Concurrent and Distributed Applications with ActoDeS

Federico Bergenti[1], Eleonora Iotti[2], Agostino Poggi[2] and Michele Tomaiuolo[2]

[1]*DMI, University of Parma, 43124 Parma Italy*
[2]*DII, University of Parma, 43124 Parma Italy*

Abstract. ActoDeS is a software framework for the development of large concurrent and distributed systems. This software framework takes advantage of the actor model and of an its implementation 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. In particular, the paper introduces the software framework and presents the results of its experimentation.

1 Introduction

Concurrent and distributed programming tools and frameworks 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 [1, 2].

Message passing is the most attractive solution because it is a concurrent model that is not based on the sharing of data and so its techniques can be used in distributed computation too.

One of the well-known theoretical and practical models of message passing is the actor model [3]. Using such a model, programs become collections of independent active objects (actors) that exchange messages and have no mutable shared state. 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 ActoDeS (Actor Development System), that has the suitable features for both simplifying the development of large and distributed complex systems and guarantying scalable and efficient applications. The next section presents related work. Section 3 introduces the software framework and provides details about its implementation. Sections 4 shows how to write the code of an application and how to configure it through a simple application. Section 5 presents the experimentation of the software framework. Finally, section 6 concludes the paper by discussing its main features and the directions for future work.

2 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 [4]. The rest of the section presents some of the most interesting works.

Salsa [5] 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 it limits the scalability. Moreover, message-passing performance suffers from the overhead of reflective method calls.

Kilim [6] is a framework used to create robust and massively concurrent actor systems in Java. It takes advantage of code annotations and of a bytecode 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 [7] is an object-oriented and functional programming language that provides an implementation

[a] Corresponding author: Agostino.poggi@unipr.it

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 [8] 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 [9] 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 [10] is a distributed object-oriented programming language that is implemented on an actor-based and event driven concurrency model, which makes it highly suitable for composing service objects across a mobile network. It provides an actor implementation based on communicating event loops [8]. 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.

3 ActoDES

ActoDeS is an actor based software framework that has the goal of both simplifying the development of concurrent and distributed complex systems and guarantying an efficient execution of applications.

ActoDeS is implemented by using the Java language and takes advantage of preexistent Java software libraries and solutions for supporting concurrency and distribution. ActoDeS 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 ActoDeS middleware infrastructures to support the development of standalone and distributed applications.

3.1. Actors

In ActoDeS an application 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 [3]. Moreover, it can create new actors, update its local state, change its behavior and kill itself.

Communication between actors is buffered: incoming messages are stored in a mailbox until the actor is ready to process them; moreover, an actor can set a timeout for waiting for a new message and then can execute some actions if the timeout fires. Each actor has a system-wide unique identifier called reference that allows it to be reached in a location transparent way. An actor can send messages only to the actors of which it knows the reference, that is, the actors it created and of which it received the references 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 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 behavior 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.

ActoDeS 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: active actors, i.e., actors that have their own thread of execution, and passive actors, i.e., actors that share a single thread of execution. In this last case, the scheduler has the duty of guaranteeing a fair execution of all the actors.

The implementation of an actor is based on four main components: a reference, a mailbox, a behavior and a state. Figure 1 shows a graphical representation of the architecture of an actor.

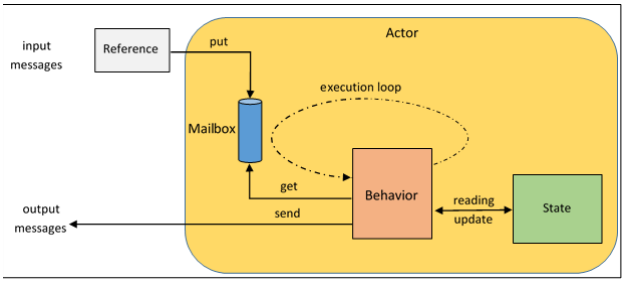

Figure 1. Actor architecture.

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 mailbox has the duty of maintaining the messages sent to its actor until it processes them. In particular, 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 the actor moves to a behavior that is able to process them. If all the behaviors of the actor cannot process such messages, then 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 ActoDeS, a behavior had the main duty of processing the incoming messages and it perform it through a set of "case" methods that have the goal of processing the messages that match a specific 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 task 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. ActoDeS 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 actor libraries implemented in Java, ActoDeS 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

3.2 Actor Spaces

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 advantage 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). Figure 2 shows a graphical representation of the architecture of an ActoDeS distributed application.

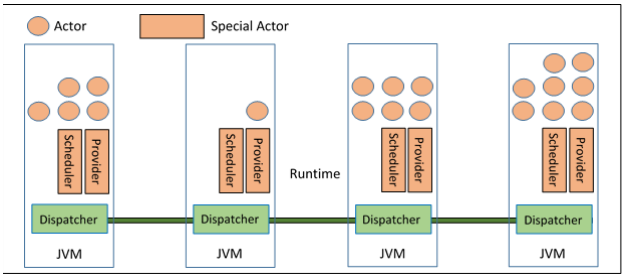

Figure 2. Distributed ActoDeS application architecture.

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 uses them delivering messages to remote actors.

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, ActoDeS

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 provide 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 copies of the 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 supporting both a declarative and procedural configuration of the actor spaces of the application. In fact, such a configuration can be defined as a properties file or as a piece of code that calls the API provided by the configurator.

3.3 Application configuration

The quality of the execution of an ActoDeS 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 ActoDeS 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, in 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 is a key parameter for the quality of applications. In ActoDeS 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. In this case, such actors have a passive implementation and are called shared actors.

Finally, in some kinds of application there is a massive number of actors, but few of them are simultaneously executed. In this case both the active and passive implementations provide a bad scalability because of the cost of maintaining the threads or for the cost of cycling on a large number of inactive actors. An efficient implementation, derived from the passive one, tries to reduce the overhead of cycling on inactive actors. In particular, every actor has a counter that maintains the current number of consecutive inactivity cycles and the scheduler remove such an actor from the scheduling list when its counter reaches a specific value. When it happens, the actor is saved in maintained in the memory of the JVM (temporary-saved actor) or saved in a persistent storage (persistent-saved actor). A saved actor is reloaded in the scheduling list when another actor sends it a message.

3.4 Scalability and performance

The scalability and performance 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 i7-4712HQ - 2.30GHz, 16 GB RAM, Windows 10 - 64bit, Java 8, 8 GB heap size. These examples involve 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 temporary-saved, (i.e., the actor space contains passive actors and the inactive ones may be save in the memory).

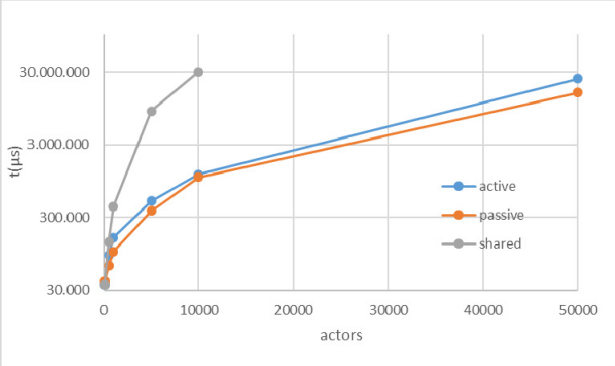

Figure 3. Messaging example performance.

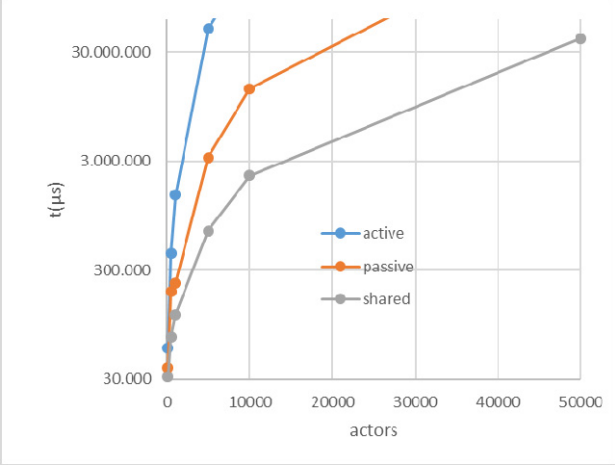

Figure 4. Broadcasting example performance.

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. Figure 3 shows the execution time of the application from 100 to 50.000 actors using the active, passive and shared configurations. The best performances are obtained with the passive configuration 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. Figure 4 shows the execution time of the application from 100 to 50.000 actors using the active, passive and shared configurations. The best performances are obtained with the shared configuration.

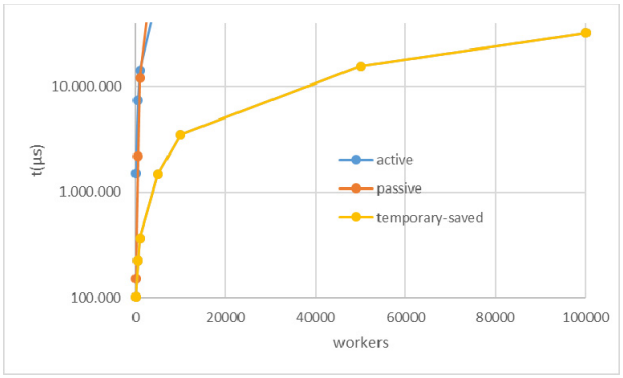

Figure 5. Master-workers example performance.

Finally, the third application can be viewed as a master - worker application. In particular, the master needs to perform a number of tasks equals to the number of workers and assigns them sequentially to one of the workers that it selects randomly. Every task consists of 1000 request - reply interactions between the master and the selected worker. Figure 5 shows the execution time of the application from 100 to 100.000 workers using the active, passive and temporary-saved actors. For high numbers of actors only the use of temporary actors provides reasonable performances

4 An Example of Application

One of the most used examples for highlight the problem due to the concurrent access to a shared resource is the bounded buffer. Its implementation with ActoDeS 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 bounded buffer can be defined by the “EmptyBuffer”, “PartialBuffer” and “FullBuffer” behaviors that defines the three relevant state of the actor. Every behavior of the buffer has a set of cases for processing the messages coming from the producers and consumers and for killing its actor when it receiver a message that concludes the application.

For example, Figure 6 shows the two “initialize” methods that define the code of the “EmptyBuffer” behavior. The first method initializes the buffer after the creation of the actor and calls the second method. The second method is directly call when the actor moves to such a behavior, defines the case for processing the request from a producer, inherits the case for killing its actor and binds the two cases to the corresponding message patterns.

The application is started by a “main” method that: defines the initialization values, the type of scheduler and the actor that the scheduler must create. This actor creates the actors that represent; the buffer, the consumers and the producers, then, at the end of a predefined “lifetime” asks the other actors to kill themselves, and, finally,

terminates the application. Figure 7 shows the code of the "main" method that starts the application.

```
private static final MessagePattern PUTPATTERN =
    new MessagePattern(new MessagePatternField(
      MessageField.CONTENT, new IsInstance(Put.class)));

public void initialize(final Binder b, final Object[] v) {
 setState(new BufferState((Integer) v[0]));
 initialize(b);
}

public void initialize(final Binder b) {
 this.state = (BufferState) getState();

 Case c = (m) -> {
   this.state.add(((Put) m.getContent()).getElement());
   send(m, Done.DONE);

   if (this.state.size() == this.state.getCapacity()) {
    return new FullBuffer();
   }
   else {
    return new PartialBuffer();
   }
 }

 b.bind(PUTPATTERN, c);
 b.bind(KILLPATTERN, this.killCase);
}
```

Figure 6. The "EmptyBuffer" behavior.

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

  Configuration c = Controller.INSTANCE.getConfiguration();

  c.setScheduler(ThreadScheduler.class.getName());

  c.addActors(
     1, Initiator.class.getName(),
     lifetime, size, producers, consumers);

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

Figure 7. The application "main" method.

5 Experimentation

We experimented and are experimenting ActoDeS in different application domains and, in particular, in the agent-based modelling and simulation (ABMS) [16], in the analysis of social networks [17, 18], in modelling e-business [19] and collaborative work services [20] and agent-based systems for the management of information in peer-to-peer [21], mobile [22] and pervasive environments [23, 24].

In particular, the analysis of large social networks is a suitable domain for testing the distribution of an application on different computing nodes. Our work on the modeling and simulation of social networks started some years ago when we used agent-based techniques for generating and analyzing different types of social network of limited size [17, 25]. Now we can take advantage of the ActoDeS software framework for coping with very large social networks. Therefore, in a ActoDeS system, actors represent the individuals of the social network and maintain their information. Moreover, such actors can exhibit different behaviors, allowing both to cooperate in the analysis of the social network and to simulate the behavior of the represented individuals by performing the actions that they can perform in the social network. Of course, some additional actors are necessary, in particular, for generating the social network and for driving its measurements.

The architecture we defined for such a kind of application is a distributed architecture based on a variable number of actor spaces. Each actor space maintains a set of actors that are managed by a temporary or persistent scheduler. Moreover, the service provider takes advantage of a naming service.

An important factor that simplifies the parallel construction of a social network is the availability of a universal unique identifier for each individual of the social network. Such an identifier permits to avoid the creation of actors representing the same individual in different actor spaces, thanks to the use of the naming service. In fact, the naming service allows to:

- maintain the binding between the references of the active actors with the identifiers of the corresponding individual;
- use the individual identifier to find an actor in the persistent storage;
- cooperate with the naming services of the actor spaces to decide if a new actor must be created.

Moreover, such a kind of application can take advantage of the fact that ActoDeS supports the migration of actors. It allows to define an initial distribution of the actors without any information about the structure of the network and then to move actors during the execution of the analysis and simulation tasks to reduce the overhead of the communication between the different computational nodes.

We started the experimentation of such a system by modelling some social networks with a number of individuals that vary from some thousands to some millions of individuals. We built such models by using the data maintained in the "Stanford Large Network Dataset Collection" [26] and up to now, we are using them for performing some simple measures (i.e., diameter, clustering coefficient and centrality). The first tests we did compare the performances of the system with a deployment on a different number of computing nodes (from one to four). The results of the tests showed that a single actor space can manage social networks with some millions of individuals, but the use of additional actor spaces on more computing nodes gives an important improvement in the performances. In fact, the advantages on performance of the partitioning of the model of large social networks on some computing nodes are relevant for both the creation and measurement phases, because it is necessary to move a smaller number of actors from the scheduler to the temporary or persistent storage and vice versa.

6 Conclusions

ActoDeS is a software framework that 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.

ActoDeS is implemented by using the Java language and is an evolution of CoDE [27] that simplifies the definition of actor behaviors and provides more scalable and performant implementations. Moreover, it takes advantages of some implementation solutions used in JADE [28] for the definition of some internal components.

ActoDeS 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, ActoDeS 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 and future research activities are and will be dedicated to: i) extend the experimentation of the software framework, ii) provide a set of actor and scheduler implementations for ABMS and multi-agent applications, and iii) extend the functionalities provided by the software framework. In particular, future activities will be dedicated to the provision of a set of security and trust services and a set of coordination protocols to support the interaction between actor spaces of different organizations [29, 30] and the development of an ontology model and language for definition of the content exchanged by actors [31, 32].

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, pp. 917-980 (2000)
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 Proc. of the 7th International Conference on Principles and Practice of Programming in Java, Calgary, Alberta, Canada, pp. 11-20 (2009)
5. C. Varela and G. A. Agha, Programming dynamically reconfigurable open systems with SALSA, SIGPLAN Notices, vol. 36, no. 12, pp. 20-34 (2001)
6. S. Srinivasan, and A. Mycroft, Kilim: Isolation-typed actors for Java, in Proc. of the ECOOP 2008 – Object-Oriented Programming Conference, Berlin, Germany, Springer, pp. 104-128 (2008)
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 (2016), Akka software, Available from: http://akka.io.
9. M. Rettig (2016), Jetlang software. Available from: https://github.com/jetlang/.
10. J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt and W. De Meuter, Ambient-oriented programming in ambienttalk, in Proc. of the ECOOP 2006 – Object-Oriented Programming Conference, Berlin, Germany, Springer, pp. 230-254 (2006)
11. M. S. Miller, E. D. Tribble and J. Shapiro, Concurrency among strangers, in Trustworthy Global Computing, Berlin, Germany, Springer, pp. 195-229 (2005)
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 (2016), Apache Mina Framework, Available from: http://mina.apache.org
15. P. Hintjens, ZeroMQ: Messaging for Many Applications, Sebastopol, CA, USA, O'Reilly, 2013.
16. A. Poggi, Agent based modeling and simulation with ActoMoS, Proc. of 16th Workshop on From Object to Agents (WOA 2015), Naples; Italy, pp. 91-96 (2015)
17. F. Bergenti, E. Franchi and A. Poggi, Agent-based interpretations of classic network models, Computational and Mathematical Organization Theory, vol. 19, no. 2, pp. 105-127 (2013)
18. E. Franchi, A. Poggi and M. Tomaiuolo, Open social networking for online collaboration, International Journal of e-Collaboration, vol. 9, no. 3, pp. 50-68 (2013)
19. A. Negri, A. Poggi, M. Tomaiuolo, and P. Turci, Agents for e-Business Applications, in 5th Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems, Hakodate, Japan: ACM, pp. 907-914 (2006)
20. F. Bergenti, and A. Poggi. An agent-based approach to manage negotiation protocols in flexible CSCW systems, in Proc. of the fourth international conference on Autonomous agents, ACM, pp. 267-268. ACM (2000)
21. A. Poggi, and M. Tomaiuolo, A DHT-based multi-agent system for semantic information sharing, in New Challenges in Distributed Information Filtering and Retrieval, Berlin, Germany: Springer, pp. 197-213 (2013)

22. F. Bergenti, and A. Poggi, LEAP: A FIPA platform for handheld and mobile devices, Intelligent agents VIII. Berlin, Germany, Springer, pp. 436-446 (2001)
23. F. Bergenti, and A. Poggi, Ubiquitous Information Agents, International Journal on Cooperative Information Systems, Vol. 11, No. 3-4, pp. 231-244, (2002)
24. A. Poggi, HDS: a Software Framework for the Realization of Pervasive Applications, WSEAS Transactions on Computers, vol. 10, no. 9, pp. 1149-1159 (2010)
25. F. Bergenti, E. Franchi, E, and A. Poggi, Selected models for agent-based simulation of social networks, in Proc. 3rd Symposium on Social Networks and Multiagent Systems (SNAMAS'11), York, UK, pp. 27-32 (2011)
26. Stanford University (2016), SNAP: Stanford Large Network Dataset Collection, Available from: http://snap.stanford.edu/data/index.html.
27. F. Bergenti, A. Poggi, and M. Tomaiuolo, An Actor Based Software Framework for Scalable Applications, in Internet and Distributed Computing Systems, Berlin, Germany: Springer, pp. 26-35 (2014)
28. A. Poggi, M. Tomaiuolo, and P. Turci, Extending JADE for agent grid applications, in Enabling Technologies: Infrastructure for Collaborative Enterprises, Modena, Italy. IEEE Computer Society, pp. 352-357 (2004)
29. 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, Springer, Berlin, Germany, pp. 162-179 (2005)
30. F. Bergenti, and A. Poggi, An agent-based approach to manage negotiation protocols in flexible CSCW systems, in Proc. 4th International Conference on Autonomous Agents, Barcelona, Spain, ACM, pp. 267-268 (2000)
31. 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, Springer, Berlin, Germany, pp. 140-153 (2006)
32. A. Poggi, Developing ontology based applications with O3L, WSEAS Trans. on Computers, vol, 8, no. 8, pp. 1286-1295 (2009)