Implementing the Duty Trip Support Application

Grzegorz Frąckowiak[*], Sang Keun Rhee[**,***], Marcin Paprzycki[*], Michał Szymczak[*], Maria Ganzha[*,****], and Myon Woong Park[**,***]

[*] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland
E-Mail: marcin.paprzycki@ibspan.waw.pl
[**] Korea Institute of Science and Technology, Seoul, Korea
[***] University of Science and Technology, Korea
****University of Gdańsk, Gdańsk, Poland
E-Mail: greyrhee@kist.re.kr

Abstract: *We are in the process of developing an agent and ontology-based Duty Trip Support application. The goal of this paper is to consider issues arising when implementing such a system. In addition to the description of our current implementation, which is also critically analyzed, other possible approaches are considered as well.*

Keywords*: software agents, agent systems, ontologies, transport objects, agent-non-agent integration.*

1. INTRODUCTION

Our recent work concerns development of an ontology and agent-based system supporting workers in a virtual organization. Thus far, focus of our work was on the conceptualization of the system [1] and its various functionalities [2], as well as design of a novel ontological matchmaking algorithm [3, 4, 5]. Treating an ontology as an ordered graph, our algorithm utilizes multi-pathway approach to establishing semantic relevance of ontologically demarcated resources. For instance, in [3] we present how the proposed algorithm allows establishing persons (and/or conferences) that are of potential interest to a researcher who travels from East Asia to Europe during a so-called *Duty Trip*. The aim of this paper is to discuss in some detail how the system is being implemented. In addition to the presentation of our current approach, we also consider other possibilities. Finally, we try to summarize most important observations made during system implementation.

To this effect we proceed as follows. We start by introducing the *Duty Trip* concept and presenting a sample scenario of how the system works. In the next section we discuss the implementation of core functionalities of the system. We follow by presenting other important functionalities that have to be included in the application. We complete the paper with lessons learned about implementing systems combining agents and ontologies.

2. DUTY TRIP SUPPORT

Concept of the *Duty Trip* (*DT*) is well-established in Korea, where sending a researcher to a conference (or any other business trip) often involves a long-distance (e.g trans-Pacific) and very expensive travel. This being the case it has been established that it makes an economic sense to combine multiple activities with a single airline travel. Therefore, in the case of a faculty member travelling to a conference, it is natural to combine such a trip with visits to near-by universities and/or research institutes. The aim of our work is to provide support for workers involved in a *Duty Trip*. Such support includes not only suggesting persons or conferences, but also travel-related entities, such as hotels or restaurants. The main assumption behind our system is that it will be based on utilization of ontologies and software agents. Let us now briefly describe the proposed system, while more details can be found in [1-11].

Let us start from ontologies and their utilization (more details concerning this topic can be found in [3, 4, 5, 7, 8]). The main assumption behind our system is that every "actor" appearing in it is uniformly treated as a *resource.* This concerns persons, institutions, computers, books, software, as well as "entities of interest during a trip" (e.g. restaurants, or hotels), etc. Furthermore, each *resource* is an instance of an ontology used in the system, and has its own ontological *resource profile*. As a result, a very large class of operations performed in the system can be conceptualized as ontological matchmaking. In particular, this concerns all functionalities that are based on (i) establishing ontological closeness (relevance) of *resources*, which allows, (ii) on the basis of the strength of the relevance, to decide if selected *resources* should be "recommended" to "each-other." For instance, the proposed matchmaking algorithm can be used to judge if a grant announcement is relevant to a researcher and should be forwarded to her, or not ([4, 5]).

Let us now present a sample scenario of how the system works. Before proceeding, we have to make a few assumptions. First, we assume that an organization will utilize (internally) a single ontology and thus there is no need for matching between ontologies, but only for establishing relevance between *resources* within that ontology. Second, within the ontology, two levels of importance of relations are distinguished: (i) resulting from the model (i.e. representing the designer-perceived importance of relations), and (ii) representing individual "preferences" (e.g. personally perceived level of expertise in a given research field). Finally, let us assume that the system is already in operation for some time; e.g. in an Advanced Research Institute of the Korean Academy of Sciences (ARIKAS). This means that workers of ARIKAS have accounts in the system and have been using it to plan, approve, and report their *Duty Trips*. This assumption is necessary to claim that the system contains substantial enough body of knowledge about resources (e.g. attended conferences, visited people, used hotels, etc.). Recall that "knowledge items" are stored as instances of ontologically demarcated *resource profiles*.

A. Duty Trip Scenario

Let us now consider Prof. Lee from ARIKAS, whose research interests are *software engineering*, *agent systems*, and *ontologies.* The *Personal Agent* (*PA*) of Prof. Lee found in the system information about a workshop, which will take place in Paris, France on October 18-20, 2010. *Resource profile* of this workshop was demarcated as concerning *software agents, agent systems,* and *medical informatics.* The *PA* used a matchmaking service and established that this event could be of interest to Prof. Lee and recommended it. Following the ARIKAS procedures (see the sequence diagram presented in [2] as Figure 2, and the discussion that concerns its content), Prof. Lee uses his *PA* to help him prepare a *Duty Trip Request* (*DT Request*). After filling the web-form with initial data concerning his trip, he sends it to the *PA*, which inquires if he would like to consider extending his trip and include additional activities. Since Prof. Lee has some extra time after the conference, he confirms. As a result the system is searched for recommendations (see, [7] for more details). In the first step the geospatial filtering is used to select *resources* stored in the system that are located within a specific distance from the target. Here, let us assume that cities within 300 km from Paris are extracted (300 kilometers is a distance that usually can be travelled to complete a one-day visit). Note that, at this stage of development of the system, only resources already stored in it can be used to provide suggestions. In other words, if someone in the ARIKAS interacted with Prof. Lacroix from Ghent, Belgium, then the *resource profile* of this person will be in the system, and thus Ghent could be selected as a result of a geospatial query. However, if no *resource* located in Compiegne, France is stored in the system, then Compiegne cannot be selected. This limitation of our system may be worthy dealing with in the future (e.g. by adding ability to search the Internet for additional resources). As a result of geospatial processing, the following cities (and people) located within 300 km from Paris could have been selected: Ghent and Prof. Lacroix from the University of Ghent, Lille and Prof. Olejnik from the EPFL, and Reims and Prof. Colombard from the University of Reims. In the second step of the process, professional *resource profiles* of the three (geospatially) selected researchers are matched against the professional *resource profile* of Prof. Lee (for a detailed description of the matching procedure, see [3]). As a result it could be established that research interests of Prof. Lacroix are too distant from those of Prof. Lee. However, interests of Prof. Olejnik and Prof. Colombard are close enough (Prof. Olejnik's interests match very closely with *ontologies*, while Prof. Colombard matches to some extent all three of research interests of Prof. Lee). Therefore, as a result of ontological matchmaking, Prof. Olejnik and Prof. Colombard, and information about their research interests, are forwarded to Prof. Lee (who will also be able to browse their profiles, as well as past *Duty Trip Reports* – from other researchers of ARIKAS mentioning them – stored in the system). Based on provided data, Prof. Lee includes visits to those two researchers (on the 21st and 22nd of October, 2010) into his *DT Request* document. Obviously, this step may involve an earlier visit approaval from the two potential hosts.

When the *DT Request* is completed, information about it is sent by Prof. Lee's *PA* to the *PA* of Prof. Park, Director of his Research Laboratory within ARIKAS. Knowledge on how to deal with the complete *DT Request* is a part of the ontology of the organization [8]. Prof. Park analyses the request, and may utilize functionalities available in his *PA* to fetch additional resources (e.g. the complete list of recent trips undertaken by Prof. Lee). When the final decision is reached, the status of the *DT Request* is changed into approved (or rejected) and this fact can be checked by the *PA* of Prof. Lee, which then informs him about the outcome of his application.

Let us now assume that the *DT Request* was approved and that Prof. Lee has completed his trip. Here, according to the ARIKAS procedures, Prof. Lee has to file a *Duty Trip Report* (*DT Report*). In this part of the process, the system supports users in two ways. First, it helps collecting information necessary to file the *DT Report*. Second, extracts from the *DT Report* data to be stored in the system. Let us assume that during his visit in Paris, Prof. Lee met Prof. Durant, from Dijon, France, who is a specialist in *agent systems*, and *sensor networks*. In this case, as a part of preparing the *DT Report*, Prof. Lee provides information about Prof. Durant, which will be used by the system to create two new *resources* (Prof. Duran and Université de Bourgogne, where Prof. Durant works) and to instantiate their *resource profiles*. It is also possible that Prof. Lee has noticed that research interests of Prof. Olejnik have changed. They are now *semantic web services* and *social networks.* Again, during the process or preparing the *DT Report*, information about modified research interests of Prof. Olejnik will be made available to the system, extracted, and stored (while the log of this modification is going to be kept as well). In a similar way information about conferences, hotels and/or restaurants is dealt with. Here we can add that not only a log of every modification, but also a log of all user interactions is kept in the system. Such logs are to be used to support user profile adaptivity (see, [9, 10] for more details).

3. IMPLEMENTING THE CORE FUNCTIONALITIES

Let us now consider various issues that arise when implementing our system. Let us start from a simple observation that, as can be seen from the above scenario, agents in the system vary in their “abilities.” Consider agents representing Prof. Lee and his Director – Prof. Park (see, also [1]). Both their *PA's* are designed to support them in their work. However, since Prof. Lee and Prof. Park have different responsibilities (researchers vs. administrator), their *PA's* have to be able to perform different actions. For instance, Director's *PA* has to help him to approve *DT Requests*, submitted by researchers from his laboratory. To achieve this goal, the *PA* needs (among others) to be able to: (i) access *DT Requests*, (ii) access data of all researchers in the laboratory (e.g. to check past travel activities, or if funds are available in their projects), (iii) change the status of the *DT Request* to approved or denied, (iv) check/confirm completion of the *DT Report* (after the trip). On the other hand, the *PA* of Prof. Lee does not have to have any of these capabilities, but has to be able to help him in a way described above. Let us now use the first part of the sample scenario: preparing the *DT Request* and receiving the decision, to discuss top level issues concerning implementation of core functionalities of the system.

A. Implementing agents – generalities

As stated above, when we consider agents supporting individual users, their *Personal Agents* will share some functionalities, while differing in others (difference will be associated primarily with the roles and positions of their “owners” in the organization; see, also [1, 11]). Specifics depend on the structure of the organization (e.g. depth and breadth of its hierarchical structure) and are represented in its ontology ([8]). However, it can be assumed that, typically, *PA's* of “workers” involve functions associated with specific projects (including the *Duty Trip* support), while agents supporting managers need functionalities involved in support of managerial functions (including the *Duty Trip* processing). Note also that, in addition to system-centric functionalities, some functions involve access to external systems (e.g. databases via the JDBC, WebServices, web clients, etc.). Therefore, it is necessary to also implement functions that allow agents communicating with external artefacts, as well as allowing agents to be exposed to external systems. One example of such functionality is the gateway between the external systems and the agent platform, which we describe Section 4.A.

Our agent platform of choice is JADE [12]. In JADE agent actions are performed as so-called behaviours. Obviously, in an actual system most of needed behaviours will be complex and/or cyclic (repeated periodically, e.g. the PA checking status of the *DT Request* submitted for approval). Let us now consider the case of a *PA* of the Director of the Research Laboratory. Such an agent has to have behaviours dealing with the *Duty Trip Request* processing (*ManagerDutyTripClientBehaviour*). This also involves the need to access personal data of workers (*ManagerPersonalDataClientBehaviour*). However, observe that workers also have to have behaviours dealing with similar issues: *WorkerDutyTripClientBehaviour* – responsible for help in preparing the *DTR* before and the *Report* after the trip; and *WorkerDataClientBehaviour* – allowing worker access to some of her/his personal data. Therefore, it is easy to observe that the core difference between *PA's* of the Director and of the worker will be: which of the behaviours will be made available to it. In this context observe also an interesting situation when a worker becomes temporarily promoted (e.g. for a 3 year period) to the position of the Director of the Laboratory. In this case her *PA* has to be modified by modifying some worker behaviours and adding appropriate managerial behaviours. First, observe that since Laboratory Directors also undertake *Duty Trips*, they still need the basic *WorkerDutyTripClientBehaviour*, However their travel requests are approved by someone "higher" in the hierarchy of the organization. This information is stored in the ontology of the organization and to if provided to the *PA* during its modification (in the form of a modified *module* containing the *WorkerDutyTripClientBehaviour*, see below). Second, the *ManagerDutyTripClientBehaviour* needs to be added to the *PA* of the promoted worker. However, when the directorial duty is over, the reverse process has to take place.

These considerations lead us to the main concept used in our agent implementation – the idea of a *module*. *Module* can be seen as a collection of behaviours, that are to be performed in order to support some high-level functionalities (e.g. *Duty Trip* processing). Note that *modules* consists of not only the list of the behaviours that can run by an agent, but also:

- the description of the module (name, version, meta-information),
- description of data required by those behaviours (e.g. specification of *resource profiles* that will need to be accessed),
- pre-launched behaviours (initialized interfaces to shared data model access/monitor services),
- the order of start of behaviours

As described in [10,11], any agent created in the system initially has only one module "built-in." It only has the behaviour which is responsible for loading other modules necessary for the agent to fulfil its role. In the next stage of their creation, agents are adapted to fulfil their specific roles. This is achieved by loading them with appropriate modules. By default modules are loaded by a specialized *Admin Agent* which is one of the auxiliary agents that are instantiated when the system is starting. The module sent to an agent is a concrete instance of *Module* class, which is created specially for a given agent. For instance, in the above presented scenario, *Personal Agent* of Prof. Lee will be loaded with modules containing the *WorkerDutyTripClientBehaviour* and the *WorkerDataClientBehaviour*, while the *PA* of Prof. Park will receive modules containing the *ManagerDutyTripClientBehaviour* and the *ManagerPersonalDataClientBehaviour*. Furthermore, if Prof. Lee was to become the Director of the Laboratory (and replace Prof. Park), the following actions would be performed (obviously, we limit our attention only to the sample behaviours, while the the scope of the change is much larger. In the case of Prof. Park, his *PA* would "loose" all modules containing managerial behaviours, while modules concerning worker behaviours would have to be modified. The reverse process would be applied to the *PA* of Prof. Lee. Unfortunately, at this stage of our understanding of the JADE agent platform, the most natural (and easiest from the point of view of the implementation) way of achieving this goal would involve taking down the whole system, performing maintenance on *PA's* of Prof. Park and Prof. Lee, and restarting the system. This issue definitely requires more attention in the future.

B. Storing and managing ontologically demarcated data

As specified above, in addition to software agents, our system is based on utilization of ontologically demarcated data and semantic reasoning. In our approach, the entire knowledge model is designed in OWL-DL [13], and the details of its structure have been presented in [8]. As far as data persistence is concerned, on the lowest level, all data is stored in the PostgreSQL relational database. This data is accessed and manipulated via the Jena2 Database Interface[14].

Observe that, among others, due to the extensive utilization of software agents (JADE is a Java-based agent system), our system is fundamentally based on Java. Therefore, we notice that, as stated in [15], there are several benefits of mapping an OWL ontology into Java; e.g. (i) keeping consistency between the design-stage specifications and applications (including agents), (ii) ease of debugging of the application or ontology via any Java IDE, and (iii) possibility of use of *javadoc* as an on-line documentation of the ontology. Therefore, a set of Java API has been generated from our ontology schema utilizing Jastor [16], which is an open source Java code generator developed on the basis of [15]. There are, however, some drawbacks to this approach. The main one is the fact that the structure of the generated objects is much more complex compared to those in generic object-oriented development. For example, a *Person* object is created to describe a person, and it is connected to multiple objects representing his/her various profiles – *PersonProfile*, *ContactProfile*, *ExperienceProfile*, and *PreferenceProfile*. Continuing, we see that the *ContactProfile* includes another object – the *Address*, which in turn includes the *City* object, and the process repeats. Although such structure is desirable in the sense of ontology design, such complexity of objects is likely to become problematic to application developers unless they participated in the ontology design as well. As a matter of fact we have run into this problem directly due to the fact that the ontology was designed in Poland, while the application using it was being implemented in Korea. One more consequence of pursuing this line of system design and implementation, was the need to develop our own *gateway* (infrastructure allowing the web-based client communicating in ontologically rich way with the agent-based core of the system), instead of using the one provided by the JADE. This was the only way we could deal with the complexity of objects passed into and from the agent system (see Section 4 for details).

C. Implementing ontological matchmaking

Let us now consider the core functionality of our current system – management of recommendation requests, which is based on ontological matchmaking. The main building block for this function is the *Relevance Calculation Engine* (*RCE*). The two main operations of this module are: (i) creating a *Relevance Graph* – a directed labeled graph structure generated from the ontology model for the purpose of calculation of semantic distance (relevance) between resources, and (ii) matching resources for the purpose of finding those that are relevant to the given source object (among a list of target objects). The source object, and a list of target objects are specified, along with other variables (e.g. the threshold of relevance), in the form of the matching criteria (discussed in [3]). The ontology model handling is performed by the Jena API [17], while the graph structure is managed utilizing the Structure Package [18] libraries.

The definition of the *Relevance Graph* and the method for generating it from an ontology model is provided in [3], so we will focus here on the implementation-related aspects of the process. During the development of the system, it was discovered that the graph generation procedure takes a rather long time, making it impractical to be performed each time there is a request for the relevance calculation. Recall that relevance calculations are the core (most often performed) operations in the system. We have also noted that the graph stays unchanged unless there is a change in the data. Therefore, it was decided to create the *Relevance Graph* as a background process and save it locally so that the system can quickly load the structure from a file instead of repeating the time-consuming generation process. However, each time there is a change in our repository, i.e. when data is added, deleted or updated, an auxiliary agent responsible for communicating with the data source, uses the Pellet [19] as a reasoner to generate a Jena ontology model from the semantic data storage. The *RCE* takes this ontology model as an input and regenerates the *Relevance Graph*, which is then locally stored as a binary file, replacing the old one.

Before considering details of the object matching process, let us briefly describe an additional module used by the *RCE* – the *GIS sub-system* [5, 6, 7], which is used for all geospatial data management. For the purpose of this module, geographic coordinates of cities were collected via the GeoMaker [20], and the resulting data was stored in the PostgreSQL database. However, this database is separate from our semantic data storage and is used only for geospatial data processing. As indicated in the sample scenario, the *GIS sub-system* on demand calculates distances between cities, using Great Circle Distance Formula [21]. Obviously, this method of distance calculation is not an optimal one, from the point of view of the actual travel (e.g. car or train distance is not a straight-line distance). However, utilizing another service like the Google car travel distance API is not feasible for filtering on a large scale. On the other hand, note that it would be possible to combine the two methods. Use the Great Circle Distance Formula for geospatial pre-filtering, while applying the Google API for distance checking for a limited number of pre-filtered geo-objects.

Note that the *GIS sub-system* is an "optional" module, and we utilized it because the main purpose of our system is to support *Duty Trips*, where the geospatial information is essential. For other implementation cases, it can be removed (as not needed), or replaced with other module(s) dealing with other types of domain specific information (e.g. dealing with inputs from sensors).

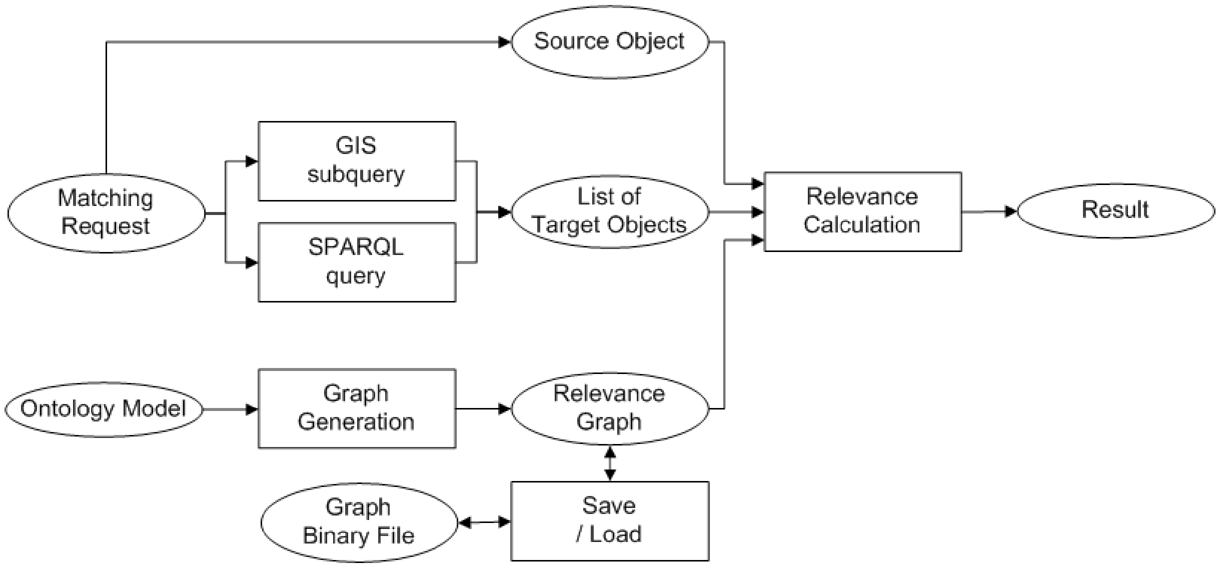

Fig. 1. Generic Matching Process

The generic matching request handling process is depicted in Figure 1. The matchmaking process can be conceptually divided into (i) synchronous matching request processing – used for handling requests requiring a real-time matching and/or requiring a single result based on the current state of the data storage, and (ii) asynchronous matching request processing – applied to low priority matching, which needs to be repeated in a certain intervals and results of which can be used multiple times (see [5] for more details). Following this, two separate matching services have been implemented – the *SyncMatchingService* and the *AsyncMatchingService*, respectively. The former is utilized in our *Duty Trip* support system, while the latter supports grant announcement services [4, 5].

Now, let us follow the example of Prof. Lee, trying to obtain recommendations of people to visit during his trip (seeking conferences, hotels, or restaurants would proceed following exactly the same procedure). Here, we focus on processing of a specific *request*, keeping in mind that such request appears in the context of user-*PA* interactions (see the sample scenario in Section 2). First, Prof. Lee initiates *request* processing by filling appropriate forms and clicking a submit button on the web client. This request is forwarded to his *PA* via the *gateway* (described in Section 4.A). The *PA* analizes the received information and forwards it to the auxiliary *Matching Agent* (*MA*), where the actual *request* is composed. In the process of composing the *request*, *t*he *MA augments* it with all variables of matching criteria (described in [3]), and calls the *SyncMatchingService*. In that service, the GIS subquery is processed in the *GIS sub-system*, and a list of cities that are within a certain range (e.g. 300 km) from the main destination (e.g. Paris) is returned. Next, the SPARQL query is generated and processed to obtain a list of (potential) target objects, filtered to obtain the list of objects whose location (specified in their profiles) is one of the cities resulting from the geospatial filtering. In our example, the *PA* is requesting person suggestion, hence a list of *ContactPerson* objects is returned. Similarly, a list of *Conference* objects, or a list of *Organization* objects would be returned if a conference or an organization recommendation was requested. The resulting list becomes the list of *target objects*, and Prof. Lee (or any other person that a specific *PA* is representing) becomes a *source object*. The *list* and the *object*, as well as other needed parameters (e.g. the *Threshold* value) and the *Relevance Graph* (loaded from the binary file, described above) are the inputs to the *RCE*. The *RCE* calculates the relevance values from the source object to each of the target objects (e.g. the relevance from Prof. Lee to each person in the target list), and generates a result in the form of *Map <Key*, *Value>* where the *Key* is the URI of an object and the *Value* is its relevance to the source object. Only objects whose relevance value is above a specific *threshold* value are returned to the *PA;* ordered by their relevance.

4. IMPLEMENTING THE AUXILIARY FUNCTIONALITIES

Let us now discuss auxiliary functionalities that were implemented for the system to work.

A. Implementing agent system gateway

Before we proceed, we need to deal with an important assumption that was made while implementing the current prototype. Overall, what we are dealing with in this section, is the meta-level question – how an agent system is to be designed. The general assumption behind the concept of agent systems is: "agents everywhere;" meaning, that the complete system is spread across all needed computers and Internet-enabled devices. This assumption has been discussed in [22, 23], and shown to be highly unrealistic. In the case of our system, this translates into the question – where should the *Personal Agent* reside. If the *PA* was to reside on the computer / device of its user, then the system could be extended to enclose also that machine (the "agents everywhere" type solution). In this case, all communication within the system could be ACL-based. Unfortunately, this approach leads to a number of open problems, for instance: (i) what if the user utilizes multiple devices; where will the *PA* be located (will it move between devices?) and how will the user be able to communicate with it? (ii) if the user shuts down her device, then she "takes down" her *PA;* is this an acceptable solution (in [1] we have assumed that the *PA* is going to persistently work to support its user)? (iii) if the user goes off-line and the mobile *PA* migrates to the main server of the organization before this happens, then the user does not have its *PA* while off-line? (iv) if we have two copies of the *PA*, one on the user device and one of the main server, how to deal with their synchronization / integration? This being the case, to avoid addressing all of these questions, in the current prototype, we have decided that all *PA's* will reside within the system, while the user will interact with her *PA* via the web interface and a *gateway.*

More generally, our system requires that agents residing within the platform communicate with software artifacts external to the agent platform. JADE provides two possibilities to achieve such communication: (i) the low level *jade.wrapper.gateway.JadeGateway* class, and (ii) the high level *JADE Web Service Integration Gateway* (*WSIG*). Here, we run into the problem caused by the fact that in the prototype system we have decided that we will utilize complex Java objects (see above, Section 3-B). Therefore, we could not easily take advantage of the *WSIG*. Instead, we have developed our own solution based on the *jade.wrapper.gateway.JadeGateway.* Specifically, we have created a multi-threaded component processing external users requests. In our solution, the system provides an API of a *gateway queue*, which allows for *synchronous* access to its resources. In the implemented prototype, we used the queue from the *java.concurrent* package, but any other queuing mechanism could have been used. Obviously, this queue has to be also accessible from within the system, where a set of *Gateway Agents* (*GA*) monitor its status and process information. In this way the queue becomes a de facto interface between the agent platform and the outside world. Unfortunately, we have to admit that this solution is not FIPA [24] compliant.

Let us now consider interactions that involve information crossing the *gateway.* A thread representing an outside entity (e.g. a user web client) utilizes the *Gateway API* to put a *request* into the *gateway queue*, and awaits notification on a specific object. More precisely, objects of the type *Event*, which are wrapping the *Request* are put into the queue. The thread is waiting on that *Event* for the agent to process the *Request.* Upon completion of *Request* processing, the agent packs the *Result* into the *Event* and calls the *Event* (thus notifying the thread that the *Result* is ready to be picked up).

The *gateway queue* is monitored by the *GA's*. These agents utilize a special *register*, which is user-configurable and which allows to map user *requests* to ACL messages. Register configuration can be achieved in many different ways. In the current implementation translation between *request* and *message* is hard-coded, however, as the system develops, we plan to design and implement special classes that will make the system more flexible and allow configuration to be completed on the basis of XML configuration files.

On the technical side, the register contains a map of objects (the *ExternalService)*. Each of them, a single service, has its own unique name and describes: (i) application-specific ontologies for agent communications, (ii) codecs needed by the agents, and (iii) actions, *ServiceAction*. Here, the *ServiceAction* contains information that will allow translation of a *request* into an ACL message. For example, let us assume that a *PA* utilizes the *DtaExternalOntology*, to understand what to do when dealing with *requests*. This is a special ontology designed to facilitate interactions with external entities. It has description of actions that can be undertaken by agents. For each action, there is an object which is an extension of the class *AgentAction.* Such an object can contain parameters used by the agent to perform an action. Let us assume that, within the ontology, the following two actions are defined – *APPROVE_DTA* and *GET_DTA_LIST.* Implementing a service which allows the use of these two actions, we create objects of class *ExternalService*, which contains the following fields:

- *name*: *DtaService*
- *ontology*: *DtaExternalOntology*
- *codec*: *SLCodec*
- *actionMap* (*external_service_map*: *ServiceAction*)

The *ActionMap*, on the other hand, contains descriptions of actions, for instance:

ServiceAction:

- *name*: ”*approveDta*”
- *ontology*: *DtaExternalOntology*
- *codec*: *SLCodec*
- *parametersMapping* (map of parameters of the request to parameters of the action)

Then, the *request* in the *gateway queue* is generalised as follows:

Request:

- *serviceName*: *DtaService*
- *actionName*: *approveDta*
- *parameters*: (*dutyTrip*: *object dutyTrip*)
- *agentName*: *name / ID of a specific Personal Agent*

This allows the *GA*, which picks the *request* from the *gateway queue*, to create an ACL message. First, it uses the *Request* to find the right service, based on the *serviceName* (here, the *DtaService*). Next, in case it does not already have the right ontology (the *DtaExternalOntology*), it registers it and also the codecs (the *SLCodec*). Then it finds the requested action(s) (the *approveDta* action). Based on that action, it generates the ACL message, with the codec, the ontology, and the action object created using the reflection. Message is sent to the agent specified in the *Request* (the *agentName* defines it). Obviously, the target agent can: (i) perform an action and send the result back, (ii) reject the request, etc. Depending of the result of the action, the *RequestResult* is generated and sent back to the *GA.* The *GA,* in turn, informs the thread that generated the *Request*, that it has been completed (by calling an appropriate *Event*; see above).

In the current version of our system, we use a 1:1 mapping between ontologies and services. In other words, services contain all actions of the ontology. Even though currently we use a special function that creates the service object using a reflection (where the input is the ontology, actions of which we would like to make available), we also acknowledge that this is not the only possible solution. It would be also possible to generate services that would contain configurations “internally.” In this case, names of actions and parameters would be identical to those in the ontology.

Please be reminded that the ontology mentioned in this section is an application-specific ontology describing the elements to be used as the content of agent messages, and do not confuse it with the semantic knowledge space used in our system.

5. CRITICAL ANALYSIS

Across the paper we have pointed out to a number of controversial points in our implementation, as well as places where it could have been decided to implement things differently. Here we would like to look into some other issues that materialized during the system implementation. The first, and the main one is our overall experience. Based on what we came across in our work, we cannot agree with N. Jennings, who (in [25]) claimed that software agents and agent systems are the future of design and implementation of complex systems. While this claim may be valid sometime in the future, today's implementation and maintenance of an agent system (maintenance understood as following the spiral model of system design, where after the initial design and testing phase, modifications ensue) turn out to be more difficult than in the case of traditional systems. For instance, one of major sources of problems turns out to be dealing with utilization of agents and ontologies in the same system. Any change in the agent side, requires immediate changes in the ontologies and such changes usually are non-trivial. This also contradicts to some extent major claims put forward by J. Handler in [26]. All of our experiences show, that currently available tools for development of agent systems have not reached the level of maturity required for the visions found in [25, 26] to materialize.

Other, lesser, problems that we came across were as follows. (i) It is extremely difficult to develop an agent system that will be at least to some extent resilient to failure. Even though existing agent tools are characterized by quite good scalability (see, for instance [27]) they turn out to be rather "fragile" and there is no simple way to "harden" them. (ii) As mentioned above, the only realistic way to introduce changes into the (JADE) system is by stopping it completely, introducing changes and restarting. While we have started working on methods to address this problem, we could not find a solution that would be simple enough to attempt at implementing it. Again, this is in conflict with conjectures presented in [25] and points to overall weakness of JADE. (iii) For all practical purposes it is impossible to assure that agent functionality is protected. This indicates, that security of agent systems remains an open research question (for an overview of agent system security, see [28]). (iv) We have found, again (see, also [29]) that strict conformance to FIPA standards is unreasonable. More precisely, remaining in strict conformance to the FIPA standard is possible primarily in systems which are built according to the "agents everywhere" metaphor. Anytime an agent system has to communicate with non-agent world, FIPA conformance becomes a problem. Furthermore, FIPA standard is also a problem as soon as system performance is considered (see, [27, 29]).

6. CONCLUDING REMARKS

The goal of this paper was to report on issues materializing when implementing a system based on joint utilization of software agents and ontologies. In addition to the description of most important facets of our prototype implementation, we have acknowledged other possible approaches to the implementation of its parts. Furthermore, we have summarized lessons learned during our work. Here, our conclusions are somewhat pessimistic. It is now 10 years after publication of the highly critical, but very insightful work of H. Nwana and D. Ndumu ([30]). This work contained pragmatic guidelines for progressing in the field of agent systems research. Unfortunately, it does not seem that the agent community has followed these guidelines, as we have found in practice that software agents combined with ontologies as an approach to complex system design and implementation, and even more so existing tools developed to help in this process, are not yet ready for prime time.

ACKNOWLEDGMENTS

Work of Grzegorz Frąckowiak, Sang Keun Rhee, Michał Szymczak, and Myon Woong Park was partially sponsored by the KIST-SRI PAS collaborative grant ”Agent Technology for Adaptive Information Provisioning”.

REFERENCES

[1] M. Szymczak, G. Frackowiak, M. Ganzha, M. Gawinecki, M. Paprzycki, M.-W. Park, "Resource Management in an Agent-based Virtual Organization – Introducing a Task Into the System, in Proceedings of the MaSeB Workshop, IEEE CS Press, Los Alamitos, CA, pp. 458 – 462, 2007.

[2] M. Ganzha, M. Paprzycki, M.Gawinecki, M. Szymczak, G. Frackowiak, C. Bădică, E. Popescu, M.-W. Park, "Adaptive Information Provisioning in an Agent-Based Virtual Organization - Preliminary Considerations," in V. Negru et. al. (eds.), Proceedings of the SYNASC Conference, IEEE CS Press, Los Alamitos, CA, pp. 235 – 241, 2007.

[3] S. K. Rhee, J. Lee, M.-W. Park, M. Szymczak, G. Frackowiak, M. Ganzha, M. Paprzycki, “Measuring Semantic Closeness of Ontologically Demarcated Resources,” in Fundamenta Informaticae, 96, pp. 395 – 418 2009.
[4] M. Szymczak, G. Frackowiak, M. Ganzha, M. Paprzycki, S. K. Rhee, J. Lee, Y. T. Sohn, Y.-S. Han, M.-W. Park, “Ontological Matchmaking in a Duty Trip Support Application in a Virtual Organization,” in Proceedings of the 2008 Multiconference on Computer Science and Information Technology, IEEE CS Press, Los Alamitos, CA, pp. 243 – 250, 2008.
[5] M. Szymczak, G. Frackowiak, M. Ganzha, M. Paprzycki, S. K. Rhee, M.-W. Park, Y.-S. Han, Y. T. Sohn, J. Lee, J. K. Kim, “Infrastructure for Ontological Resource Matching in a Virtual Organization,” in C. Badica et. al. (eds.), Intelligent Distributed Computing, Systems and Applications, Springer, Berlin, pp. 11 – 22, 2008.
[6] G. Frackowiak, M. Ganzha, M. Gawinecki, M. Paprzycki, M. Szymczak, M.-W. Park, Y.-S. Han, “Considering Resource Management in Agent-Based Virtual Organization,” in N. Nguyen, L. C. Jain (Eds.), Intelligent Agents in the Evolution of Web and Applications, Springer, Berlin, pp. 161 – 190, 2009
[7] G. Frackowiak, M. Ganzha, M. Gawinecki, M. Paprzycki, M. Szymczak, M.-W. Park, Y.-S. Han, “On Resource Profiling and Matching in an Agent-Based Virtual Organizatnion.” in: L. Rutkowski et. al. (eds.), Artificial Intelligence and Soft Computing – ICAISC 2008, LNAI, Springer, Berlin, pp. 1210 – 1221, 2008.
[8] M. Szymczak, G. Frackowiak, M. Gawinecki, M. Ganzha, M. Paprzycki, M.-W. Park, Y.-S. Han, Y. T. Sohn, “Adaptive Information Provisioning in an Agent-Based Virtual Organization – Ontologies in the System,” in N. T. Nguyen (ed.), Proceedings of the AMSTA-KES Conference, LNAI 4953, Springer, Heidelberg, Germany, pp. 271 – 280, 2008.
[9] C. Bădică, E. Popescu, G. Frackowiak M. Ganzha, M. Paprzycki, M. Szymczak, M.-W. Park, “On Human Resource Adaptability in an Agent-Based Virtual Organization,” in: N. T. Nguyen and R. Katarzyniak (eds.), New Challenges in Applied Intelligence Technologies, Springer, Berlin, pp. 111 – 120, 2008.
[10] M. Ganzha, M. Gawinecki, M. Szymczak, G. Frackowiak, M. Paprzycki, M.-W. Park, Y.-S. Han, Y. T. Sohn, “Generic Framework for Agent Adaptability and Utilization in a Virtual Organization - Preliminary Considerations,” in: J. Cordeiro (et. al.), Proceedings of the 2008 WEBIST Conference, INSTICC Press, Setubal, Portugal, pp. IS-17 – IS-25, 2008.
[11] G. Frackowiak, M. Ganzha, M. Gawinecki, M. Paprzycki, M. Szymczak, C. Badica, Y.-S. Han, M.-W. Park, “Adaptability in an Agent Based Virtual Organization,” in: International Journal of Agent-Oriented Software Engineering, Vol. 3, No. 2/3, pp. 188 – 211, 2009
[12] JADE, http://jade.tilab.com/
[13] OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features/
[14] Jena2 Database Interface, http://jena.sourceforge.net/DB/
[15] A. Kalyanpur, D. J. Pastor, S. Battle, J. Padget, “Automatic Mapping of OWL Ontologies into Java,” in Proceedings of the 6th Int. Conference on Software Engineering and Knowledge Engineering, 2004.
[16] Jastor, http://jastor.sourceforge.net/
[17] Jena – A Semantic Framework for Java, http://jena.sourceforge.net/
[18] Java Structure, http://www.cs.williams.edu/~bailey/JavaStructures/Software.html
[19] Pellet: The Open Source OWL 2 Reasoner, http://clarkparsia.com/pellet/
[20] Geomaker, http://pcwin.com/Software_Development/GeoMaker/index.htm
[21] http://www.meridianworlddata.com/Distance-calculation.asp
[22] M. Gordon, M. Paprzycki, V. Galant, “Agent-Client Interaction in a Web-based E-commerce System” in D. Grigoras (ed.), Proceedings of the International Symposium on Parallel and Distributed Computing , University of Iaşi Press, Iaşi, Romania, pp. 1 – 10, 2002.
[23] M. Gawinecki, M. Gordon, P. Kaczmarek, M. Paprzycki, The Problem of Agent-Client Communication on the Internet,” in Scalable Computing: Practice and Experience , 6(1), pp. 111 – 123, 2005.
[24] FIPA, http://www.fipa.org
[25] N. R. Jennings, “An agent-based approach for building complex software systems,” in Communications of the ACM, 44 (4), pp. 35 – 41, 2001.
[26] J. Hendler, “Agents and the Semantic Web,” in IEEE Intelligent Systems, 16(2), pp. 30-37, 2001.
[27] K. Chmiel, M. Gawinecki, P. Kaczmarek, M. Szymczak, M. Paprzycki, “Efficiency of JADE Agent Platform,” in Scientific Programming, 13(2), pp. 159 – 172, 2005.
[28] Ł. Nitschke, M. Paprzycki, M. Ren, “Mobile Agent Security,” in J. Thomas, M. Essaidi (eds.), Information Assurance and Computer Security, IOS Press, Amsterdam, 102 – 123, 2006.
[29] K. Wasielewska, M. Gawinecki, M. Paprzycki, M. Ganzha, P. Kobzdej, “Optimizing Blackboard Implementation of Agent-Conducted Auctions,” in IADIS International Journal on WWW/Internet , 6(1), pp. 50 – 60, 2008.
[30] H. S. Nwana & D. T. Ndumu, “A Perspective on Software Agents Research,” in The Knowledge Engineering Review, 14(2), pp. 1 – 18, 1999