Replaceable Implementations for Actor Systems

Agostino Poggi
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
Parma, Italy
agostino.poggi@unipr.it

***Abstract* — CoDE is an actor-based software framework aimed at both simplifying the development of large and distributed complex systems and guarantying an efficient execution of applications. This software framework takes advantage of a concise actor model that makes easy the development of the actor code by delegating the management of events (i.e., the reception of messages) to the execution environment. Moreover, it allows the development of scalable and efficient applications through the possibility of using different implementations of the components that drive the execution of actors. This paper introduces the software framework and shows how the performance of applications can be optimized by choosing the best combination among the alternative implementations of its components.**

Keywords - Actor model, software framework, concurrent programming, distributed systems.

I. INTRODUCTION

Distributed and concurrent programming have lately received enormous interest because multi-core processors make concurrency an essential ingredient of efficient program execution and because distributed architectures are inherently concurrent. However, distributed and concurrent programming is hard and largely different from sequential programming. Programmers have more concerns when it comes to taming parallelism. In fact distributed and concurrent programs are usually bigger than equivalent sequential ones and models of distributed and concurrent programming languages are different from familiar and popular sequential languages [12][14].

Message passing models seem be the more appropriate solution because they replace the sharing of data with the exchange of messages. One of the well-known theoretical and practical models of message passing is the actor model. Using such a model, programs become collections of independent active objects (actors) that exchange messages and have no mutable shared state [1][2][9]. Actors can help developers to avoid issues such as deadlock, live-lock and starvation, which are common problems for shared memory based approaches. There are a multitude of actor oriented libraries and languages, and each of them implements some variants of actor semantics. However, such libraries and languages use either thread-based programming, which makes easy the development of programs, or event-based programming, which is far more practical to develop large and efficient concurrent systems, but also is more difficult to use.

This paper presents an actor based software framework, called CoDE (Concurrent Development Environment), that has the suitable features for both simplifying the development of large and distributed complex systems and guarantying scalable and efficient applications. The next two sections introduce the software framework and its implementation. Section 4 shows how the possibility of configuring an application with different implementations of its components allows coping with performance and scalability problems. Section 5 introduces two simple applications and shows their execution times obtained with different configurations. Section 6 introduces related work. Finally, section 7 concludes the paper by discussing the main features of the software framework and the directions for future work.

II. CODE

In CoDE a system is based on a set of interacting actors that perform tasks concurrently. An actor is an autonomous concurrent object, which interacts with other actors by exchanging asynchronous messages. Communication between actors is buffered: incoming messages are stored in a mailbox until the actor is ready to process them. Each actor has a system-wide unique identifier called its address that allows it to be referenced in a location transparent way. An actor can send messages only to the actors of which it knows the address, that is, the actors it created and of which it received the addresses from other actors. After its creation, an actor can change several times its behavior until it kills itself. Each behavior has the main duty of processing a set of specific messages through a set of message handlers called cases. Therefore, if an unexpected message arrives, then the actor mailbox maintains it until a next behavior will be able to process it.

An actor can perform five types of action:

- It can send messages to other actors or to itself.
- It can create new actors.
- It can update its local state.
- It can change its behavior.
- It can kill itself.

In particular, an actor has not explicit actions for the reception of messages, but its implementation autonomously extracts the new messages from the actor mailbox and then executes the actions for their processing.

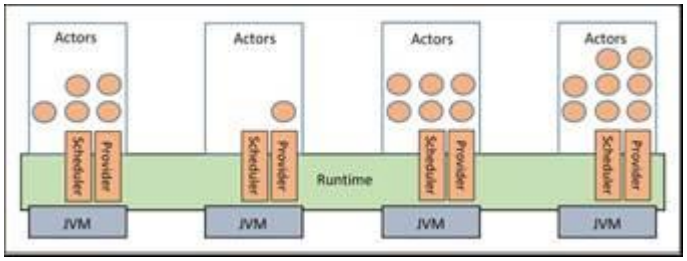

Fig. 1. CoDE distributed system architecture.

An actor can set a timeout for waiting for the next message and then execute some actions if the timeout fires. However, it has not explicit actions for monitoring the firing of such a timeout: its implementation autonomously observes the firing of the timeout and then executes the actions for its management.

Depending on the complexity of the application and on the availability of computing and communication resources, one or more actor spaces can manage the actors of the application. An actor space acts as "container" for a set of actors and provides them the services necessary for their execution. In particular, an actor space takes advantages of two special actors: the scheduler and the service provider. The scheduler manages the concurrent execution of the actors of the actor space. The service provider enables the actors of an application to perform new kinds of action (e.g., to broadcast a message or to move from an actor space to another one). Fig. 1 shows a graphical representation of the architecture of a CoDE distributed application.

III. IMPLEMENTATION

CoDE is a software environment implemented by using the Java language and takes advantage of preexistent Java software libraries and solutions for supporting concurrency and distribution. CoDE has a layered architecture composed of an application and a runtime layer. The application layer provides the software components that an application developer needs to extend or directly use for implementing the specific actors of an application. The runtime layer provides the software components that implement the CoDE middleware infrastructures to support the development of standalone and distributed applications.

A. Actor Implementation

An actor can be viewed as a logical thread that implements an event loop [4][13]. This event loop perpetually processes events that represent: the reception of messages, the behavior exchanges and the firing of timeouts. In particular, an actor is defined by five main components: a reference, a mailer, a behavior, a state and an execution manager. Fig. 2 shows a graphical representation of the architecture of an actor.

A reference supports the sending of messages to the actor it represents. Therefore, an actor needs to have the reference of another actor for sending it a message. In particular, an actor has have the reference of another actor if:

- It created such an actor (in fact, the creation method returns the reference of the new actor).
- It received a message from such an actor (in fact, each message contains the reference of the sender) or whose content enclosed its reference.

A reference has an attribute, called actor address, that allows to distinguish itself (and then the actor it represents) from the references of the other actors of the application where it is acting. To guarantee it and to simplify the implementation, an actor space acts as "container" for the actors running in the same Java Virtual Machine (JVM) and an actor address is composed of three components:

- An actor identifier that is different for all the actors of the same actor space.
- An actor space identifier that is different for all the actor spaces of the same computing node.
- The IP address of the computing node.

A mailer provides a mailbox for the messages sent to its actor until it processes them, and delivers its messages to the other actors of the application. As introduced above, a behavior can process a set of specific messages leaving in the mailbox the messages that is not able to process. Such messages remain into the mailbox until a new behavior is able to process them and if there is not such a behavior they remain into the queue for all the life of the actor. A mailbox has not an explicit limit on the number of messages that can maintain. However, it is clear that the (permanent) deposit of large numbers of messages in the mailboxes of the actors may reduce the performances of applications and cause in some circumstances their failure.

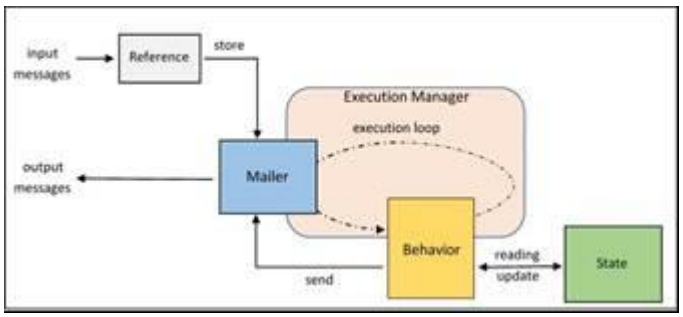

Fig. 2. Actor Architecture.

The original actor model associates a behavior with the task of messages processing. In CoDE, a behavior can perform three kinds of tasks: its initialization, the processing of messages and the management of message reception timeouts. In particular, a behavior does not directly process messages, but it delegates the task to some case objects, that have the goal of processing the messages that match a specific (and unreplaceable) message pattern.

Often the behaviors of an actor need to share some information (e.g., a behavior may work on the results of the previous behaviors). It is possible thank to a state object. Of course, the kind of information that the behaviors of an actor need to share depends on the type of tasks they must perform in an application. Therefore, the state of an actor must be specialized for the task it will perform.

A message is an object that contains a set of fields maintaining the typical header information and the message content. Moreover, each message is different from any other one. In fact, messages of the same sender have a different identifier and messages of different senders have a different sender reference.

An actor has not direct access to the local state of the other actors and can share data with them only through the exchange of messages and through the creation of actors. Therefore, to avoid the problems due to the concurrent access to mutable data, both message passing and actor creation should have call-by-value semantics. This may require making a copy of the data even on shared memory platforms, but, as it is done by the large part of the actors libraries implemented in Java, CoDE does not make data copies because such operations would be the source of an important overhead. However, it encourages the programmers to use immutable objects (by implementing as immutable all the predefined message content objects) and delegates the appropriate use of mutable object to them.

As introduced above, an actor behavior processes the received messages through a set of case objects and each of them can process only the messages that match a specific message pattern. In CoDE, a message pattern is an object that can apply a combination of constraint objects on the value of all the fields of a message and on the actor state. It improves the adaptability of actors to the changes of the environment they live. In fact, an actor can react to the changes by either moving to another behavior or by enabling, disabling or changing the cases that process the received messages depending on their current state.

An execution manager implements the basic functionalities of an actor on the top of the services provided by the runtime layer. In particular, it manages the life cycle of the actor by initializing its behaviors, by processing the received messages and the firing of message reception timeouts, and by moving it from a behavior to another one. The type of the implementation of an execution manager is one of the factors that mainly influence the attributes of the execution of an application. In particular, execution managers can be divided in two classes that allow to an actor either to have its own thread (from here named active actors) or to share a single thread with the other actors of the actor space (from here named passive actors).

B. Actor Space Implementation

An actor space has the duty of supporting the execution of the actions of its actors and of enhancing them with new kinds of action. To do it, an actor space takes advantage of some main runtime components (i.e., factory, dispatcher and registry) and of the two special actors: the scheduler and the service provider.

The factory has the duty of creating the actors of the actor space. In particular, it also creates their initial behavior, chooses their most appropriate execution manager and delegates the creation of their references to the registry.

The dispatcher has the duty of supporting the communication with the other actor spaces of the application. In particular, it creates connections to/from the other actor spaces, maps remote addresses to the appropriate output connections, manages the reception of messages from the input connections, and delivers messages through the output connections. This component works in collaboration with another component called connector.

A connector has the duty of opening and maintaining connections toward all the other actor spaces of the application. In particular, the connector of one of the actor spaces of the application plays the role of communication broker and has the additional duty of maintaining the information necessary to a new actor space for creating connections towards the other actor spaces of the application.

The registry supports the work of both the factory and the dispatcher. In fact, it creates the references of the new actors and supports the delivery of the messages coming from remote actor by proving the reference of the destination actor to the dispatcher. In fact, as introduced in a previous section an actor can send a message to another actor only if it has its reference, but while the reference of a local actor allows the direct delivery of messages, the reference of a remote actor delegates the delivery to the dispatchers of the local and remote actor spaces (see Fig. 3).

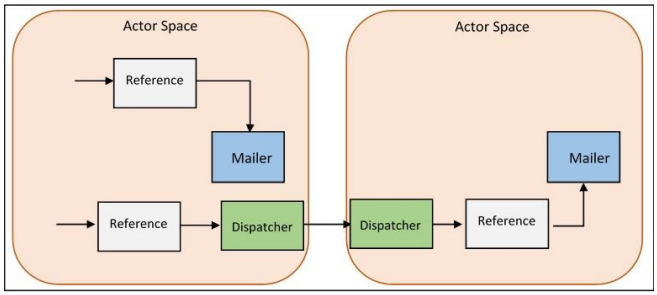

Fig. 3. Message dispatching.

The scheduler is a special actor that manages the execution of the actors of an actor space. Of course, the duties of a scheduler depend on the type of execution manager and, in particular, on the type of threading solutions associated with the actors of the actor space. In fact, while Java runtime environment mainly manage the execution of active actors, CoDE schedulers completely manage the execution of passive actors.

The service provider is a special actor that offers a set of services for enabling the actors of an application to perform new kinds of actions. Of course, the actors of the application can require the execution of such services by sending a message to the service provider.

Moreover, an actor space can enable the execution of an additional runtime component called logger. The logger has the possibility to store (or to send to another application) the relevant information about the execution of the actors of the actor space (e.g., creation and deletion of actors, exchange of messages, processing of messages and timeouts, exchange of behaviors). The logger can provides both textual and binary information that can be useful for understanding the activities of the application and for diagnosing the causes and solving the possible execution problems. Moreover, the binary information contain real copies of the objects of the application (e.g.,

messages and actor state); therefore, such an information can be used to feed other applications (e.g., monitoring and simulation tools).

Finally, the actor space provides a runtime component, called configurator, which simplifies the configuration of an application by allowing the use of either a declarative or a procedural method (i.e., the writing of either a properties file or a code that calls an API provided by the configurator).

IV. CONFIGURATION

One of the most important features of CoDE is the possibility of configuring an application with different implementations of the runtime components. For example, CoDE supports the communication among the actor spaces through four kinds of connector that respectively use ActiveMQ [23], Java RMI [15], MINA [3] and ZeroMQ [11]. Moreover, the service provider actor can offer an extensible set of services for enhancing the set of actions that the actors can perform. The current implementation of the software framework provides services for supporting the broadcast of messages, the exchange of messages through the "publish and subscribe" pattern, the mobility of actors, the interaction with users through emails and the creation of actors (useful for creating actors in other actor spaces).

However, the most important components that influence the quality of the execution of an application are the execution manager and the associated scheduling actor. In fact, the use of one or another couple of execution manager and scheduling actor causes large differences in the performance and in the scalability of the applications.

CoDE provides three types of execution managers and four types of execution scheduling actors. The first two types of execution manager respectively support the implementation of active and passive actors (active and passive executors). The third type provides a special implementation of passive actors in which the actors receive message from a shared queue (shared executors). The first two types of scheduling actors manage the execution of the actor spaces, which contain either active or passive actors (active and passive schedulers). The third type manages the execution of the actor spaces where passive actors share the message queue (shared schedulers). Finally, the forth type manages the execution of the actor spaces where both active and passive actors are present (hybrid schedulers).

The identification of the best couple of execution manager and scheduling actor for a specific application mainly depends on the number of actors, the number of exchanged messages, the preeminent type of communication used by actors (i.e., point-to-point or broadcast) and the presence of a subset of actors that consume a large part of the computational resources of the application. Table 1 shows what should be the best choices for binary partition of the values of the previous parameters. In particular, the third column indicates the preeminence of either point-to-point communication (P) or broadcast communication (B), the forth column indicates the presence/absence of a subset of heavy actors and the word "any" is used when the value of the associate parameter has not effect on the choice of execution manager and scheduler.

TABLE 1

actors	messages	P/B	Heavy	scheduler
few	any	any	any	active
many	any	P	no	passive
many	few	B	no	passive
many	many	B	no	shared
many	any	any	yes	hybrid

V. EXPERIMENTATION

The performances of the different types of execution managers and scheduling actors can be analyzed by comparing the execution times of two simple applications on a laptop with an Intel Core 2 - 2.90GHz processor, 16 GB RAM, Windows 8 OS and Java 7 with 4 GB heap size.

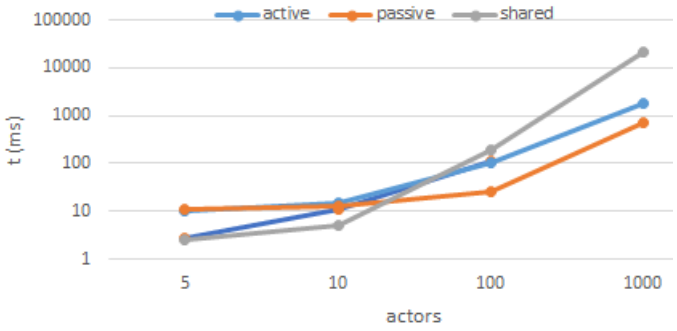

Fig. 4. Point-to-point message exchange example performances.

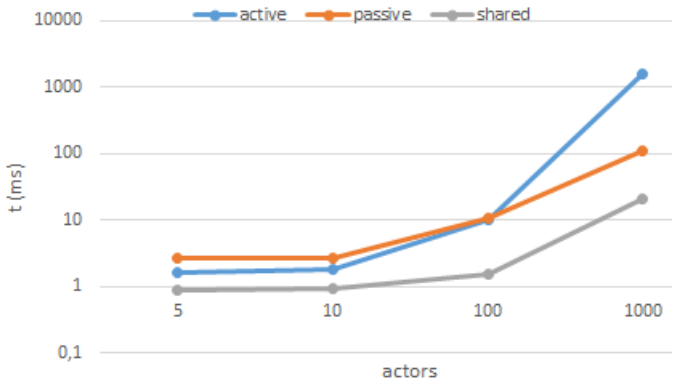

Fig. 5. Broadcasting example performances.

The first application is based on the point-to-point exchange of messages between the actors of an actor space. The application starts an actor that creates a certain number of actors, sends 1000 messages to each of them and then waits for their answers. Fig 4 shows the execution time of the application for 5, 10, 100 and 1.000 actors and, as introduced in table 1, the best performances are obtained with a passive executor and scheduling actor couple when the number of actors increases.

The second application is based on the broadcasting of messages to the actors of an actor space. The application starts an actor that creates a certain number of actors and then sends a broadcast message. Each actor receives the broadcast message,

then, in its response, sends another broadcast message and finally waits for all the broadcast messages. Fig. 5 shows the execution time of the application for 5, 10, 100 and 1.000 actors and, as introduced in table 1, the best performances are obtained with a shared executor and scheduling actor couple.

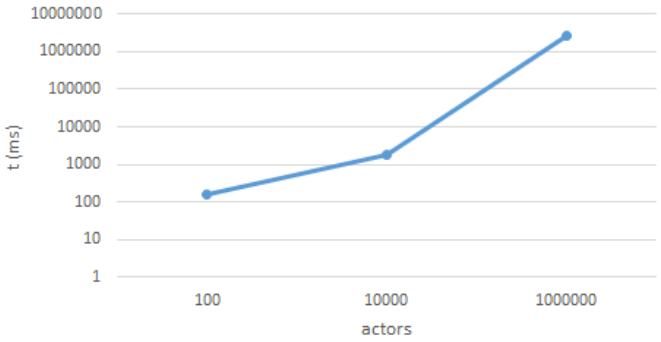

Fig. 6. Game of life performances.

Moreover, the use of passive actors allows the development of applications that scale to a large number of actors. In particular, the current implementation of the framework allows to scale up to a million of actors. Fig.6 show the execution times for 100 cycles of simulation of the game of live [8] for 100, 10.000 and and 1.000.000 actors.

VI. Related Work

Several actor-oriented libraries and languages have been proposed in last decades and a large part of them uses Java as implementation language. The rest of the section presents some of the most interesting works.

Salsa [27] is an actor-based language for mobile and Internet computing that provides three significant mechanisms based on the actor model: token-passing continuations, join continuations, and first-class continuations. In Salsa each actor has its own thread, and so scalability is limited. Moreover, message-passing performance suffers from the overhead of reflective method calls.

Kilim [24] is a framework used to create robust and massively concurrent actor systems in Java. It takes advantage of code annotations and of a byte-code post-processor to simplify the writing of the code. However, it provides only a very simplified implementation of the actor model where each actor (called task in Kilim) has a mailbox and a method defining its behavior. Moreover, it does not provide remote messaging capabilities.

Scala [9] is an object-oriented and functional programming language that provides an implementation of the actor model unifying thread based and event based programming models. In fact, in Scala an actor can suspend with a full thread stack (receive) or can suspend with just a continuation closure (react). Therefore, scalability can be obtained by sacrificing program simplicity. Akka [26] is an alternative toolkit and runtime system for developing event-based actors in Scala, but also providing APIs for developing actor-based systems in Java. One of its distinguishing features is the hierarchical organization of actors, so that a parent actor that creates some children actors is responsible for handling their failures.

Jetlang [22] provides a high performance Java threading library that should be used for message based concurrency. The library is designed specifically for high performance in-memory messaging and does not provide remote messaging capabilities.

AmbientTalk [4] is a distributed object-oriented programming language whose actor-based and event driven concurrency model makes it highly suitable for composing service objects across a mobile network. It provides an actor implementation based on communicating event loops [13]. However, each actor is always associated with its own JVM thread and so it limits the scalability of applications on the number of actors for JVM.

VII. Conclusions

This paper presented a software framework, called CoDE, which allows the development of efficient large actor based systems by combining the possibility to use different implementations of the components driving the execution of actors with the delegation of the management of the reception of messages to the execution environment.

CoDE is implemented by using the Java language and is an evolution of HDS [19] and ASIDE [20] from which it derives the concise actor model, and takes advantages of some implementation solutions used in JADE [16]. CoDE shares with Jetlang, [22] Kilim [24] and Scala [9] the possibility to build applications that scale applications to a massive number of actors, but without the need of introducing new constructs that make complex the writing of actor based programs. Moreover, CoDE has been designed for the development of distributed applications while the previous three actor based software were designed for applications running inside multi-core computers. In fact, the use of structured messages and message patterns makes possible the implementation of complex interactions in a distributed application because a message contains all the information for delivery it to the destination and then for building and sending a reply. Moreover, a message pattern filters the input messages on all the information contained in the message and not only on its content.

Current research activities are dedicated to extend the software framework to offer it as means for the development of multi-agent systems. Future research activities will be dedicated to the extension of the functionalities provided by the software framework and to its experimentation in different application fields. Regarding the extension of the software framework, current activities have the goal of providing a passive threading solution that fully take advantage of the features of multi-core processors, of enabling the interoperability with Web services and legacy systems [18], and of enhancing the definition of the content exchanged by actors with semantic Web technologies [21]. Moreover, future activities will be dedicated to the provision of a trust management infrastructure to support the interaction between actor spaces of different organizations [17][25]. Current experimentation of the software framework is performed in the field of the modeling and simulation of social networks [6], but in the next future will be extended to the collaborative work

services [4] and to the agent-based systems for the management of information in pervasive environments [4].

REFERENCES

[1] G.A. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, Cambridge, MA: MIT Press, 1986.

[2] G.A. Agha, I.A. Mason, S.F. Smith and C.L. Talcott, "A Foundation for Actor Computation," J. of Functional Programming, vol. 7, no. 1, pp. 1-69, 1997.

[3] Apache Software Foundation. (2013). Apache Mina Framework [Online]. Available: http://mina.apache.org

[4] F. Bergenti and A. Poggi, "Ubiquitous Information Agents," Int. J. on Cooperative Information Systems, vol. 11, no. 3-4, pp. 231-244, 2002.

[5] F. Bergenti, A., Poggi and M. Somacher, "A collaborative platform for fixed and mobile networks," Communications of the ACM, vol. 45, no. 11, pp. 39-44, 2002.

[6] F. Bergenti, E. Franchi and A. Poggi, "Selected models for agent-based simulation of social networks," in 3rd Symposium on Social Networks and Multiagent Systems (SNAMAS'11), York, UK: Society for the Study of Artificial Intelligence and the Simulation of Behaviour, 2011. pp. 27-32.

[7] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt and W. De Meuter, "Ambient-oriented programming in ambienttalk," in ECOOP 2006 – Object-Oriented Programming, Berlin Heidelberg: Springer, 2006, pp. 230-254.

[8] M. Gardner, The fantastic combinations of John Conway's new solitaire game Life. Scientific American 223:120-123, 1970.

[9] P. Haller and M. Odersky, "Scala Actors: Unifying thread-based and event-based programming," Theoretical Computer Science, vol. 410, no. 2-3, pp. 202–220, 2009.

[10] C. E. Hewitt, "Viewing controll structures as patterns of passing messages," Artificial Intelligence, vol. 8, no. 3, 1977, pp. 323–364.

[11] P. Hintjens, ZeroMQ: Messaging for Many Applications, Sebastopol, CA: O'Reilly, 2013.

[12] C. Leopold, Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches, New York, NY, USA: John Wiley & Sons, 2001.

[13] M. S. Miller, E. D. Tribble and J. Shapiro, "Concurrency among strangers," in Trustworthy Global Computing, Berlin Heidelberg: Springer, 2005, pp. 195-229.

[14] M. Philippsen, "A survey of concurrent object-oriented languages," Concurrency: Practice and Experience, vol. 12, no. 10, pp. 917-980, 2000.

[15] E. Pitt and K. McNiff, Java.rmi: the Remote Method Invocation Guide. Boston, MA, USA: Addison-Wesley, 2001.

[16] A. Poggi, M. Tomaiuolo and P. Turci, "Extending JADE for agent grid applications," in 13th IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2004), Modena, Italy, 2004, pp. 352-357.

[17] A. Poggi, M. Tomaiuolo and G. Vitaglione, "A Security Infrastructure for Trust Management in Multi-agent Systems," in Trusting Agents for Trusting Electronic Societies, Theory and Applications in HCI and E-Commerce, LNCS, vol. 3577, R. Falcone, S. Barber, and M. P. Singh, Eds. Berlin, Germany: Springer, 2005, pp. 162-179.

[18] A. Poggi, M. Tomaiuolo and P. Turci, "An Agent-Based Service Oriented Architecture", in Proc of. WOA, Genova, Italy, 2007, pp. 157-165.

[19] A. Poggi, "HDS: a Software Framework for the Realization of Pervasive Applications," WSEAS Trans. on Computers, vol. 10, no. 9, pp. 1149-1159, 2010.

[20] A. Poggi, "ASiDE - A Software Framework for Complex and Distributed Systems," in Proc. of the 16th WSEAS Int. Conf. on Computers, Kos, Greece, 2012, pp. 353-358.

[21] A. Poggi, "Developing ontology based applications with O3L," WSEAS Trans. on Computers, vol. 8 no. 8, pp. 1286-1295, 2009.

[22] M. Rettig. (2013). Jetlang software [Online]. Available: http://code.google.com/p/jetlang/

[23] B. Snyder, D. Bosnanac and R. Davies, ActiveMQ in action, Westampton, NJ, USA: Manning, 2001.

[24] S. Srinivasan and A. Mycroft, "Kilim: Isolation-Typed Actors for Java," in ECOOP 2008 – Object-Oriented Programming, LNCS, vol. 5142, J. Vitek, Ed. Berlin ,Germany: Springer, 2008, pp. 104-128.

[25] M. Tomaiuolo, "dDelega: Trust Management for Web Services," Int. J. of Information Security and Privacy, vol. 7, no. 3, pp. 53-67, 2013.

[26] Typesafe. (2013) Akka software [Online]. Available: http://akka.io

[27] C. Varela and G.A. Agha, "Programming dynamically reconfigurable open systems with SALSA," SIGPLAN Notices, vol. 36, no. 12, pp. 20-34, 2001.