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.

4.1 Actor

An actor can be viewed as a logical thread that implements an event loop [10][11]. This event loop perpetually processes events that represent: the reception of messages, the behavior exchanges and the firing of timeouts. The life of an actor starts from the initialization of its behaviors that then processes the received messages and the firing of message reception timeouts. During its life, an actor can move from a behavior to another one more times and its life ends when it kills itself.

CoDE provides different actor implementations and the use of one or of another implementation represents one of the factors that mainly influence the performance of an application. In particular, actor implementations can be divided in two classes that allow to an actor either to have its own thread of execution (from here named active actors) or to share a single thread of execution with other actors of the actor space (from here named passive actors). In this last case, the scheduler has the duty of guaranteeing a fair execution of the actors of the actor space.

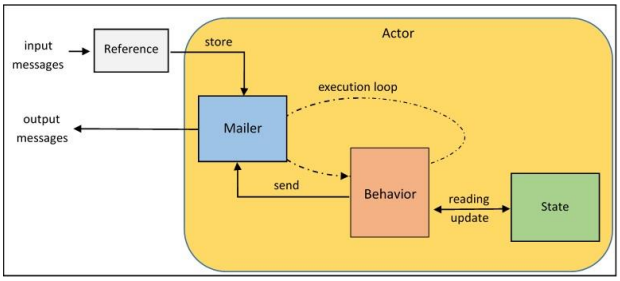

Fig. 2. Architecture of an actor.

In particular, the implementation of an actor is based on four main components: a reference, a mailer, a behavior and a state. 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 the reference of another actor if either it created such an actor (in fact, the creation method returns the reference of the new actor) or it received a message that either has been sent by such an actor (in fact, each message contains the reference of the sender) or whose content enclosed its reference.

References act as identifiers of the actors of an application. 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 the string representation of a reference is composed of an actor identifier, an actor space identifier and the IP address of the computing node. In particular, the actor identifier is different for all the actors of the same actor space, and the actor space identifier is different for all the actor spaces of the same computing node.

A mailer provides a mailbox for the messages sent to its actor until it processes them, and delivers the output messages of its actor 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 may cause in some circumstances their failure.

As in the original actor model, in CoDE, a behavior had the main duty of processing the incoming messages. It 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 that drive the life 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 tasks it will perform (i.e., different behaviors can have different state representations).

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.

A message pattern is an object that can apply a combination of constraint objects on the value of all the fields of a message. CoDE provides a set of predefines constraints, but new ones can be easily added. In particular, one of such constraints allows the application of a pattern to the value of a message field. Therefore, the addition of field patterns (the current implementation offer only a regular expression pattern) will allow the definition of sophisticated filters on the values of all the message fields and in particular on the content of the message.

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.

4.2 Actor space

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 two main runtime components (i.e., the dispatcher and the registry) and of two special actors (the scheduler and the service provider).

The dispatcher has the duty of supporting the communication with the actors of the other actor spaces of the application. In particular, it creates connections to/from the other actor spaces, manages the reception of messages from the input connections, maps remote references to the appropriate output connections, and delivers messages through the output connections.

The registry supports the creation of actors and the reception of the messages coming from remote actors. In particular, it has the duties of creating new references and of providing the reference of a local actor to the dispatcher when it is managing a message coming from a remote actor. In fact, while the reference of a local actor allows the direct delivery of messages, the reference of a remote actor delegates the delivery of messages to the dispatcher of the actor space. Such a dispatcher delivers the message to the dispatcher of the actor space where the remote actor lives and the latter dispatcher takes advantage of the registry for mapping the remote reference to the local reference of the actor.

The scheduler is a special actor that manages the execution of the "normal" actors of an actor space. Of course, the duties of a scheduler depend on the implementation of the actors of the actor space and, in particular, on the type of threading solutions associated with them. In fact, while the Java runtime environment mainly manages 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 "normal" 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. In particular, 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).

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 and processing of messages, and behavior replacements). The logger can provides both textual and binary information that can be useful for understanding the activities of the application and for identifying the causes and of possible execution problems. In particular, the binary information contains 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, whose duty is to simplify 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).

4.3 Configuration Profiles

The quality of the execution of a CoDE application mainly depends on the implementation of the actors and of the schedulers of its actor spaces. Another important factor that influences its execution is the implementation of the runtime components that support the exchange of messages between both local and remote actors.

However, a combination of such implementations, that maximizes the quality of execution of an application, could be a bad configuration for another type of application. Moreover, different instances of the same application can work in different conditions (e.g., different number of users to serve, different amount of data to process) and so they may require different configurations.

As introduced in a previous section, actor implementations can be divided in two classes that allow to an actor either to have its own thread (active actor) or to share a single thread with the other actors of the actor space (passive actor).

The use of active actors has the advantage of delegating the scheduling to the JVM with the advantage of guaranteeing actors to have a fair access to the computational resources of the actor space.

However, this solution suffers from high memory consumption and context-switching overhead and so it can be used in actor spaces with a limited number of actors. Therefore, when the number of actors in an actor space is high, the best solution is the use of passive actors whose execution is managed by a scheduler provided by the CoDE framework. Such a scheduler uses a simple not preemptive round-robin scheduling algorithm and so the implementation of the passive actor has the duty of guaranteeing a fair access to the computational resources of the actor space, for example, by limiting the number of messages that an actor can process in a single execution cycle.

Moreover, is some particular applications is not possible to distribute in equal parts the tasks among the actors of an actor space and so there are some actors that should have a priority on the access to the computational resources of the actor space. Often in this situation, a good solution is the combination of active and passive actors.

In an actor-based system where the computation is mainly based on the exchange and processing of messages, the efficiency of the communication supports are a key parameter for the quality of applications. In CoDE both local and remote communication can be provided by replaceable components. In particular, the current implementation of the software framework supports the communication among the actor spaces through four kinds of connector that respectively use ActiveMQ [12], Java RMI [13], MINA [14] and ZeroMQ [15]. Moreover, when in an application the large part of communication is based on broadcast and multicast messages, the traditional individual mailbox can be replaced by a mailbox that transparent extracts the messages for its actor from a single queue shared with all the other actors of the actor space.

5 Application Development

The development of an application involves the design and the coding of the Java classes defining the different behaviors of the actors involved in the application and the configuration of its actor spaces.

```
public final class EmptyBuffer extends Behavior {
  public List<Case> initialize(final Object[] v) {
    BufferState s = new BufferState();

    s.setCapacity((Integer) v[0]);
    setState(s);

    return initialize();
  }

  public List<Case> initialize() {
    ArrayList<Case> l = new ArrayList<>();

    l.add(new PutItem());
    l.add(new Killer());

    return l;
  }
}
```

Fig. 3. "EmptyBuffer" behavior.

For example, the modelling of the classical bounded buffer problem involves the definition of the behaviors that drive the actors representing the bounded buffer and the producers and consumers acting on it. In particular, the "EmptyBuffer", "PartialBuffer" and "FullBuffer" behaviors can drive the execution of the bounded buffer, and the "Producer" and Consumer" behaviors can respectively drive the execution of the producers and of the consumers. Each behavior has some cases for processing the input messages, and each behavior of the bounded buffer can move to one of the other two when its state (empty, partial and full) changes. Moreover, is necessary an additional actor whose behavior has only the goal of creating all the other actors of the application. Fig. 3 and Fig. 4 show the code of the "EmptyBuffer" behavior and of the "GetItem" case.

```
public final class GetIten extends Case {
  GetItem() {
    super(new MessagePattern(
        MessagePattern.CONTENT,
        new IsInstance(Get.class)));
  }
public void process(final Message m) {
    BufferState s = (BufferState) getState();
    send(m, s.remove());
    if (s.size() == 0) {
      become(EmptyBuffer.class);
    }
    else if (getBehavior().equals(
                FullBuffer.class.getName())) {
      become(PartialBuffer.class);
    }
  }
}
```

Fig. 4. "GetItem" case.

```
public static void main(final String[] v) {
  final long time = 1000;
  final int size = 10;
  final int producers = 10;
  final int consumers = 10;

  Configuration c =
    Controller.INSTANCE.getConfiguration();

  c.setScheduler(ThreadScheduler.class.getName());
  c.setArguments(
    Initiator.class.getName(),
    new Object[] {time, size, producers, consumers});

  Controller.INSTANCE.run();
}
```

Fig. 5. Application starting code.

After the writing of the code of the behaviors used by the actors of the application, the final step is the writing of the code that configures and starts the application. This code must configure the actor scheduler and call the runtime method that starts the application. Moreover, it may configure: i) the dispatcher for allowing the communication with the actors of other actor spaces, ii) the additional services that the service provider actor should offer to the "normal" actors of the application, and, iii) the logger. Fig. 5 shows the code that starts the application modelling the bounded buffer problem.

6 Performance Analysis

The performances of the different implementations of actors and scheduling actors can be analyzed by comparing the execution times of three 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. These examples involves four kinds of configuration: active (i.e., the actor space contains active actors), passive (i.e., the actor space contains passive actors), shared (i.e., the actor space contains passive actors whose mailboxes get messages from a unique message queue), and hybrid, (i.e., the actor space contains both active and passive actors).

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 6 shows the execution time of the application from 5 to 1.000 actors and the best performances are obtained with the passive configuration when the number of actors increases.

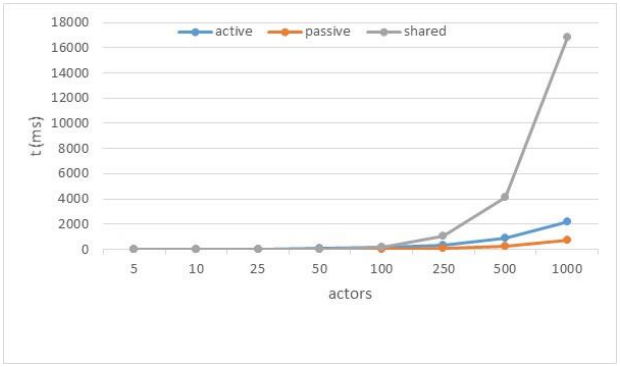

Fig. 6. Point-to-point example performance.

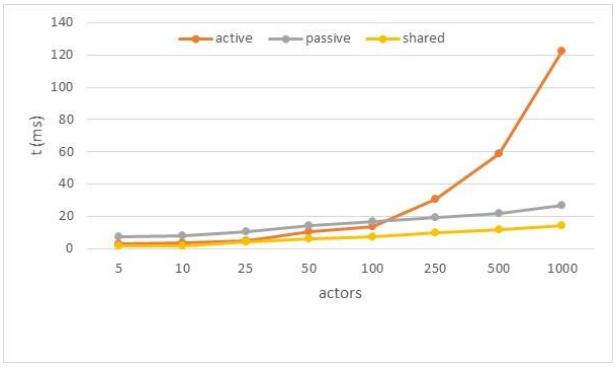

Fig. 7. Broadcasting example performance.

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. 7 shows the execution time of the application from 5 to 1.000 actors and the best

performances are obtained with the shared configuration.

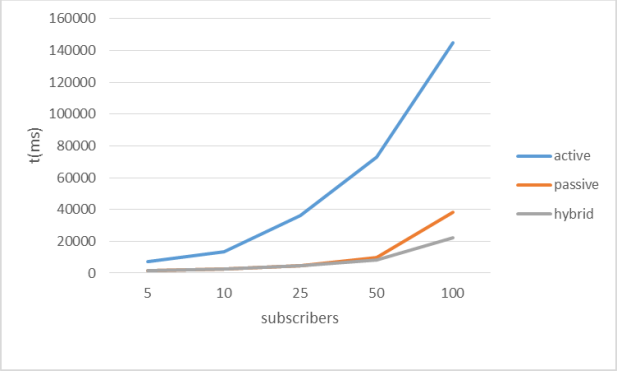

Fig. 8. Publish-subscribe example performance.

Finally, the third is a typical publish – subscribe application. In particular, there is a set of subscribers, which register their interest on the messages sent by a set of publishers. Each publisher cyclically sends a message until it reach a predefined number of messages. Each subscriber processes all the messages sent by the publishers and then kills itself. Fig. 8 shows the execution time of the application from 5 subscribers to 100 subscribers and with 1000 publishers that send 1000 messages. The hybrid configuration runs the subscribers as active actors and the publishers as passive actors. As in the other cases, passive actors offer better performances than active actors do. However, if some of the actors perform a lot of work respect to the other, then the use of an active implementation for such actors can increase the performance for a subset of the possible configurations. In particular, the performances of passive and hybrid configurations are similar up to 50 subscribers, and then the best solution is the use of the hybrid configuration.

7 Experimentation

We experimented and are experimenting CoDE in different application domains and, in particular, in the analysis of social networks [16][17] and in the agent-based modelling and simulation (ABMS) [18][19].

The features of the actor model and the flexibility of its implementation make CoDE suitable for building ABMS applications.

In fact, the use of actors may simplify the development of ABMS applications because of the use of direct communication and the possibility to use actors as middle agents. In fact, actors interact only through messages and there is not a shared state among them (e.g., it is not necessary to maintain an additional copy of the environment to guarantee that agents decide their actions with the same information).

Moreover, conflicts among agents (e.g., movement conflicts among agents in a spatial domain) can be solved using additional actors (acting as middle agents) that inform the other agents about the effect of their actions both on the other agents and on the environment. Moreover, agents do not access directly to the code of the other agents, and so the modification of the code of a type of agent should cause lesser modifications in the code of the other types of agent. Finally, the use of actors simplifies the development of agents in domain where they need to coordinate themselves through direct interactions.

The use of CoDE simplify the development of flexible and scalable ABMS applications. In fact, the use of active and passive actors allows the development of applications involving large number of actors, and the availability of different schedulers and the possibility of their specialization allow a correct and efficient scheduling of the agents in application domains that require different scheduling algorithms [20]. Moreover, the efficient implementation of broadcasting and multicast allow the reduction of the overhead given to the need that agents must often diffuse the information about their state to the other agents of the application (e.g., their location in a spatial domain).

In particular, we are using CoDE for the simulation of some of the most known spatial models: the game of life [21], prey–predator [22], boids [23] and crowd evacuation [24].

The definition of the previous four spatial models is very simple because each agent needs only to get information about its surround (i.e., about a subset of the other agents) and then to use such information for deciding its actions.

Therefore, the simulation algorithm is also very simple if the agents have direct access to the information about the world. It might not happen in an actor-based implementation where the agents can share information only by exchanging messages.

Such agents can be implemented taking advantage of the passive actor implementations provided by the CoDE software that maintains messages in a shared queue, but it is necessary to develop a specific scheduler. The scheduler executes repeatedly all the individuals and after each execution step broadcasts them a “cycle” message.

The agents are modeled by actor behaviors that provides two cases. The first case processes the

messages informing it about the state of the other agents. The second case processes the "clock" messages coming from the scheduler that inform the agent that it owns all the information for deciding the next actions and, of course, provides the code that performs the agent actions.

Moreover, we are using CoDE for simulating and analyzing social networks. The definition of a model for studying social networks is based on an agent that can interact with the agents representing its friends and that can perform actions either in response to messages from other agents or on its own initiative.

Such an agent can be modeled by an actor behavior that provides two cases. The first case processes the messages coming from other agents and executes the actions in their response. The second case is fired by a timeout message, decides the actions that the agent must perform on its own initiative, and then executes them.

Of course, the code of this agent can be very simple, if the decision about both reactive and proactive actions to perform is only based on non-deterministic rules. However, it may become very complex if the agent uses a trust model and/or a user model (up to now, we simulated the propagation of friendship by using both simple non-deterministic rules and trust models).

From an implementation point of view, a massive number of agents are necessary to model a real social network; however, only a part of them are simultaneously active and their actions do not need a synchronization. Therefore, it is necessary a scheduler that can manage a massive number of agents, but that try to optimize the execution by scheduling only the active agents.

The solution we implemented derives from the virtual memory techniques used by operating systems: the scheduler associates with each agent a value that indicates the number of its last inactive cycles and fixes a maximum number of inactive cycles for which an inactive agent can be maintained in the scheduler. When an agent reaches such a number of inactivity cycles, then it is moved in a persistent store and it is reloaded in the scheduler when it receives a new message from another agent.

Of course, the number of active agents can vary over the simulation, but the quality of the simulation can be guarantee if the number of the agents maintained by the scheduler remain in a range that depends on the available computational resources.

The solution adopted to limit to the number of active actors and to guarantee good performances is the use of a variable maximum number of inactive cycles. In fact, this number is high when the number of active agents is low (i.e., the scheduler does not spend time for storing agents in the persistence storage and reloading them) and becomes more and more low with the increasing of the number of active agents.

Two important features that an ABMS framework should provide are the availability of graphical tools for the visualization of the evolution of simulations and the possibility of analyzing the data obtained from simulations.

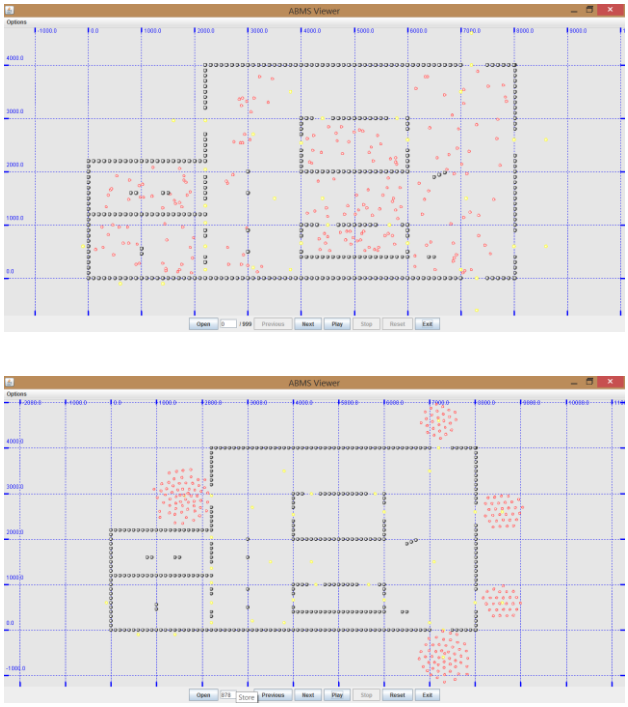

Fig. 9. Initial and final view of the simulation of a crowd evacuation.

CoDE does not provide any specific tool for ABMS, but provides a logging service that allows to record in textual or binary (i.e., Java objects) forms the relevant actions of an actor (i.e., its initialization, reception, sending and processing of messages, creation of actors, change of behavior, and its shutdown).

Therefore, we developed two graphical tools, for visualizing the evolution of simulations based on continuous and discrete representations of a 2D space, and another tool able to extract statistical information from the data obtained from the simulations. Of course, all the three tools get all the information they need from the records coming from the logging service. Fig. 9 shows two views of the GUI that supports 2D spatial simulations. In particular, it presents the initial and final views of the evacuation of a large number of pedestrians from a building.

8 Conclusion

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 [25] and ASIDE [26] from which it derives the concise actor model, and takes advantages of some implementation solutions used in JADE [27].

CoDE shares with Kilim [6], Scala [7] and Jetlang [9] the possibility to build applications that scale 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. In particular, they have the goal of: i) providing a passive threading solution that fully takes advantage of the features of multi-core processors, ii) supporting the creation of distributed computation infrastructures [28], and iii) enhancing the definition of the content exchanged by actors with semantic Web technologies [29][30].

Future research activities will be oriented to the extension of the functionalities provided by the software framework. In particular, they will be dedicated to the provision of a trust management infrastructure to support the interaction between actor spaces of different organizations [31] and to development of an "intelligent protocols" library for making easy the use of CoDE for multi-agent application.

Current experimentation of the software framework is performed in the field of the modeling and simulation of social networks [32], but in the next future will be extended to the collaborative work services [33] and to the agent-based systems for the management of information in peer-to-peer [34] and pervasive environments [35].

References:

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

[2] M. Philippsen, A survey of concurrent object-oriented languages, *Concurrency: Practice and Experience*, Vol. 12, No. 10, 2000, pp. 917-980.

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

[4] R.K. Karmani, A. Shali and G.A. Agha, Actor frameworks for the JVM platform: a comparative analysis, in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, Calgary, Alberta, Canada, 2009, pp. 11-20.

[5] C. Varela, and G.A. Agha, Programming dynamically reconfigurable open systems with SALSA, *SIGPLAN Notices*, Vol. 36, No 12, 2001, pp. 20-34.

[6] S. Srinivasan, and A. Mycroft, Kilim: Isolation-typed actors for Java, in *Proceedings of the ECOOP 2008 – Object-Oriented Programming Conference*, Berlin, Germany, Springer, 2008, pp. 104-128.

[7] 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.

[8] Typesafe, *Akka software*, 2014. Available from: http://akka.io.

[9] M. Rettig, *Jetlang software*, 2014. Available from: http://code.google.com/p/jetlang/.

[10] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt and W. De Meuter, Ambient-oriented programming in ambienttalk, in *Proceedings of the ECOOP 2006 – Object-Oriented Programming Conference*, Berlin, Germany, Springer, 2006, pp. 230-254.

[11] M.S. Miller, E.D. Tribble, and J. Shapiro, Concurrency among strangers, in *Trustworthy Global Computing*, Berlin, Germany, Springer, 2005, pp. 195-229.

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

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

[14] Apache Software Foundation, *Apache Mina Framework*, 2014. Available from: http://mina.apache.org.

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

[16] F. Bergenti, E. Franchi and A. Poggi, Agent-based interpretations of classic network models. *Computational and Mathematical Organization Theory*, Vol. 19, No. 2, 2013, pp. 105-127.

[17] E. Franchi, A. Poggi and M. Tomaiuolo, Open social networking for online collaboration. *International Journal of e-Collaboration*, Vol. 9, No. 3, 2013, pp. 50-68.

[18] K. Altenburg, j. Schlecht and K.E. Nygard, An agent-based simulation for modeling intelligent munitions, in *Proceedings of the 2nd WSEAS International Conference on Simulation, Modeling and Optimization*, Skiathos, Greece, 2002.

[19] A. Poggi, Replaceable Implementations for agent-based Simulation, *SCS M&S Magazine*, 2014.

[20] P. Mathieu and Y Secq, Environment Updating and Agent Scheduling Policies in Agent-based Simulators, in *Proceeding of the 4th International Conference on Agents and Artificial Intelligence (ICAART)*, Algarve, Portugal, 2012, pp. 170-175.

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

[22] M. Mimura and J.D. Murray, On a diffusive prey-predator model which exhibits patchiness, *Journal of Theoretical Biology*, Vol. 75, No. 3, 1978, pp. 249-262.

[23] C.W. Reynolds, Flocks, herds and schools: A distributed behavioral model, *ACM SIGGRAPH Computer Graphics*, Vol. 21, No. 4, 1987, pp. 25-34.

[24] M.H. Zaharia, F. Leon, F. C. Pal, and G. Pagu, Agent-based simulation of crowd evacuation behavior, *in Proceedings of the 11th WSEAS international conference on Automatic control, modelling and simulation (ACMOS'09)*, Istanbul, Turkey, 2009, pp. 529-533.

[25] A. Poggi, HDS: a Software Framework for the Realization of Pervasive Applications, *WSEAS Transactions on Computers*, Vol. 10, No. 9, 2010, pp. 1149-1159.

[26] A. Poggi, ASiDE - A Software Framework for Complex and Distributed Systems, in *Proceeding of the 16th WSEAS International Conference on Computers*, Kos, Greece, 2012, pp. 353-358.

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

[28] A. Negri, A. Poggi, M. Tomaiuolo, P. Turci. Dynamic Grid Tasks Composition and Distribution through Agents. *Concurrency and Computation: Practice and Experience*, Vol. 18, No. 8, 2006, pp. 875-885.

[29] M. Tomaiuolo, P. Turci, F. Bergenti, and A. Poggi, An ontology support for semantic aware agents, in Agent-Oriented Information Systems III, LNCS, Vol. 3529, Berlin, Germany, Springer-Verlag, 2006, pp. 140-153.

[30] A. Poggi, Developing ontology based applications with O3L, *WSEAS Transactions on Computers*, Vol. 8 No. 8, 2009, pp. 1286-1295.

[31] 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, Berlin, Germany, Springer, 2005, pp. 162-179.

[32] F. Bergenti, E. Franchi and A. Poggi, Selected models for agent-based simulation of social networks, in *Proceeding of the 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.

[33] F. Bergenti, A., Poggi and M. Somacher, A collaborative platform for fixed and mobile networks, *Communications of the ACM*, Vol. 45, No. 11, 2002, pp. 39-44.

[34] A. Poggi and M. Tomaiuolo, Integrating Peer-to-Peer and Multi-agent Technologies for the Realization of Content Sharing Applications, in *Information Retrieval and Mining in Distributed Environments*, SCS, Vol. 324, Springer, Berlin, Germany, 2011, pp. 93-107.

[35] F. Bergenti and A. Poggi, Ubiquitous Information Agents, *International Journal on Cooperative Information Systems*, Vol. 11, no. 3-4, 2002, pp. 231-244.