Using HDS for Realizing Multi-Agent Applications

Federico Bergenti, Enrico Franchi, Agostino Poggi

Abstract— **Although some of the most important works on multi-agent systems focused on interoperating multi-agent systems with legacy applications, the main results consisted in the definition of two agent communication languages, i.e., KQML and FIPA ACL, and a set of specifications, i.e., the FIPA specifications, for the realization of interoperable multi-agent systems. Nowadays, web services are the primary mean to provide interoperability with legacy applications and the large part of multi-agent applications have been realized without any strong requirement for the interoperability with other multi-agent applications. This paper presents the HDS software framework, which provides a software infrastructure to realize multi-agent applications that either take advantage of the specifications for agent-to-agent interoperability or are implemented for optimizing their performance, reducing their development cost and/or simplifying their interaction with some specific legacy applications. Typed messages and message filters are the elements that mainly characterize such a software framework. Besides describing the main features of such a software framework, this paper introduces two different application scenarios designed to exploit HDS features: i) a prototype framework for distributed constraint satisfaction algorithms and ii) a distributed social network system.**

Index Terms—**software framework, multi-agent systems, typed messages, composition filters, Java.**

I. INTRODUCTION

SOME of the most important works on multi-agent systems considered them the solution to provide and maintain the interoperability among legacy applications [1][2][3]. This expectation motivated researchers to work on the problem of proving interoperability both between agents and legacy applications and among agents that are realized by different people and with different software tools. The main results of such works were not related to the interoperability between agents and legacy applications, but consisted in the definition of two agent communication languages, i.e., KQML and FIPA ACL, [4][5][6] and a set of specifications, i.e., the FIPA specifications [7], for the realization of interoperable multi-agent systems. Nowadays, the solution for providing the interoperability among legacy applications has been identified in the Web services technologies and the large part of multi-agent applications have been realized without any strong requirement for the interoperability with other multi-agent applications.

In this paper, we present a software framework, called HDS whose goal is to simplify the realization of multi-agent system by taking advantage of typed messages and message filters and avoiding to be constrained by the use of a specific ACL and by the rules of any specification for the realization of multi-agent systems.

We are not concerned with non-agent based software frameworks; among the agent based software frameworks (Jade, AgentFactory [8]) the dominant approach is using FIPA ACL, whose focus is on interoperability. However, in FIPA based frameworks the communication time is almost always dominated by the parsing and construction times of FIPA messages [9][10]. On the other hand, HDS approach permits to shift the focus on performance, without hindering interoperability by design.

Section II gives a short introduction to HDS framework architecture; Section III presents the three models that concur to the definition of the architecture of a HDS application, while Section IV presents some details on the implementation of HDS. Section V and VI discuss two experimentations where HDS has been used to study distributed constraint solving algorithms and to design distributed social network systems. Eventually, section VII concludes the paper sketching some future research directions.

II. SOFTWARE FRAMEWORK OVERVIEW

HDS (Heterogeneous Distributed System) is a software framework that has the goal of simplifying the realization of distributed applications by merging the client-server and the peer-to-peer paradigms and by implementing all the interactions among all the processes of a system through the exchange of messages.

This software framework allows the realization of systems based on two types of processes: actors and servers. Actors have their own thread of execution and perform tasks by interacting, if necessary, with other processes through synchronous and asynchronous messages. Servers perform tasks on request of other processes by composing, if necessary, the services offered by other processes through

synchronous messages. Moreover, while both servers and actors may directly take advantage of the services provided by other kinds of application, only the servers can provide services to external applications by simply providing one or more public interfaces.

Actors and servers can be distributed on a (heterogeneous) network of computational nodes (thereafter called runtime nodes) for the realization of different kinds of application. In particular, actors and servers are grouped into some runtime nodes that realize a platform. An application can be obtained by combining some preexistent applications by realizing a federation.

III. APPLICATION ARCHITECTURE MODEL

The software architecture of a HDS application can be described through the three different models:

- the concurrency model, which describes how the processes of a runtime node can interact and share resources.
- the runtime model, which describes the services available for managing the processes of an application.
- the distribution model, which describes how the processes of different runtime nodes can communicate.

A. The concurrency model

The concurrency model is based on seven main elements: process, description, description selector, mailer, message, content and message filter.

A process is a computational unit able to perform one or more tasks taking, if necessary, advantage of the tasks provided by other processes. To facilitate the cooperation among processes, a process can advertize itself making available to the other processes its description. The process identifier and the process type represent the default information contained in a description; however, a process may introduce some additional information in its description.

A process can be either an actor or a server. An actor is an active process that can have an active behavior and so can start the execution of some tasks without the request of other processes. A server is a passive process that is only able to perform tasks in response of the request of other processes.

A process can interact with the other processes through the exchange of messages based on one of the following three types of communication:

- synchronous communication, the process sends a message to another process and waits for its answer;
- asynchronous communication, the process sends a message to another process, performs some actions and then waits for its answer;
- one-way communication, the process sends a message to another process, but it does not wait for an answer.

In particular, while an actor can start all the three previous types of communication with all the other processes, a server can only respond to the requests of the other processes it serves them, composing the services provided by other processes through synchronous communications. Moreover, a server can respond to a request through more than one answer

(e.g., when it acts as a broker in a publisher subscriber system) and can forward a request to another server for its execution.

A process has also the ability of discovering the other processes of the application. In fact, it can both get the identifiers of the other mailers of the systems and check if an identifier is bound to another mailer of the system taking advantage of the registry service provided by HDS middleware. Moreover, a process can take advantage of some special objects, called description selectors, for requiring the listing of specific subsets of mailer identifiers. In fact, a description selector allows the definition of some constraints on the information maintained by the process descriptions (e.g., the process must be of a specific type, the process identifier must have a specific prefix and the process must be located in a specific runtime node) and the registry service is able to apply their constraints on the information of the registered descriptions for building the required subsets of identifiers.

A process does not exchange directly messages with the other processes, but delegates this duty to a mailer. In fact, a mailer provides a complete management of the messages of a process: it receives messages from the mailers of the other processes, maintains them up to the process requests theirs processing and, finally, sends messages to the mailers of the other processes.

In a way similar to a process, a mailer can be either an actor mailer or a server mailer. Of course, it depends on the fact that, as described above, an actor and a server can assume a different set of roles in message exchanging.

A message contains the typical information used for exchanging data on the net, i.e., some fields representing the header information, and a special object, called content, that contains the data to be exchanged. In particular, the content object is used for defining the semantics of messages (e.g., if the content is an instance of the Ping class, then the message represents a ping request and if the content is an instance of the Result class, then the message contains the result of a previous request).

Normally, a mailer can communicate with all the other mailers and the sending of messages does not involve any operation that is not related to deliver messages to the destination; however, the presence of message filters can modify the normal delivery of messages.

A message filter is a composition filter [11] whose primary scope is to define the constraints on the reception/sending of messages; however, it can also be used for manipulating messages (e.g., their encryption and decryption) and for the implementation of replication and logging services.

Each mailer has two lists of message filters: the ones in the first list (input message filters) are applied to the input messages and the others (output message filters) are applied to the output messages (Fig 1 shows the flow of the messages from the input message filters to the output message filters). When a new message arrives or is be sent, the message filters of the appropriate list are applied in sequence until a message filter fails; therefore, such a message is stored in the input queue or is sent only if all the message filters have success.
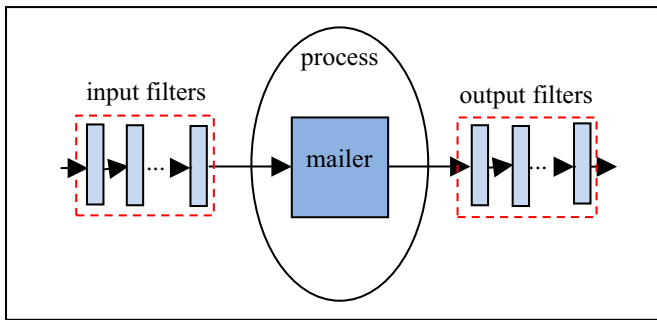
Figure 1. Flow of the messages from the input to the output message filters.

Message filters are not only used for customizing the reception and sending of messages, but are also used by the processes for asking their mailer for the input messages they need for completing their current task. In fact, as described above, a message filter allows to define the constraints that are necessary to identify a specific message and a mailer is able to use it for selecting the first message in the input queue that satisfies its constraints (e.g., the reply to a message sent by the process, a message sent by a specific process and a message with a specific kind of content).

B. The runtime model

The runtime model defines the basic services provided by the middleware to the processes of an application. This model is based on four main elements: registry, processer, filterer and porter.

A registry is a runtime service that allows the discovery of the processes of the application. In fact, a registry provides the binding and unbinding of the processes with their identifiers, the listing of the identifiers of the processes and the retrieval of a special object, called reference, on the basis of the process identifier.

A reference is a proxy of the process that makes transparent the communication respect to the location of the process. Therefore, when a process wants to send a message to another process, it must obtain the reference to the other process and then use it for sending the message.

A processer is a runtime service that has the duty of creating new processes in the local runtime node. Of course, an important side effect of the creation of a process is the creation of the related mailer. The creation is performed on the basis of the qualified name of the class implementing the process, a list initialization parameters.

The processes cannot directly modify the lists of message filters, but they can take advantage of a filterer to do it. A filterer is a runtime service that allows the creation and modification of the lists of message filters associated with the processes of the local runtime node. Therefore, a process can use such a service for managing the lists of its message filters, but also for modifying the lists of message filters associated with the other processes of the local runtime node.

Finally, a porter is a runtime service that has the duty of creating some special objects, called ports, that allows an external application to use the services implemented by a

server of the local runtime node. In particular, a port is a wrapper that encapsulates a server for limiting the access to the functionalities of the process by masquerading the use of some its services and by adding some constraints on the use of some other its services.

C. The distribution model

The distribution model has the goal of defining the software infrastructure that allows the communication of a runtime node with the other nodes of an application possibly through different types of communication supports, guaranteeing a transparent communication among their processes. This model is based on three kinds of element: distributor, connector and connection.
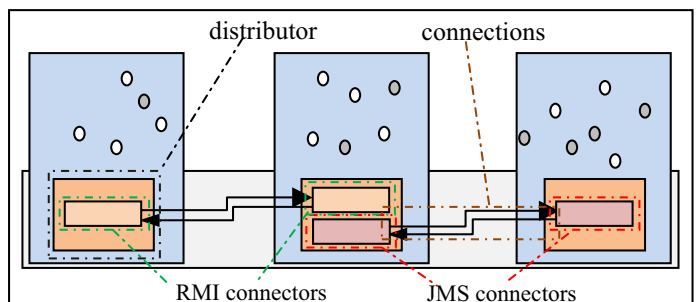

Figure 2. An HDS application based on three runtime nodes connected through RMI and JMS technologies.

A distributor has the duty of managing the connections with the other runtime nodes of the application. This distributor manages connections that can be realized with different kinds of communication technology through the use of different connectors (see Figure 2). Moreover, a pair of runtime nodes can be connected through different connections.

A connector is a connections handler that manages the connections of a runtime node with a specific communication technology allowing the exchange of messages between the processes of the accessible runtime nodes that support such a communication technology.

A connection is a mono-directional communication channel that provides the communication between the processes of two runtime nodes through the use of remote references. In particular, a connection provides a remote lookup service offering the listing of the remote processes and the access to their remote references.

IV. SOFTWARE FRAMEWORK IMPLEMENTATION

The HDS software framework has been realized taking advantage of the Java programming language. The application architecture model has been defined through the use of Java interfaces and its implementation has been divided in two modules.

The first module contains the software components that define the software infrastructure and that are not directly used by the developer, that is, all the software components necessary for managing the lifecycle of processes, the local and remote delivery of messages and their filtering. In particular, the remote delivery of messages has been provided

through both Java RMI [12] and JMS [13] communication technologies.

The second module contains both the software components that application developers extend, implement or, at least, use in their code, and the software components that help them in the deployment and execution of the realized applications. The identification of such software components can be easily done by analyzing what application developers need to realize: i) the actor and server classes used for the implementation of the processes involved in the application, ii) the description of selector classes used for the discovery of the processes involved in common tasks, iii) the message filter classes used for customizing the communication among the processes, iv) the typed messages used in the interaction among the processes, and v) the artifacts (i.e., Java classes and/or configuration files) for the deployment of the runtime nodes and of the communication channels among runtime nodes, and for the startup of the initial sets of processes and message filters.

The above items imply that such a module needs to contain; i) some software components for simplifying the realization of actors, servers, description selectors and message filters (realized through four abstract classes called AbstractActor, AbstractServer, AbstractSelector and AbstractFilter), ii) a set of abstract and concrete typed messages useful for realizing the typical communication protocols used in distributed applications, and iii) a software tool that allows the deployment of a HDS software application through the use of a set of configuration files (realized through a concrete class called Launcher).

In regard to type messages and the related communication protocols, the software framework provides the basis interfaces and classes for realizing application dependent client-server protocols and the basic interfaces and classes for supporting the interaction among processes through the use of communication language derived by the agent communication language (ACL) defined in the FIPA specifications [7].

In particular, besides realizing an implementation of the FIPA ACL, that completely satisfies the FIPA specifications and uses the SL language for the content, we used some of the FIPA ACL performatives as top layer interfaces for the definition of typed message classes that combines the semantics of the performative with the semantics of the ACL content in a particular ontology (e.g., the typed message, Sell, is sent to another process for requiring it to sell something to the requester; of course, typed messages are usually specialized for an application domain and it can be easily done grouping the typed messages related to a domain ontology in a Java package. In a similar way, we provided an abstract implementation of some interaction protocols (i.e. the English and Dutch auction protocols, the Contracted Net and the iterated Contract Net protocols and the brokering and the recruiting protocols) that derive from the interaction protocols defined in the FIPA specifications [7]. This implementation replaces the ACL messages with typed messages and delegates to the application developer only the duty of writing the code for processing the content of the messages, selecting the messages to be sent and building their content.

For example, the abstract implementation of the iterated Contract Net protocol is based on two abstract classes, that describe the two roles involved in the protocol, i.e., the initiator and the participant, and an interface, called Contract, used in the content of the exchanged messages for maintaining the information about both the task to be executed and the bids of the participants.

The abstract class that represents the initiator role defines three main methods; the first method sends an "offer" message to the list of processes acting as participants. The second method is an abstract method whose implementation must select the participant to which send either an "accept" or another "offer" message. Finally, the third method is an abstract method whose implementation must process the message containing the results of the execution of the required task.

The abstract class, that represents the participant role, defines two main methods. The first method is an abstract method whose implementation must decide to propose a bit for the task described by the "offer" message or to refuse it. The second message must decide to execute the task and then must send the information about the results of its execution.

Therefore, using the iterated Contract Net protocol inside an application requires: i) the definition of concrete class implementing the Contract interface, ii) the definition of a concrete class that extend the initiator abstract class implementing the methods for accepting, refusing or sending an updated contract and for processing the result received by the participant(s) to which the contract(s) have been assigned, and iii) the definition of at least a concrete class that extend the participant abstract class implementing the methods for accepting or refusing an offer and for performing the task associated with the contract.

V. HDS FOR DISTRIBUTED CONSTRAINT SATISFACTION

Recently we used HDS to develop a Java framework for prototyping and evaluating distributed constraint satisfaction algorithms. A brief introduction to distributed constraint satisfaction is needed to better explain the role of HDS; see [14] for an in-depth introduction to the subject and for a discussion of possible application scenarios.

Distributed Constraint Satisfaction Problems (*DCSPs*) are a very general class of problems that extend *Constraint Satisfaction Problems* (*CSPs*) to the realm of distributed computing; the literature defines DCSPs as a distributed and decentralized generalization of CSPs.

A CSP consists of *(i)* n variables $<x_1, x_2, \ldots , x_n>$, whose values are taken from finite, discrete domains $<D_1, D_2, \ldots , D_n>$, respectively, and *(ii)* a set of constraints on such variables. In very general terms, a constraint is defined by a relation on a subset of the Cartesian product $D_1 \times D_{k2} \times \ldots \times D_n$ that holds for certain assignments of values to variables. Solving a CSP is equivalent to finding an assignment of values to all variables such that all constraints are satisfied. Since constraint satisfaction is NP-complete in general, a trial-and-error exploration of alternatives is inevitable.

A DCSP is a CSP in which variables and constraints are distributed among agents; each agent has some owned variables and it tries to determine their values. Agents independently try to find assignments to their variables and the problem is solved when all variables are assigned consistent values.

More precisely, when dealing with DCSP, we take the following assumptions:

1. No central orchestration is allowed and the problem is solved by peer agents in cooperative/competitive ways.
2. Agents communicate by means of directed messages.
3. Each agent has a unique identifier and an agent can send messages to other agents if and only if it knows the unique identifiers of the receiving agents.
4. The delay in delivering a message is finite, though unknown and possibly random.
5. For the transmission between any pair of agents, messages are received in the order in which they were sent.
6. Each agent has exactly one variable and it knows all constraint predicates relevant to its variable.
7. All constraints are binary.

It is worth noting that although algorithms for solving DCSPs are similar to parallel/distributed processing methods for solving CSPs (see, e.g., [15] [16]), the applicability of both approaches is fundamentally different. The primary concern in parallel/distributed processing is efficiency, and we can choose any type of parallel/distributed computer architecture for solving a given problem efficiently. In contrast, in a DCSP, there already exists a situation where knowledge about the problem is distributed among agents and no central orchestration is available. This is the case, e.g., of sensor networks where nodes interact independently and strive to coordinate with no central master. If all knowledge about the problem could be gathered into a single master agent, such an agent could solve the problem more effectively alone by using every day, centralized constraint satisfaction algorithms.

The Asynchronous Backtracking Algorithm (ABT) is one of the algorithms, that we developed using HDS. This algorithm is a distributed, asynchronous version of a backtracking algorithm. The main message types communicated among agents are *ok?*, to communicate the current assigned value, and *nogood* to communicate a new constraint.

In the ABT algorithm, the priority order of agents is predetermined, and each agent communicates its tentative value assignment to neighboring agents via *ok?* messages. An agent changes its assignment if its current value assignment is not consistent with the assignments of higher priority agents. If there exists no value that is consistent with the higher priority agents, the agent generates a new constraint, called a *nogood*, and it communicates the *nogood* to a higher priority agent; thus the higher priority agent changes its value.

A *nogood* is a subset of an *agent view*, i.e., the current value assignment of other agents from its viewpoint, where the agent is not able to find any consistent value with the subset. Ideally, generated *nogood* should be minimal, i.e., no subset of them

should be a *nogood*. However, since finding minimal *nogood*s requires certain computation costs, an agent can do with non-minimal *nogood*s and, in the simplest case, it could use its entire agent view as a valid *nogood*.

It must be noted that since each agent acts asynchronously and concurrently and agents communicate by sending messages, the agent view may contain obsolete information. Therefore, if x_i does not have a consistent value with the higher priority agents according to its agent view, we cannot use a simple control method such as x_i orders a higher priority agent to change its value, since the agent view may be obsolete. Each agent needs to generate and communicate a new *nogood*, and the receiver of the new *nogood* must check whether the *nogood* is actually violated based on its own agent view.

The potential growth of the size of *nogood*s is a severe issue that went often unnoticed and that we identified during our initial experiments on solving Sudoku puzzles; this was the main reason why we switched our initial implementation from JADE to HDS. Moreover, we found HDS ideal for the implementation of this kind of algorithms because:

1. Performances are important as all such algorithms are typically demanding in terms of communication throughput;
2. Most of such algorithms are expressed in terms of reactions to typed messages; and
3. Composition filters allow instrumenting code with no modifications to developed algorithms, thus enabling performance measurement, debugging and fine tuning.

Finally, it is worth noting that HDS gave us a new dimension for experimentations, i.e., the impact of the underlying transport mechanism on the performances of algorithms. Actually, distributed constraint satisfaction algorithms use messages with very diverse sizes, ranging from few bytes in initial stages of the process to megabytes when agents send entire agent views across the network. We noted that different transport protocol exhibit different performances with massage sizes with strange and unforeseen behaviors.

VI. USING HDS IN SOCIAL NETWORK SYSTEMS

Moreover, we are using HDS for the realization of an agent based support layer for the interaction among users in a social network (SN). In particular, we associate an agent with each user and such an agent can also proactively act on her/his behalf by taking advantage the information contained in the profile of the user.

The agent has two main roles: i) it mediates access to the profile information, allowing or refusing queries from other agents; ii) it uses information in the profile in order to discover new friendships and acquaintances on his owner's behalf. While the first role does not need a full-fledged software agent, since a simple rule-based strategy suffices, the second role exhibits a typical proactive behavior, as agents actively pursue their owner's goal, without direct human intervention.

Currently available SNs are implemented with centralized systems where information is stored on a logical central server and users simply connect to that server. The system as a whole

has proactive behavior proposing the users new acquaintances, but its monolithic structure places the system outside the multi-agent paradigm. Moreover, the system has access to every piece of information users provided: this both raises security and privacy concerns and simplifies the proposal of new friendships.

We have designed a system where independent multiple proactive agents exchange minimal sets of data in order to discover relationship suggested by the user profiles. For example, if two users work in the same company, it is likely they know each other, thus they are to be connected in the SN.

Data are distributed among the agents and are protected by the agents themselves, since every access to a datum is mediated through an agent. Thus, privacy is not an issue.

The system supports "typed" connections, where the parts involved are aware they are connected, for example, because both attended to the same University or worked in the same company. In order to store data in the profile, we use FOAF [17] and DOAC [18] and the "type" of the connections is derived from those RDF descriptions. However, we do not detail the semantics of connections in order to focus the system presentation from a multi-agent modeling point of view.

The HDS framework is used as the foundation of our system because of its high efficiency and built-in support for typed messages, which are of paramount importance in expressing our connection negotiation algorithm.

Since the HDS framework distinguishes between active processes (actors) and servers, and since our agents feature both proactive and passive behavior, we decided to model the abstract agents with more than one concrete HDS process. Essentially, the agent discovery algorithm can be decomposed in three main tasks: i) search new connections and friendships according to the data available; ii) broker connections between possibly mutual friends iii) accept/refuse connections proposed by some other agent performing function i and ii. Tasks i) and ii) are clearly proactive, since the agent has to actively contact other agents, thus both tasks are implemented through HDS actors. Although task iii) is not proactive, and can be modeled with a server process. A passive server process mediates access to the profile and this can be seen as a fourth task.

Since agents are implemented through multiple processes, they are essentially only logical entities in the system, the only indication of their existence being a unique id in the system (such as, e.g., their owner's username) and rules granting full access among processes implementing the same agent.

We use capital letters to refer to the agents ids (e.g., A), and the same capital letter with a subscript (A_1, A_2, A_3, A_4) to refer to the HDS ids of the processes implementing the agent, e.g., A_1 implements the first task and so on.

In the following paragraphs we describe the connection discovery algorithm (Fig. 3), which is the component in our system that more heavily exploits HDS typed messages.
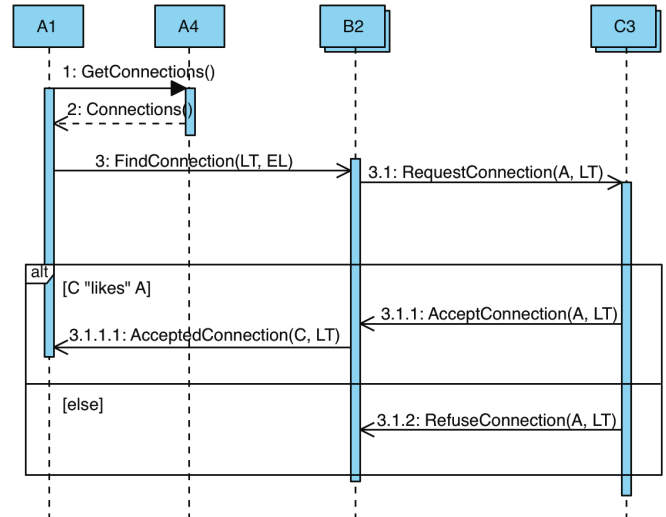

Figure 3. A sequence diagram presenting the connection discovery algorithm.

In order to describe the algorithm, we assume agent A wants to find new friends. As the first step, actor A_1 sends a GetConnections message to A_4 to obtain the list of connections. Each entry in the lists consists of an agent id and an RDF payload specifying the type of connection. Of course, the same pair of agents may be connected through multiple connections.

Let B an identifier in the list: A sends some FindConnection(LT, EL) messages to B_2, where each message has a different link type LT (derived from the types of links connecting A and B). B_2 is then entitled to share pieces of information derived from LT with every agent C that is connected with B through a LT connection and not present in the exclude list EL. EL contains both the ids of agents A is already connected with and the ids of agents A does not want to connect with.

Essentially B_2 acts as a broker between A and C, since mailer filters are configured not to accept connections from processes implementing unknown agents and consequently A and C cannot communicate directly. Notice that A determines the exact amount of information it wants to use in order to find new friends. B is not allowed to use information on A to find new friends for A or for himself until A allows usage of information contained in LT sending the FindConnection(LT, EL) message. For example, the link type is "attended University of Parma". C already knows that both he and B attended the University of Parma: A allowed B to inform C that A attended that University as well.

The next step consists in B_2 sending C_3 a RequestConnection(A, LT) message. If C_3 answers with a RefuseConnection(A, LT), B will not tell A that he is connected with C (and not even C existence). If C wants to be connected with A, C_3 sends an AcceptConnection(A, LT) to B_2 and consequently B_2 sends an AcceptedConnection(C, LT) message to A_3.

A can confirm the connection to C or refuse it. In the former situation, A_4 will be notified it is allowed to share some information with C and a direct negotiation between A and C

is started in order to establish further connections (or more specific connections, e.g. attended the University of Parma between 2002 and 2005, in place of the simple "attended University of Parma").

VII. Conclusion

This paper presented the HDS software framework, with the goal of simplifying the realization of distributed applications by merging the client-server and the peer-to-peer paradigms and by implementing the interactions among all the processes of a system through the exchange of typed messages.

HDS is implemented by using the Java language and its use simplify the realization of systems in heterogeneous environments where computers, mobile and sensor devices must cooperate for the execution of tasks. Moreover, since different protocols can be used to exchange messages between processes of different computational nodes, it is possible to use multiple implementations of the HDS framework for different languages in the same application as long as there are some shared protocols; this way it is possible to integrate hardware and software platforms without Java support.

HDS can be considered a software framework for the realization of any kind of distributed system. Some of its functionalities derive from the one offered by JADE [19][20], a software framework that can be considered one of the most known and used software framework for the developing of multi-agent systems. This derivation does not depend only on the fact that some of the people involved in the development of the HDS software framework were involved in the development of JADE too, but because HDS proposes a new view of multi-agent systems where the respect of the FIPA specifications are not considered mandatory and ACL messages can be expressed in a way that is more usable by software developers outside the multi-agent system community. This work may be of interest not only for enriching other theories and technologies with some aspects of multi-agent system theories and technologies, but also for providing new opportunities for the diffusion of both the knowledge and use of multi-agent system theory and technologies.

HDS is a suitable software framework for the realization of pervasive applications. Some of its features introduced above (i.e., the java implementation, the possibility of using different communication protocols and the possibility a multi-language implementation) are fit for such kinds of application. However, the combination of multi-agent and aspect-oriented techniques [21] might be one of the best solutions for providing an appropriate adaptation level in a pervasive application. In fact, this solution allows to couple the power of multi-agent based solutions with the simplicity of compositional filters solutions guaranteeing both a good adaptation to the evolution of the environment and a limited overhead to the performances of the applications.

Current and future research activities are dedicated, besides to continue the experimentation and validation of the HDS software framework in the realization of collaborative services for social network, to the improvement of the HDS software framework. In particular, current activities are dedicated to: i) the automatic creation of the Java classes representing the typed messages from OWL ontologies taking advantage of the O3L software library [22], and iii) the extension of the software framework with a high-performance software library to support the communication between remote processes, i.e., MINA [23].

References

[1] Genesereth, M.R., Ketchpel, S.P. Software agents. Communications of ACM, 37(7): 48-53, 1994.

[2] Genesereth, M.R. An agent-based framework for interoperability. In J. M. Bradshaw (Ed.). Software Agents, pp. 317-345. MIT Press, Cambridge, MA, 1997.

[3] O'Brien, P.D., Nicol, R.C. FIPA - Towards a Standard for Software Agents. BT Technology Journal, 16(3):51-59. 1998.

[4] Finin, T., Fritzson, R., McKay, D. McEntire, R. KQML as an agent communication language. In Proc. of the 3rd Int. Conf. on information and Knowledge Management, pp. 456-463, Gaithersburg, MD, 1994.

[5] Labrou, Y., Finin, T., Peng, Y. Agent Communication Languages: The Current Landscape. IEEE Intelligent Systems, 14(2):45-52, 1999.

[6] Singh, M.P. Agent Communication Languages: Rethinking the Principles. IEEE Computer, 31(12):40-47, 1998.G. O. Young, "Synthetic structure of industrial plastics (Book style with paper title and editor)," in Plastics, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964, pp. 15–64.

[7] FIPA Consortium. FIPA Specifications. Available from http://www.fipa.org.

[8] Collier, R.: Agent Factory: An Environment for the Engineering of Agent-Oriented Applications. Ph.D. Thesis, University College Dublin, Ireland, 2001

[9] Mulet, L., Such, J. M., and Alberola, J. M. 2006. Performance evaluation of open-source multiagent platforms. Proceedings of the Fifth international Joint Conference on Autonomous Agents and Multiagent Systems, Hakodate, Japan, , 2006.

[10] G. Vitaglione, F. Quarta, and E. Cortese, "Scalability and performance of jade message transport system," 2002.

[11] Bergmans, L., Aksit, M. Composing crosscutting concerns using composition filters. Communications of ACM, 44(10):51-57, 2001.

[12] Pitt, E. McNiff, K. Java.rmi: the Remote Method Invocation Guide. Addison-Wesley, 2001.

[13] Monson-Haefel, R. Chappell, D. Java Message Service. O'Reilly & Associates, 2000.

[14] Yokoo, M., Katsutoshi, H. Algorithms for Distributed Constraint Satisfaction: A Review, Procs. Int'l Conf. Autonomous Agents and Multiagent Systems, Vol. 3, pp. 185-207, 2000.

[15] Zhang, Y., Mackworth, A. Parallel and distributed algorithms for finite constraint satisfaction problems. Procs. 3rd IEEE Symposium on Parallel and Distributed Processing. 394-397, 1991.\

[16] Collin, Z., Dechter, R. , Katz S. On the Feasibility of Distributed Constraint Satisfaction. Procs. 12th Int'l Joint Conference on Artificial Intelligence. 318-324, 1991.

[17] D. Brickley and L. Miller, http://www.foaf-project.org

[18] R. Antonio, http://ramonantonio.net/doac

[19] Bellifemine, F., Poggi, A., Rimassa, G. Developing multi agent systems with a FIPA-compliant agent framework. Software Practice & Experience, 31:103-128, 2001.

[20] Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.. JADE: a Software Framework for Developing Multi-Agent Applications. Lessons Learned. Information and Software Technology Journal, 50:10-21, 2008.

[21] Kiczales, Gregor; John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin (1997). "Aspect-Oriented Programming". Proceedings of the European Conference on Object-Oriented Programming, vol.1241. pp. 220.242. The paper generally considered to be the authoritative reference for AOP.

[22] Poggi, A. Developing Ontology Based Applications with O3L. WSEAS Trans. on Computers, 8(8):1286-1295, 2009

[23] Apache Foundation. MINA software. Web site. Available from: http://mina.apache.org.