The SEWASIE multi-agent system

S. Bergamaschi[1], P. Filottrani[2], and G. Gelati[1]

[1] Dipartimento di Ingegneria dell'Informazione
Università di Modena e Reggio Emilia
Via Vignolese 905, 41100 Modena, Italy
[2] Faculty of Computer Science
Free University of Bolzano-Bozen
Piazza Domenicani 3, 39100 Bolzano, Italy
sonia.bergamaschi@unimo.it
p.filottrani@inf.unibz.it
gelati@dbgroup.unimo.it

Abstract. Data integration, in the context of the web, faces new problems, due in particular to the heterogeneity of sources, to the fragmentation of the information and to the absence of a unique way to structure and view information. In such areas, the traditional paradigms, on which database foundations are based (i.e. client server architecture, few sources containing large information), have to be overcome by new architectures. The peer-to-peer (P2P) architecture seems to be the best way to fulfill these new kinds of data sources, offering an alternative to traditional client/server architecture. In this paper, we envisage a multi-level architecture, with local nodes and communities strongly tied with a semantic context which is well defined and offers a globally integrated ontology to represent everything. At a wider level the relationships among distinct nodes are established by means of weaker mappings.

1 Introduction

Data integration in the context of the web faces new problems, due to the heterogeneity of data sources, to the fragmentation of the information and to the absence of a unique way to structure and view information. Internet can be viewed as a P2P data-sharing system with an enormous amount of data, where, to overcome information overload, it is necessary to develop new mechanisms allowing users to quickly understand and search for desired data. Designing such a mechanism is difficult, mainly because peers have different semantics and issues concerning the logical topology of peers and the semantical topology of the network (how data and metadata are related and distributed) have to be solved. It is no longer realistic to assume that the peers composing the system act as if they were a single data virtual source. We rather replace the role of a single virtual data source schema with a P2P approach relying on limited shared or overlapping vocabularies between peers. Our approach has brought to the definition of a new type of mediator system intended to operate in web economies, called the SEWASIE system. The idea underlying our proposal is that at local level things may be done more richly than at wider level. We envision a multi-level architecture, with local nodes and communities strongly tied with a semantic context which is well defined and offers a globally integrated ontology to represent everything. At a wider level the relationships among distinct nodes are established by means of weaker mappings.

Our approach has brought to the definition of the SEWASIE system architecture. We can think to the SEWASIE architecture as composed by a logical topology and a semantic network. The logical topology is realised through the adoption of an agent framework. Our reference implementation uses Jade, the popular agent/peer platform for building distributed systems. This has very important implication, as Jade is guided by the FIPA standards, the leading organisation in the field of agent technology specifications. On the top of this framework, we added a security layer, that allows to deploy Jade in firewalled networks. The semantic network is realised by agents offering services and interacting in semantically rich manners. In the SEWASIE architecture, each peer (or 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 have the side effect of changing the data structures managed by the agent, and/or sending messages to other agents requesting services. There are four basic types of agents in the SEWASIE architecture: query agents (QAs), brokering agents, monitoring agents and communication agents. The multi-agent system is a typed-closed agent system, in the sense that is is not expected to have functionalities uncovered by the defined type of agents. The logical topology and the semantic network have been designed to meet the functional requirements of the SEWASIE project. Other requirements arise when we have to deploy the system. Deployment requirements are usually specific to the particular installation to set up. One of the major issues we have to face in a number of cases is how to deploy a MAS in server environments, where the purpose is to avoid affecting existing system configuration as much as possible. For our MAS platform, this deployment requirement is met using a tunneling technique.

The paper is organised as follows. In Section 2 we describe the general arhcitecture of the SEWASIE system. In Sections 3 and **??** we detail the agent types we have defined and how they behave and interoperate. In Section **??** we report the implementaiton choices we have made. In Section **??** we present the SEWASIE system deployment architecture. Finally, in Section 8 we draw some conclusions.

2 General SEWASIE architecture

The SEWASIE system architecture[**?**] satisfying the is shown in figure 1.

Brokering agents (BAs) are the peers responsible for maintaining a view of the knowledge handled by the network. This view is maintained in *ontology mappings*, that are composed by the information on the specific content of the SINodes which are under the direct control of the BA, and also by the information on the content of other BAs. Thus, BAs must provides means to publish the locally held information within the network.

Query agents (QAs) are the carriers of the user query from the user interface to the SINodes, and have the task of solving a query by interacting with the BAs network. Once a BA is contacted, it informs the QA a) which SINodes under its control contain relevant information for the query, and b) which other BAs may be further contacted. Therefore, the QA translates the query according to the ontology mappings of the BA, and directly ask the SINodes for collection partial results. Also, it decides whether to continue the search with the other BAs. Once this process is finished, all partial results are integrated into a final answer for the user.

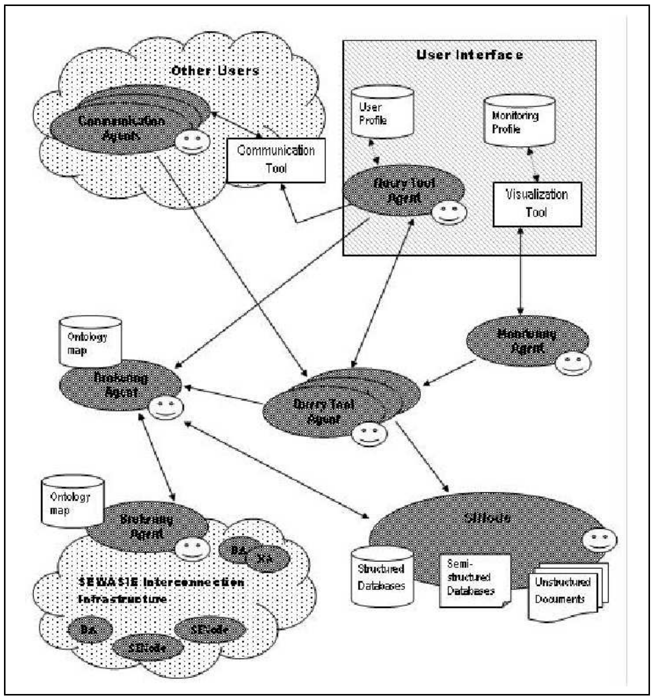

Fig. 1. The SEWASIE system architecture.

The SEWASIE information nodes (SINodes) are mediator-based systems, each including a global view of the overall information managed within. The managed information sources are heterogeneous collections of structured, semi-structured or unstructured data, e.g. relational databases, XML or HTML documents. SINodes are accessed by QAs in order to obtain data, and also by the managing BAs in order to build the ontology mappings. In order to create and maintain a global view of its information sources, SINodes require an ontology builder. This component performs in a semi-automatic way the enrichment process to create the SINode *ontology*. In turn, this SINode ontology is also integrated with other similar components into the BAs ontology mappings.

The user interface is a group of modules which work together to offer an integrated, easy to the user interaction with the SEWASIE system. It includes a query tool that guides the user in composing queries. In doing so, it requires the ontology of a *starting brokering agent* which helps in the interface presentation and behavior. Each instance of the query tool includes a Query Tool Agent (QTA) that is responsible to carry out communications with other agents in the SEWASIE system. In general, QTA are need to obtain the initial ontology from a BA, and also to create QAs that will solve the queries generated by the user.

Two extra elements of the user interface are the visualization tool and the communication tool. The visualization tool is responsible for monitoring information sources according to user interests which are defined in monitoring profiles. To perform this task, the visualization tool generates one *monitoring agent* for each topic of interest. Each monitoring agent contains a fixed internal ontology (so-called domain model) which is linked to higher level SEWASIE ontologies. Agents of this type regularly set up QAs, filter the results, and fill monitoring repositories with observed documents.

The communication tool supports negotiation between the user and other parties. Any query whose result includes contact information sets the context to launch the communication tool. This tool create several types of communication agents (CAs) that help in 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. CAs belong to one of the following four types:

– *Initiation agents*: that maintain the pre-negotiation phase for each search.
– *Filtering and Ranking agents*: that evaluate offers and ranks them according to given user preferences.
– *Resource Management agents*: that check the capacity of the negotiator and notify him by over-commitment.
– *Negotiation agents*: that conducts the negotiation when a well defined state is achieved, and the user releases the control.

In order to fulfill their tasks, initiation agents and filtering and ranking agents can create query agents to search for information in the SEWASIE network.

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 previously described pre-defined types. In next section we will describe the services and the life cycle of each type of agent. Once the agent platform is chosen, and the available tools are known, then these descriptions are the basis for implementing the agents.

3 Agent Descriptions

In this section, we detail the behaviour of each agent present in the SEWASIE system architecture.

3.1 Query Agent

Query Agents are in charge of the global query execution strategy, namely addressing the initial BA, receiving SINodes and BA references, actually querying specific SINodes 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[**?**].

A QA is the "motion item" of the system, and is 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 BAs to 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. This service should be invoked through an asynchronous messaging protocol.

This service is available only to QTA, MA and CAs. A specific query language for the initial query plus the context information, and the language for results, are defined in XML schema.

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. QAs are instantiated by users for each request to the system, but also by other agents like monitoring agents. In principle, it seems preferable that QAs reside in a single Server node, sending remote messages to BA and SINodes on other Server nodes, so mobility is not an issue for QA. Anyway, 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). These mobility could be supported and/or implemented by the underlying agent platform.

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

- **validate-query** when receiving a query, the agent must parse it in order to check whether it is well-defined, and to extract from it the information about the initial BA.
- **query-BA(BA)** given a well defined query, this action translates it in terms of the ontology understood by the 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 BAs 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 [**?**] 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 well defined query, this action translates it in terms of the virtual global view associated to a SINode, retrieves the answer, and merges it with previous results. The rewriting of the original query in order to be understood by the SINode, and the merging of partial results are defined according to the given semantics (see [**?**]). 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 [**?**,**?**]. 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.

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 Agent

BAs are responsible for maintaining the meta data about the SEWASIE network. This meta data consists in the ontologies which are present in the underlying SINodes, and also information about ontologies in other BAs. 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. Alternatively, a company specialized on information brokering may establish a BA that combines ontologies provided by several other BAs.

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.
- **receive-ontology(ontology, BA)** the BA is informed about the ontology of another BA. The received material is incorporated in the current ontology mapping of the BA, and it is evaluated for forwarding to others BAs. This forwarding process may take time to complete, so the invoking message of this service should be asynchronous. 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. The BA responds with a set of mapping including other BAs and SINodes.
- **get-info-ontology** other SEWASIE agents (CA, QT, MA) request the ontology mappings of the BA.

The life cycle of a BA is initiated when an authorized user creates the BA. In this process, knowledge of existing SINodes and other BAs 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.

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.
- **find-relevant-SINodes** this action is taken when a query is submitted by a QA. It is outlined in [**?**].
- **find-relevant-BA** this action is also taken when a query is submitted by a QA. It is described in [**?**].
- **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. Adaptation to faulty network conditions should be considered for this type of agents. 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.

4 SINodes

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 [**?**].

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 [**?**].
– **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 BAs that are managers of the SINodes upon the establishing of the link, and may be required later when BAs are updating their ontologies.

Once SINodes are created, in order to belong to the SEWASIE network, they should be related to one or more BA. This process is carried out in a semi-automated way using the ontology builder in the MOMIS system [**?**]. Afterwards, a QA that have become aware of the SINode through its manager BAs, 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.

4.1 Query Tool Agent

QTAs are the initiators of all search activities in the SEWASIE network. Likewise SINodes, they are not very interesting agents, since all their work consists in building messages, and waiting for results. No service is provided to other agents. They are created whenever a Query Tool is instantiated, and the first task is to find a suitable BA whose ontology is related to the user profile. Once the user defines the query through the query tool, these agents must translate the query into the internal language, pass it to a fresh QA, and wait for the results.

– **find-initial-BA(user-profile)** an initial BA must be selected, and if it is available, contacted in order to get its ontology mappings.
– **process-query** the user interface finishes the process of creating a query. First, a QA is created and the query is translated into the internal query language. The query is passed within a message to the QA, and an accept message from the QA is expected.
– **receive-results** results from a QA are received, processed and transferred to the query tool in order to inform the user.

4.2 Monitoring Agents

MAs are responsible for monitoring information sources according to user interests. The description of the architecture of MA is in [5] and D4.3. These user interests are defined as monitoring profiles, or are explicitly stated in the user interface. The monitoring agent monitors information according to its internal domain model and filters it according to the respective monitoring profiles of the user. To perform this task, MA regularly generates queries in the SEWASIE network, by creating new QAs each time.

The MA serves the following requests:

– **monitor(BA-ontology)** initiates the process of periodically retrieving data and inserting it into the contextualized document repository which is maintained by the monitoring agent.
– **filter(userID, monitoringProfile)** delivers the data that was retrieved and is new to the user with respect to his last filter request. Filter requests should be invoked by the corresponding user interface.

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 customers interested in the domain. In either case, from the agent point of view, a given MA should interact only with QAs, calling for the solve-query service. The life cycle of a MA is defined by serving one specific domain model. The suspended state should be used while waiting between generating queries, and also while waiting for query results.

4.3 Communication Agents

CAs are in general very simple agents, that respond to unique services. Initiator Agents are created when the user clicks on the corresponding button in the Query Tool interface that shows the result of a query. A message is received from the QTA, containing the result of the search in order to be parsed, and the contact information to be extracted. Then, these agents sends emails to the potential business partners. If at least one contacted user is registered, then the agent creates a negotiation process in the communication tool [**?**]. So it serves to services:

– **initial-contact(query-results)** the initial message for the agent. Extract contact information from query results, and sends the corresponding emails.
– **user-contacted(userID)** a potential partner accepts the registration into the communication tool, so this message is broadcasted through all active initiator agents in order to check if the user belongs to the corresponding query. In such a case, the user is added to a new negotiation process within the communication tool.

Initiator agents terminates after a given deadline.

A negotiation process is usually concerned with multi-attribute contracts. Filtering and ranking agents are used to help the user in the decision-making process of selecting the best agreement. One of this agent is created for each negotiation topic. It serves the following services:

- **set-preferences(negotiation)** the initial message for the agent. Options and preferences from the negotiation attributes are set up or changed through this message.
- **process(message)** the communication tool receives a message from a partner involved in this negotiation, and passes it to the filtering and ranking agent. This agent computes the utility function of the messages, filters it if the offer violates the constraints and ranks the negotiations of the process according to the latest updates.
- **ranking** a negotiation agent requires a current ranking of the messages in the negotiation process assigned to the filtering and ranking agent.

These agents are terminated once the negotiation process is completed, with or without contract.

Negotiation agents help the user in negotiating business contracts in cases where a fixed set of simple issues, i.e. price, delivery dates, etc, has to be settled. The negotiator communicates its negotiation strategy to the agent when it is created. The agent will create request/offer/counter-offer messages to partners according to this strategy. Negotiation protocols are described in [**?**]. This process is fully automatic, so the agent only need to implement the services:

- **initiate-negotiation(strategy)** this messages starts up the automatic negotiation process. The agent generates emails to partners, and process their responds. In case of unexpected conditions, the automatic negotiation terminates, and control is transferred to the user. In order to create the messages, this agent can consult the filtering and ranking agent, or resource monitoring agents assigned to the current negotiation process.
- **terminate-negotiation** the user wishes to regain control of the negotiation, so this message is send through the communication tool.

The agent is terminated when it receives this last message, or when the negotiation ends.

A company may be involved simultaneously in several negotiation processes. Resource monitoring agents helps to guard the company resources in order to avoid over-commitments. One resource monitoring agent is created for each business partner, that will continuously monitor the attribute values against the resource management system in the communication tool.

- **initiate-resource-monitoring(userID)** this messages starts up the resource monitoring process for a given user.
- **process-message(message**] the agent extracts information from the message, and compares it with the information stored in the resource management system. In case of over-commitment, it notifies the user by sending an email.
- **terminate-resource-negotiation** the user notifies the agent to terminate the resource monitoring process, so the agent will be deleted.

5 Implementation of agents in JADE

6 implementation

In this section we describe the implementation process in the first integrated prototype of the SEWASIE system. We recall that some of the components, like the communication tool, the visualization tool and the MOMIS system were already implemented as separate systems. Therefore, agent technology serves as an unifying framework for these existing components. We note that the current prototype has several SINodes but only one BA, so some interactions between agents are simplified.

Jade agents are implemented as a set of "behaviors". A behavior represents a task that an agent can carry out, and the platform provides a set of general behaviors already implemented. In order to make an agent execute the task implemented by a behavior, it is sufficient to add the behavior object to the agent´s behaviors list. Behaviors can be added anytime, when the agent starts, or within other behaviors. An agent can execute several behaviors concurrently. However, scheduling of concurrent behaviors is not pre-emptive in Jade, but cooperative. This means that when a behavior is scheduled for execution, its action method is called and runs until it returns. Therefore, it is the programmer who defines when an agent switches from the execution of one behavior to the execution of the next one.

Several behaviors are already implemented in the Jade library[?], ready to be reused by inheritance and redefinition of its action method. The simpler ones are the `CyclicBehavior` which never completes and executes the same actions each time it is called, and the `OneShotBehavior` which immediately completes after it is executed once. Jade also provides the possibility of combining simple behaviors together to create complex behaviors. For example, `SequentialBehavior` is a composite behavior that executes exactly once its component behaviors, or the `TickerBehavior` that executes the action repetitively after waiting a given period of time. We next show how these behaviors can be used to implement the Query Agent and the Brokering Agent in the SEWASIE system.

In the implementation of agent we also consider the standard FIPA Interaction Protocols. Each type of agent is involved in a limited set of interactions with other agents, so each one of these interactions is mapped into a standard protocol. In this way,

Next we describe some implementation issues for the most interesting types of agents: QA and t BA. In both cases, we first describe the interaction protocol in which these agents are involved, and then specify the set of behaviors that are activated along their life cycle.

In order to implement QAs and BAs based on JADE behaviors, we must take into account that these behaviors need in general to be dynamically created and deleted. Therefore, we need a "deliberative" behavior that does the control reasoning, deciding when and how the creation of other behaviors is done. These other behaviors will exhibit a typically "reactive" implementation:

they are activated when messages are received (the only sensor information SEWASIE agents have about their environment), and do some processing actions. This control architecture for agents is similar to some proposed hybrid architectures for robots[**?**]. The deliberative behavior is implemented through a subclass of `TickerBehaviour`, so it is activated at agent creation and never completes, being its job to add and delete reactive behaviors in the agent.

6.1 Query Agent

We stated that a QA has only one service, the **solve-query** service. This service can be called using the standard FIPA Request interaction protocol [**?**], being the initiator agent either a QTA, or a MA. The QA also require the **get-mapping** service from BAs, and the **solve-query** service from SINodes. In these cases, QA initiates Request interaction protocols, being the participant agent the BA or the SINode.

In addition to the previously mentioned deliberative behavior, we implemented the QA with the following reactive behaviors:

– a `CyclicBehavior` for receiving Request messages. The first request message the agent receives, is the one that is processed by the QA. Further requests are replied by not-understood messages.
– a `SequentialBehavior` for implementing query processing. Their sub-behaviors implement: query parsing, query schema creation, and BA contacting (the **query-BA** action in the QA, calling the **get-mapping** service in the BA). Some of this behaviors can be reused within the same sequential behavior, for example in the case of several BA contacting, but this is not currently available in this prototype. A composite behavior is needed in this case because normal execution of this actions is too time consuming, blocking the other behaviors in the agent.
– a `CyclicBehavior` to wait for the expanded queries from BAs, and, once they are received, doing the rewriting process and contacting SINodes (the **query-SINode** action).
– finally, another `CyclicBehavior` for processing SINodes responds (the **deliver-result** action). Since for default there may be several SINodes contacts, this behavior is cyclic in order to be able to process all of them.

6.2 Brokering Agent

BAs have to serve several services. For the **manage-SINode** service, the original SINode initiates a FIPA Propose Interaction Protocol[**?**], asking the BA for permission to be handled. In case of acceptance, the BA finishes the Propose interaction but also initiates a Subscribe Interaction Protocol with the SINode, in order to be communicated of every change in the SINode ontology.

As we already mentioned, the **get-mapping** service is called from QA through a Request interaction protocol. The **get-info-ontology** service is also contacted through Request interaction protocol where the BA plays the participant role.

The **receive-ontology** service is not implemented since it is not needed in the current prototype. The following are the reactive behaviors that compose BA:

– a `CyclicBehavior` for waiting and receiving request from QAs.
– a `SequentialBehavior` for implementing query expansion and materialization.
– a `CyclicBehavior` for waiting, receiving, and solving request for the BA ontology.
– a `CyclicBehavior` for waiting, and accepting SINodes to be managed by this BA.
– a `CyclicBehavior` for waiting, receiving and processing messages from SINodes communicating changes in their ontologies.

7 Security issues

The architecture we have presented so far has been devoloped to satisfy the application requirements of the SEWASIE project. It describes how the functionalities of the systems can be modelled and implemented using a multi-agent system. Besides this design architecture, the deployment of the SEWASIE system gives rise to new requirements. Deployment requirements are posed by the specific installation that we have to set up.

The SEWASIE system is conceived to operate in an environemnt where hosts usually provide information services, such as running a DBMS. Playing a delicate role, such machines are usually servers and are subject to security restrictions in order to guarantee the best possible level of service.

While some choices pertain to specific settings, a general trend we can notice is the preference by system administrators towards software products that are compliant with firewalled networks. It is not much about security, but being able to make available services through a web server. This way, the security policy can be focussed on the web server functioning, monitoring the traffic on the port it is listening to.

We thus couple the architecture of the SEWASIE system with a web server, making provision of a tunneling technique. Tunnelling is a popular technique in these days, as it allows to expose services on the standard port of a web server. Client applications can then reach the service by executing an HTTP request. The web server will redirect the request to the particular service addressed. Responses are sent following the backward path.

Each host taking part to the SEWASIE system has a deployment infrastructure like the one depicted in Figure 2:

The Web server acts as gate to the network environment. Messages and objects to and from an agent container belonging to the platform are *HTTP* requests going through the web server. This is made possible because Jade manages remote objects and remote calls using Java RMI [**?**]. When an RMI server is activated, a registry to keep track of all (possibly remote) objects registered is inititated which listen to incoming requests on a given port number. An RMI

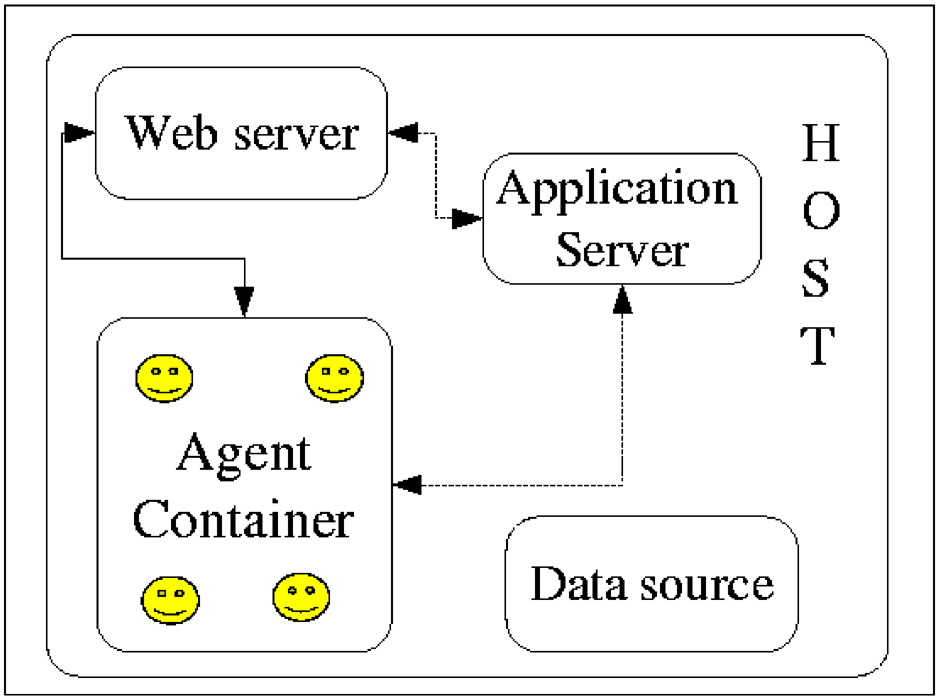

Fig. 2. The deployment architecture of a SEWASIE host.

client can call this service in order to remotely connect and use objects. When an RMI request has to be sent, the client first tries to connect to an *RMI* server listenining on the specified host and port. If this request is blocked by a firewall, the same RMI client tries to invoke the remote method making use of the HTTP protocol, by sending an HTTP POST message. The endpoint of the communication is obtained by using the same host address as for the direct connection on the default web server port 80. The URL is then completed with a call to a *cgi script*, passing the port of the RMI server as parameter. The script forwards the invocation to the *RMI* server on the specified port. All this is part of the Java RMI specification (see [**?**]).

By means of the cgi script, requests are served spawning a new virtual machine for each invocation. Given the scenario the SEWASIE system addresses, this represents a performance limitation. It is thus required to refine the architecture, making available a more sophisticated mecahnism that allows not only the simple redirection (the continuos line in Figure 2) but also to serve requests within one single Java virtual machine. Better performance can be reached introducing an application server, in our case a Java servlet container. Instead of using the cgi script, we can demand to a servlet the management of incoming requests (the dotted line in Figure 2). This brings into play more flexibility as we can program how the servlet behaves. Of course, the drawback lies in the added layer to our software infrastructure, which means a potential weakining of reliability as more things can go wrong. This represent a tradeoff.

Both solutions respond to the need of firewalled networks. In general, a SEWASIE system can be composed by nodes using only a web server and nodes using a web server and an application server. What differs is how they internally manage requests.

8 Conclusions

In this paper, we have provided a general description of the SEWASIE system architecture and of an SINode. We have then detailed how the system has been designed and implemented using agent technology. We have in fact shown the different types of agents and how they are organised. While tackling implementation issues, we have made some observation on the deployment architecture of the SEWASIE system.

Acknowledgements
This work is supported in part by the 5th Framework IST programme of the European Community through project SEWASIE within the Semantic Web Action Line. The SEWASIE consortium comprises in addition to the author' organization (Sonia Bergamaschi is the coordinator of the project), the Universities of Aachen RWTH (M. Jarke), Roma La Sapienza (M. Lenzerini, T. Catarci), Bolzano (E. Franconi), as well as IBM Italia (G. Vetere), Thinking Networks AG (C. Engels) and CNA (A. Tavernari) as user organisation.