian´s ADK (Agent Development Kit)[1] is a mobile component-based development platform that allows to build distributed applications based on agent technology. It

is designed and build in JAVA, and supports XML, JXTA and JNDI. The ADK consists of two parts, the Agent Foundation Classes (AFC) and the Agent Runtime Environment (ARE). The AFC is an interface layer that contains all classes and interfaces needed to build agents. It encapsulates all the functionalities of the ARE, so access to this part is not normally needed for programmers. The product contains a tool for visually designing, implementing and deploying agents.

In ADK agents are implemented as JAVA clases, and have three distinct parts: body, knowledge, and behavior. The body is the part of an agent that takes care of executing tasks, sending messages, and moving the agent around a network. The knowledge part contains information the agent has about itself and its environment. Lastly, the behavior portion specifies which actions the agent will make and it is usually customizable for each agent.

Considering the evaluation criteria, ADK has the following properties:

1. standard compatibility: it supports FIPA 2000 standards.

2. agent features: communication between agents can be done according to several protocols. It supports X.509 standard for encryption, strong mobility (in the sense that agents can move between hosts during runtime, taking along their states and code), and persistency through a number of database software products.

3. availability: The ADK is available for purchase (prices are not available on-line, although third parties state that it is not a cheap product) or for free evaluation to educational institutions under a research license program. The current available version is 2.1. System requirements are Windows or Unix operating system, 70Mb free in hard disk and a 64Mb memory. Persistency requires database software installed.

4. documentation and development tools: There is very good documentation available on-line on both the AFC and the ARE. A JAVA programmer can easily adapt its code to use AFC. ADK development tools and modules are publicly available and do not require purchase of a license.

5. practical applications: it is claimed that a number of projects have been built using ADK, but no details are available. There is an ADK user community where developers may contribute with publicly available tools to be added to the environment.

5.2 Aisland

Aisland[2] is a hosted project inside Sun´s project JXTA for peer-to-peer systems. JXTA defines a set of protocols that can be implemented by peers to communicate and collaborate with other peers implementing these protocols. It provides a means of standardizing messaging systems. AIsland's goal is to provide a framework to build and distribute agents using JXTA protocols. The framework exposes objects (graphic module, audio

module, neural network, fuzzy logic ...) to an agent developer, who can glue these modules together using a scripting language, currently JavaScript. An Agent is represented by an XML document, and contains script code, which is interpreted by an AIsland (a JXTA peer) in a secure manner.

Considering the evaluation criteria, AISLAND exhibits the following properties:

1. standard compatibility: it does not support FIPA standards.
2. agent features: communication between agents is done using the transport module, but no documentation is provided. Weak mobility is possible (in the sense that agent may migrate when not executing something). Not all states in the FIPA agents life cycle are supported.
3. availability: the project is currently in the building prototype status. Source code is available, last release is number 0.1.2 from February 2003.
4. documentation and development tools: documentation is being developed, currently it is minimal. A graphical GUI for creating and managing agents is available.
5. practical applications: none, since the project is in an early stage of development.

5.3 April Agent Platform

The April Agent Platform[3] is a lightweight FIPA-compliant agent platform developed by the Imperial College and Fujitsu Laboratories Ltd., which has been written in April (Agent Process Interaction Language). Basically, it consists of the `agent management system`, the `directory facilitator` and the `message transport system` from the FIPA standards. These components are implemented as separate entities with their own thread of execution. Agents communicate in an interoperable fashion using FIPA ACL language over HTTP (other transport protocols are planned).

April presents the following properties:

1. standard compatibility: follows FIPA 2000 specifications.
2. agent features: communication is done through ICM (InterAgent Communication System) over HTTP.
3. availability: last version is 0.9.0 from September 2002. It is available for the Linux i386 operating systems under the GNU license. It requires the April programming language and the ICM software to be installed. The graphical interface requires the DialoX Graphical Environment.
4. documentation and development tools: available documentation is incomplete, some aspects of the platform are not described. No mobility and persistence. April includes a graphical interface to manage and control aspects of the platform and its functionalities.

5. practical applications: no application is described in the available documentation.

5.4 Comtec Agent Platform

Comtec Agent Platform[8] is a straightforward implementation of FIPA 97 Spec 1 (Agent Management), Spec 2 (Agent Communication Language), Spec 3 (Agent/Software Integration) and FIPA 98 Spec 12 (Ontology Service) from Communication Technologies Inc, Japan. Precisely its unique feature is the implementation of the FIPA Ontology Service specification. Java is the foundation of the platform, and Kawa (a Scheme interpreter written in Java) is used for building the agents.

Considering the evaluation criteria, Comtec has the following characteristics:

1. standard compatibility: some FIPA 97 and FIPA 98 specifications.
2. agent features: both inter-agent and inter-platform communication is based on IIOP. No mobility is supported.
3. availability: it is a freeware software, available for Windows 95, Windows 98 and Unix operating system. Latest known version is 3.0 from 1999.
4. documentation and development tools: minimal documentation in English. No development tool is documented.
5. practical applications: no application is described in the available documentation.

5.5 FIPA-OS

FIPA-OS[11] is an open source implementation of the mandatory elements contained within the FIPA specification for agent interoperability. In addition to supporting the FIPA interoperability concepts, FIPA-OS also provides a component based architecture to enable the development of domain specific agents which can utilize the services of the FIPA Platform agents. FIPA-OS is an experimental agent framework, originated from research at Nortel Networks Harlow Laboratories in the UK. It is fully implemented in Java. The aim of FIPA-OS is to reduce the barriers in the adoption of FIPA technology by supplementing the technical specification documents (available at Fipa.org) with managed open source code.

Considering the evaluation criteria, FIPA-OS presents the following properties:

1. standard compatibility: it supports the majority of FIPA 2000 standards.
2. agent features: inter-agent communication is done in ACL over IIOP or RMI. Secure communication is possible. Mobility has been implemented only in prototypes. Agents are provided with an ontology, representing the known classification of what exists, and how it exists, in a domain. It consists of a domain model and

domain querying facilities, that can use OIL, DAML, DAML+OIL, DAML-S inferene languages. Thus, it supports reasoning in the Belief-Desire-Intention model, and also Case Based Reasoning. Connections to databases is possible.

3. availability: FIPA-OS 2.2.0 of March 2002 is available through Emorphia Public License (EPL), for Windows and Unix platforms with Java 2 installed. It requires a Pentium 166Mhz processor, 64MB memory and 20MB free disk space.

4. documentation and development tools: good documentation, exemplars agents code is available. No development tool is documented.

5. practical applications: prior to public release, FIPA-OS was being used within European Collaborative projects: Cameleon and MAPPA. After that, it was used also in other collaborative projects: FACTS, SHUFFLE, CRUMPET, PATTERNS NGNi - SMONET, ONTOWEB. A pre-release of FIPA-OS has also been used in a number of Universities (notably Imperial College and EPFL). A small footprint version of FIPA-OS aimed at PDA´s and smart mobile phones has been developed by the University of Helsinki.

5.6 Grasshopper

Grasshopper[15] is a mobile agent platform that is built on top of a distributed processing environment. It is developed by IKV++ Technologies AG, Germany. It is built in Java 2, and a web plugin is available for multi-platform execution.

Considering the evaluation criteria, Grasshopper presents the following properties:

1. standard compatibility: it support both FIPA 2000 specifications and OMG MASIF standards by additional packages to the basic platform.

2. agent features: ACL is used for inter-agent communication, supporting synchronous and asynchronous messages, dynamic message creation, and multicast facilities over different transport protocols (sockets, RMI, IIOP), possible encrypted. It provides external security using X.509 certificates, as well as internal security based on the mechanisms provided by the JDK. Weak mobility is implemented by a transport service, built on top of the communication service. Strong mobility can be simulated. Persistency can be implicitly or explicitly specified for each agent, according to policies. It provides a set of additional agent concepts (like place, agency, region, etc) for the management and control of agents.

3. availability: Grasshopper 2.2 is distributed free of charge for non commercial use, over Windows and Linux platforms. This version was released in January 2003.

4. documentation and development tools: very good documentation. Visual development tools are also available.

5. practical applications: Grasshopper has been used in several research projects: ANIMA (Architecture Neutral Intelligent Mobile Agents), CAMELEON (Communication Agents for Mobility Enhancements in a Logical Environment of Open Networks), FACTS (FIPA Agents Communication Technologies and Services), MARINE(Mobile Agent Environment for Intelligent Networks), MARINER (Multi-Agent Architecture for Distributed-IN Load Control and Overload Protection) and MIAMI(Mobile Intelligent Agents for Managing the Information Infrastructure).

5.7 JACK

JACK Intelligent Agent[16] is an agent oriented development environment integrated in the Java programming language. JACK provides agent-oriented extensions to Java, in the sense C++ provides object-oriented extensions to C. JACK also provides a development environment and a set of library classes which support process of creating JACK applications. The JACK source code is firstly compiled into regular Java code before being executed. JACK is a software developed and maintained by Agent Oriented Software Pty Ltd, Australia.

JACK agents are built using the BDI (Belief, Desire and Intention) model: they have a set of beliefs to represent what the agent ”knows”; a set of desires/goals that represent what the agent is trying to achieve; a set of intentions to achieve the agent’s current goals; and finally a set of plans that are combinations of actions which achieve certain outcomes or respond to events, and are used by the agent to further its intentions. When an event occurs the agent: looks for relevant plans (plans that responds to this type of event); then, for each relevant plan, the agent examines its appropriateness to the situation the agent finds itself in; then the agent selects and starts executing the most appropriate plan found. Additionally the agent performs ongoing reasoning functions to decide what goal to pursue or alternatively what event to react to; how to pursue the desired goal; when to suspend/abandon the goal, or change to another goal.

The agent may also vary its balance between reactive and deliberative behavior by changing the amount of time allowed for deciding what to do next. This enables the agent to be more or less sensitive to changes in the environment, that is, be more or less ”committed” to its current plan.

Considering the evaluation criteria, JACK has the following features:

1. standard compatibility: JACK is listed in the publicly available implementations conforming the FIPA standards, but there are no available details of which standards are supported.

2. agent features: JACK agents can be organized into teams for modeling purposes or for performing actual joint tasks. JACK has a language specification and object oriented design targeted to allow easy extension for new agent models, such as recognition of intentions, or transaction-based agents. It also provides a lightweight and flexible communications model. By default JACK uses a fast TCP-

based protocol for communicating with other agents, back-end systems and GUIs. Other protocols can be layered on top of this or can replace the default protocol entirely. This can be used to provide secure communication between agents. No mobility is supported. Internal security based on the mechanisms available in the JDK.

3. availability: The latest release is JACK 4.1 from June 2003. JACK is implemented entirely in Java and runs on Java versions 1.1.3 and later. The graphical components require Java 2 v1.3 or later. JACK has been tested on the following platforms: Windows 95/98/Me, Windows XP/2000/NT 4.5, Solaris 2.5, Solaris 2.6, Solaris 7 (also called 2.7), Linux i386 (libc5 and libc6) and Mac OS X 10.1. JACK agents can be developed and deployed on any platform for which a suitable Java implementation is available. The JACK software package can be downloaded free of charge for an evaluation period of 60 days.

4. documentation and development tools: very good documentation. Exercises and some agents examples are also available. JACK includes a graphical development environment and a debugging tool.

5. practical applications: It is claimed that JACK agents are being applied in a broad scope of industries such as telecommunications, manufacturing, finance, air traffic management, aerospace, etc. No details are given.

5.8 JADE

JADE (Java Agent DEvelopment Framework)[17] is a software framework that supports through a middle-ware the implementation of distributed multi-agent systems with peer-to-peer networking. It claims to comply with the FIPA specifications, and provides a set of tools for the debugging and deployment phase. The agent platform can be distributed across machines (which not even need to share the same OS) and includes graphical tools supporting the deploying, debugging, management and monitoring phases. The configuration can be also changed at run-time by moving agents from one machine to another one, as and when required. JADE provides a homogeneous set of APIs that are independent from the underlying network and Java version, but also includes other APIs specific for J2EE, J2SE and J2ME.

JADE is completely implemented in Java language and the minimal system requirement is the version 1.2 of JAVA (the run time environment or the JDK). JADE is a trademark of TILAB (formerly CSELT) and it has been jointly developed by TILAB (Telecom Italia Labs) and AOT Labs at the University of Parma.

JADE presents the following properties:

1. standard compatibility: JADE is compliant with FIPA 2000 specifications, and also works with CORBA (Orbacus).

2. agent features: The communication architecture offers flexible and efficient messaging, where JADE creates and manages a queue of incoming ACL messages, private to each agent; agents can access their queue via a combination of several modes: blocking, polling, timeout and pattern matching based. The communication model has been implemented and its components have been clearly distincted and fully integrated: interaction protocols, envelope, ACL, content languages, encoding schemes, ontologies and, finally, transport protocols. The transport mechanism, in particular, is like a chameleon because it adapts to each situation, by transparently choosing the best available protocol. Java RMI, event-notification, and IIOP are currently used. Most of the interaction protocols defined by FIPA are already available and can be instantiated after defining the application-dependent behavior of each state of the protocol. SL and agent management ontology have been implemented already, as well as the support for user-defined content languages and ontologies that can be implemented, registered with agents, and automatically used by the framework. JADE has also been integrated with JESS, a Java shell of CLIPS, in order to exploit its reasoning capabilities. Supports weak mobility, secure communications and connection authentication.

3. availability: JADE is distributed in open source software license under the terms of the LGPL (Lesser General Public License Version 2). The latest official version is JADE 3.0b1 released on March 2003.

4. documentation and development tools: very good documentation. Tools for debugging, deploying and managing agents are available.

5. practical applications: recently, JADE has been integrated with the results of the LEAP project in a FIPA-compliant agent platform with reduced footprint and compatibility with mobile Java environments down to J2ME-CLDC, in collaboration with Motorola, ADAC, Broadcom, BT, and Siemens. Also, JADE is being used by a number of companies and academic groups, such as CNET, NHK, Imperial College, IRST, KPN, University of Helsinky, INRIA, ATOS and many others.

5.9 Java Agent Services API

The JAS (Java Agent Services) API[18] is a project for defining industry standard specification for APIs for deployment of agent platform-service infrastructures. It is intended to form the basis for creating commercial grade applications based on FIPA specifications. The project consists of a Java API (in the javax.agent namespace) for deploying open platform specifications that support the plugin of third party platform service technology. The API provides interfaces for message creation, message encoding, message transport, directory and naming. The design is intended to ensure that a JAS based system deployment remains transparent to shifts in the underlying technology. The project also delivers a reference implementation of the API, including sample services for RMI,

LDAP and HTTP. The project is carried out by several companies, like Fujitsu, Sun, IBM, HP, Spawar, InterX, Verizon, etc.

Following the evaluation criteria, JAS includes:

1. standard compatibility: it is an implementation of the FIPA Abstract Architecture within the Java Community Process (JCP) initiative. It also implements other FIPA standards like the FIPA Agent Communication Language Specification, the FIPA Content Languages Specification, FIPA RDF Content Language Specification, FIPA Message Transport Service Specification, etc.

2. agent features: JAS is a project for standard definition of agent APIs for the Java language, and the standarization process has not yet finished. Therefore, it is not possible to have a precise definition of the available services.

3. availability: the latest version of JAS was available for the public review process until May 2002. There is no information about further steps of this projects.

4. documentation and development tools: very little documentation is publicly available.

5. practical applications: none.

5.10 Evaluation

Standard compatibility is not a necessary feature, since a standard FIPA agent platform could be build on top of a non-standard lower layer. Anyway it is desirable in order to simplify agent implementation, and to improve understandability of agent features in the platform. In this sense, all platform except AISLAND supports some set of FIPA specifications. Comtec does not support latest standards.

The SEWASIE architecture does not require extremely advanced agent features, as has been described in previous sections. Secure communication, life cycle and white pages services, and some facilities for mobility and persistency are the most important characteristics. In order to facilitate the implementation of agents, the chosen platform must present simple and reliable services for these features. For example, a BDI complex model should be more an obstacle than an advantage for SEWASIE agent coding. In this item, ADK, Grasshopper and JADE seem to be the best choices.

Availability is a more important characteristic. The chosen agent platform should follow the requirements specified in [6]. Thus, it should be available both for Linux and Windows operating systems, integration with Java code should be easy, and the technology should be publicly available. Moreover, we prefer platforms that offer a fully functional product in a stable state (not a beta realease), but also are currently under further development or improvement, and that give support to potential users. In this sense, commercial platforms like Tryllian´s ADK and Agent Oriented Software´s Jack are the best positioned, but they do not comply to the publicly available software requirement.

Therefore, we consider FIPA-OS, Grasshopper and JADE as the best platform for this item. AISLAND and JAS have not produced yet a fully functional version, April doesn´t exhibit a seamless integration with Java code, and Comtec has not been updated since more than three years.

The chosen platform must be well documented, so that people that are not used to program agent code could use the platform services without much learning effort. Development tools for programming, debugging, deploying and managing agents are necessary not only for the agent code, but also for tuning several aspects in the SEWASIE architecture implementation. In order to provide a more accurate evaluation in this point, practical tests should be carried out, so we consider only the presence or absence of these tools in the platform. ADK, Grasshopper, JADE and Jack are the platform that present the best documentation, and some kind of tool for agent control.

Finally, the fact that a platform was selected for being used in other successful projects is a fact that confirms its quality as a software product. FIPA-OS, Grasshopper and JADE present documentation in this aspect, while commercial products like ADK and Jack only claim the fact.

Taking all these aspects into consideration, we consider the following platforms to the adequate for implementing the SEWASIE agent environmet and the SEWASIE agents:

1. JADE
2. Grasshopper
3. FIPA-OS

The platforms are listed in order of preference. The difference between Grasshopper and JADE is not significant, mainly in the set of available development tools. But since this evaluation was done only from the on-line documentation, we consider further empirical tests can be made in order to take a final decision. These tests should focus on the details of agent features and implementation, and usability of development tools.

References

[1] Agent Development Kit. `www.tryllian.com/technology/product1.html`.

[2] Aisland Project. `aisland.jxta.org`.

[3] April Agent Platform. `www.nar.fujitsulabs.com/aap`.

[4] T. Aura. On the structure of delegation networks. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[5] Andreas Becks. Specification of the architecture of the monitoring agent / visualization component. SEWASIE Deliverable D4.1, November 2002.

[6] Sonia Bergamaschi. Specification of the common platforms, languages, methods, and technological basis: first release. SEWASIE Deliverable D1.1, September 2002.

[7] Sonia Bergamaschi. Global architecture of the SEWASIE system. SEWASIE Deliverable D1.3a, April 2003.

[8] Comtec Agent Platform. `ias.comtec.co.jp/ap`.

[9] Yun Ding, Patrick Horster, and Holger Petersen. A new approach for delegation using hierarchical delegation tokens. In *Communications and Multimedia Security*, pages 128–143, 1996.

[10] J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman. On cooperation in multi-agent systems. *The Knowledge Engineering Review*, 12(3):309–314, 1997.

[11] FIPA-OS. `fipa-os.sourceforge.net`.

[12] Foundation for Intelligent Physical Agents. Publicly available implementations of FIPA specifications. `www.fipa.org/resources/livesystems.html`.

[13] Foundation for Intelligent Physical Agents. FIPA abstract architecture specification. FIPA 00001, 1996-2002.

[14] Foundation for Intelligent Physical Agents. FIPA agent management specification. FIPA 00023, 1996-2002.

[15] Grasshopper. `www.grasshopper.de`.

[16] JACK intelligent agents. `www.agent-software.com`.

[17] JADE. `jade.cselt.it`.

[18] JAS API. `www.java-agent.org`.

[19] T. Magedanz, K. Rothermel, and S. Krause. Intelligent agents: An emerging technology for next generation telecommunications? In *INFOCOM'96*, San Francisco, CA, USA, 24–28 1996.

[20] Zoran Majkić. General framework for query reformulation. SEWASIE Deliverable D3.1, February 2003.

[21] Zoran Majkić. Techniques for query reformulation, query merging, and information reconciliation - part a. SEWASIE Deliverable D3.2a, May 2003.

[22] Christoph Quix. Specification of the architecture of the brokering agent. SEWASIE Deliverable D4.1a, February 2003.

[23] Mareike Schoop. Specification of the architecture and conceptual design. SEWASIE Deliverable D5.1, November 2002.

[24] Sergio Tessaris. Specification of the tool for end-user query management. SEWASIE Deliverable D6.3, March 2003.