An Actor based Software Framework for Scalable Applications

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

[1] DMI, University of Parma, Parma, Italy
federico.bergenti@unipr.it
[2] DII, University of Parma, Parma, Italy
{agostino.poggi,michele.tomaiuolo}@unipr.it

Abstract. The development of scalable and efficient applications requires the use of appropriate models and software infrastructures. This paper presents a software framework that enables the development of scalable and efficient actor-based applications. Each application can be configured with different implementations of the components that drive the execution of its actors. In particular, the paper describes the experimentation of such a software framework for the development of agent-based modelling and simulation applications that involve a massive number of individuals.

Keywords: Actor model, software framework, concurrent systems, distributed systems, scalable applications, Java.

1 Introduction

Concurrency and parallelism are becoming the most important ingredients for developing applications running on nowadays computing platforms. However, one of the main obstacles that may prevent the efficient usage of such platforms is the fact that traditional (sequential) software is not the most appropriate means for their programming.

Message passing models and technologies seem be one of the most attractive solution for the programming of current computing platforms because they are defined on 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 [1]. Using such a model, programs become collections of independent active objects (actors) that do not have shared state and communicate only through the exchange of messages. Actors can help developers to avoid issues such as deadlock, live-lock and starvation, which are common problems for shared memory 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 facilitates the development of programs, or event-based programming, which is far more practical to develop large and efficient concurrent systems, but is also 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 section describes the software framework. Section 3 presents its initial experimentation in the agent-based modelling and simulation of Web and social networks. Section 4 introduces related work. Finally, section 5 concludes the paper by discussing the main features of the software framework and the directions for future work.

2 CoDE

CoDE (Concurrent Development Environment), is an actor based software framework that has the goal of both simplifying the development of large and distributed complex systems and guarantying an efficient execution of applications.

CoDE is 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 a runtime and an application layer. The runtime layer provides the software components that implement the CoDE middleware infrastructures to support the development of standalone and distributed applications. 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. In particular, the development of an application usually consists in the development of the actor behaviors implementing the functionalities of the application and in the definition of the configuration (i.e., the selection of the most appropriate implementations) of the components the drive the execution of such behaviors.

2.1 Actors

In CoDE an application is based on a set of interacting actors that perform tasks concurrently. Actors are autonomous concurrent objects, which interact with each other by exchanging asynchronous messages. Moreover, they can create new actors, update their local state, change their behavior and kill themselves. Finally, they can set a timeout for waiting for a new message and receive a timeout message if it fires.

An actor can be viewed as a logical thread that implements an event loop [2,3]. This event loop perpetually processes events that represent: the reception of messages and the behavior exchanges. CoDE provides two types of implementation of an actor, that allow it either to have its own thread (from here named active actor), or to share a single thread with the other actors of the actor space (from here named passive actor). Moreover, the implementation of an actor takes advantage of other four main components: a reference, a mailer, a behavior, and a state. Fig. 1 shows a graphical representation of the architecture of an actor.

A reference acts as address and proxy of an actor. 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 is sent by such an actor or whose content enclosed its reference. In fact, a message is an object that contains a set of fields maintaining the typical header information (e.g., the sender and the receiver references) and the message content.

A mailer provides a mailbox, maintaining 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 it can maintain. However, it is clear that the (permanent) deposit of a large number of messages in the mailboxes of the actors may reduce the performances of applications and, in some circumstances, cause their failure.

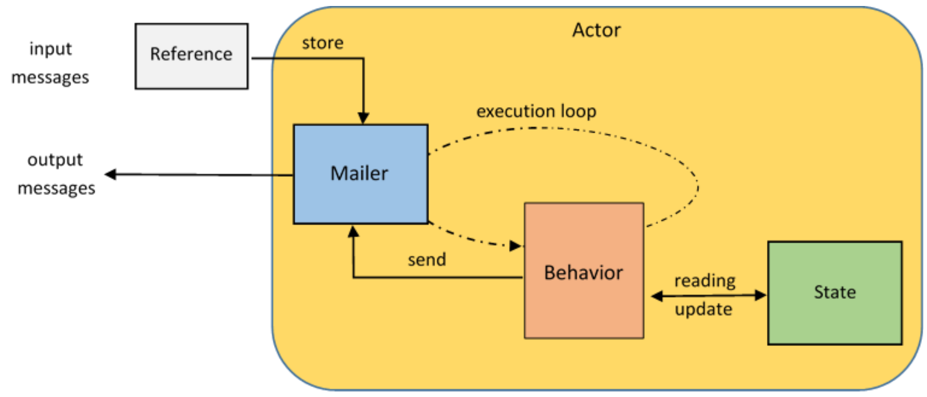

Fig. 1. Actor architecture

The original actor model associates a behavior with the task of message processing. In CoDE, a behavior can perform two kinds of tasks: its initialization and the processing of messages. 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.

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 will allow the definition of sophisticated filters on the values of all the message fields and in particular on the content of the message.

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 thanks 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.

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 the large part of the actors libraries implemented in Java do, 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 all the predefined message content objects implementing as immutable) and delegates the appropriate use of mutable object to them.

2.2 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. To do it, an actor space takes advantage of two main runtime components (i.e., the registry and the dispatcher) and two special actors (i.e., the scheduler and the service provider).

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. CoDE allows the use of different implementations of such a communication component. In particular, the current implementation of the software framework supports the communication among the actor spaces using ActiveMQ [4], Java RMI [5], MINA [6] and ZeroMQ [7].

The registry supports the creation of actors and the reception of the messages coming from remote actors. In fact, it has the duties of creating new references and of providing the reference of the destination actor to the dispatcher. which manages a message coming from a remote actor. In fact, as introduced in the 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 two actor spaces involved in the communication.

The scheduler is a special actor that manages the execution of the actors of an actor space. CoDE provides different implementations of such a special actor, and the use of one or another implementation represents another factor that have big influence on the attributes of the execution of an application. Of course, the duties of a scheduler depend on the type of actor implementation and, in particular, on the type of threading solutions associated with the actors of the actor space. 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 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 offers services for supporting the broadcast of messages, the exchange of messages through the “publish and subscribe” pattern, the binding between names and references, the mobility, the interaction with users through emails and the creation of new actors (useful for creating actors in empty actor spaces).

2.3 Actor and Scheduler Implementations

The quality of the execution of a CoDE application mainly depends on the implementation of the actors and schedulers of its actor spaces. However, a combination of such implementations, that maximizes the quality of execution of an application, could be a bad combination for another 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 combinations.

As introduced above, from an implementation point of view, actors are divided in active and passive actors. The use of active actors delegates their scheduling to the JVM, with the advantage of guaranteeing them 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 is suitable only for applications whose actor spaces have 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 and schedulers. In this case, the scheduler implements a simple not preemptive round-robin scheduling algorithm for the execution of various actors. On the other hand, each actor provides a method that allows it to perform a piece (from here called step) of the work (i.e., the processing of some messages) in each scheduling cycle. This last solution is suitable when it is possible to distribute the tasks in equal parts among the actors. If it does not happen, heavy actors should have a priority on the access to the computational resources of the actor space. In this situation, a good solution is to provide a hybrid scheduler able to manage together active and passive actors and delegating to the active actors the heavy tasks.

However, guaranteeing a good quality of execution in different application scenarios often requires the satisfaction of some constraints, that cannot be achieved by the same actor and scheduler implementations. For example, applications where actors act as proxy of real users should guarantee a fair access to the computational resources of the actor space, by limiting the number of messages that an actor can process in a single step. Heavy applications should try both to reduce the overhead of the scheduler and to offer an acceptable fair execution of the actors, for example, by extending the processing of a single step to all the messages received before the scheduling of the actor. Applications where actors mainly communicate through the exchange of broadcast messages should try to reduce the overhead of the delivery of such messages. Finally, applications that involve a massive number of actors should try to reduce the overhead of managing the inactive actors.

CoDE provides some actor and scheduler implementations that allow the improvement of the quality of execution for different types of applications, including the ones described in the previous paragraph. A large part of the implementations has few differences among them. The most particular implementations are the ones that cope with the overhead of the delivery of broadcast messages and with the overhead of the management of inactive actors.

For reducing the overhead of the delivery of broadcast messages, an actor implementation (called shared actor) uses a mailbox that transparently extracts the messages from a single queue, shared with all the other actors of the actor space, and a scheduler implementation (called shared scheduler) that has the duty of the management of such a queue. To simplify the management of the queue, the shared actors can only get the messages sent in the previous scheduling cycle and, at the end of each scheduling cycle, the shared scheduler must add an "end cycle" message at the end of the queue and then remove the messages before the previous "end-cycle" that are already processed by the actors.

For reducing the overhead of the management of inactive actors, an actor implementation (called measurable actor) offers a method providing the number of scheduling cycles from which it does not perform actions. Two scheduler implementations (called temporary and persistent schedulers) use such an information for removing actors from their scheduling list. After removing an actor, the temporary scheduler maintains the actor in the JVM memory and the persistent scheduler moves it in a persistent storage. This solution requires two different implementations of the registry component (called temporary and persistent registries) whose duty is to reload an actor, either from the JVM memory or from the persistence storage, when another actor sends a new message to it.

3 Experimentation

We experimented and are experimenting CoDE in different application domains and, in particular, in the agent-based modelling and simulation (i.e., the game of life, the prey-predators pursuit game, the flocking behavior of birds, the movement of crowds in indoor and outdoor environments, and the analysis of social networks) [8].

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 [9,10,11]. Now we can take advantage of the CoDE software framework for coping with very large social networks. Therefore, in a CoDE 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 measurements 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 agent-based modelling and simulation is a distributed architecture based on a variable number of actor spaces (Fig. 2 shows its graphical representation). Each actor space maintains a set of measurable actors that are managed by a persistent scheduler. Moreover, the service provider takes advantage of a naming service.

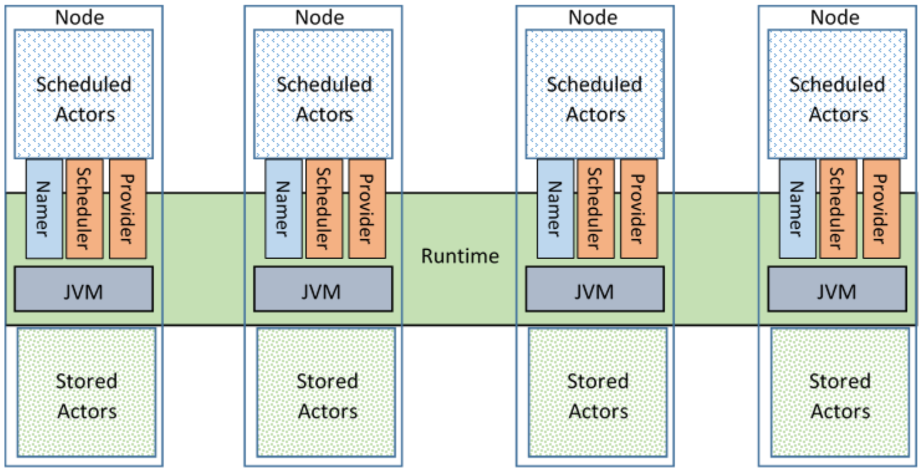

Fig. 2. Massive agent-based modelling and simulation system architecture

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.

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" [12] 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 persistent storage and vice versa. Of course, our experimentation is at the beginning and the results are only of qualitative level. However, we are working hard for enriching the measurement phase with new functionalities and for preparing a set of tests for acquiring a set of accurate measures of the performance of the system in its different configurations.

4 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 [13] 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 [14] is a framework used to create robust and massively concurrent actor systems in Java. It takes advantage of code annotations and 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 [15] 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 [16] 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 [17] 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 [2] 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 [3]. However, each actor is always associated with its own JVM thread and this limits the scalability of applications with respect to the number of actors for JVM.

5 Conclusions

This paper presented an actor-based software framework, called CoDE, that enables the development of scalable and efficient applications by configuring them with different implementations of its components. Moreover, such a software framework is based on a simple actor model that simplifies the development of applications. In fact, the development of application consists in the development of the actor behaviors that implement its functionalities and the definition of a configuration that choose the most suitable implementations for the components that drive the execution of the actors of the application.

CoDE is implemented by using the Java language and is an evolution of HDS [18] and ASIDE [19] from which it derives the concise actor model. CoDE shares with Kilim [14], Scala [15] and Jetlang [17] the possibility to build applications that scale to a massive number of actors, but without the need of introducing new constructs that complicate 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 enables the implementation of complex interactions in a distributed application, because a message contains all the information for its delivery to the destination and then for building and sending a reply. Moreover, a message pattern filters the input messages not only with respect to their content, but also with respect to all the information they contain.

CoDE has been mainly experimented in the agent-based modelling and simulation. Such an experimentation involved the development of systems with different features (number of actors, types of communication, ratio between active and inactive actors, etc.) and demonstrated that different configurations are necessary to obtain the best performance for different types and setup of systems.

Current research activities are dedicated to extend the software framework to offer it as means for the development of multi-agent systems taking advantages of some design and implementation solutions used in JADE [20]. 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: i) providing a passive threading solution that fully take advantage of the features of multi-core processors, ii) enabling the interoperability with Web services and legacy systems [21], and iii) enhancing the definition of the content exchanged by actors with semantic Web technologies [22]. Moreover, future activities will be dedicated to the provision of a trust management infrastructure to support the interaction between actor spaces of different organizations [23], [24]. Experimentation of the software framework will be extended to the development of: i) collaborative work services [25] and ii) agent-based systems for the management of information in pervasive environments [26].

References

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

2. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented programming in ambienttalk. In: ECOOP 2006 – Object-Oriented Programming, pp. 230-254, Springer, Berlin, Germany (2006)
3. Miller, M.S., Tribble, E.D., Shapiro, J.: Concurrency among strangers. In: Trustworthy Global Computing. pp. 195-229, Springer, Berlin, Germany (2005).
4. Snyder, B., Bosnanac, D., Davies, R.: ActiveMQ in action. Manning, Westampton, NJ, USA (2001)
5. Pitt, E., McNiff, K.: Java.rmi: the Remote Method Invocation Guide. Addison-Wesley, Boston, MA, USA (2001)
6. Apache Software Foundation: Apache Mina Framework. http://mina.apache.org
7. Hintjens, P.: ZeroMQ: Messaging for Many Applications. O'Reilly, Sebastopol, CA, USA, (2013)
8. Poggi, A.: CoDE - A Software Framework for Agent-based Simulation. In: 17th WSEAS International Conference on Computers, pp. 50-55, Rhodes, Greece (2013)
9. Bergenti, F., Franchi, E., Poggi, A.: Selected models for agent-based simulation of social networks. In: 3rd Symposium on Social Networks and Multiagent Systems (SNAMAS'11), pp. 27-32, Society for the Study of Artificial Intelligence and the Simulation of Behaviour, York, UK (2011)
10. Franchi, E.: A Domain Specific Language Approach for Agent-Based Social Network Modeling. In: IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp. 607-612. IEEE, (2012)
11. Bergenti, F., Franchi, E., Poggi, A.: Agent-based interpretations of classic network models. Computational and Mathematical Organization Theory 19(2), 105-127 (2013)
12. SNAP: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/index.html
13. Varela, C., Agha, G.A.: Programming dynamically reconfigurable open systems with SALSA. SIGPLAN Notices 36(12), 20-34 (2001)
14. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: ECOOP 2008 – Object-Oriented Programming, pp. 104-128, Springer, Berlin, Germany (2008)
15. Haller, P., Odersky, M.: Scala Actors: unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
16. Typesafe: Akka software Web site. http://akka.io.
17. Rettig, M.: Jetlang software Web site. http://code.google.com/p/jetlang
18. Poggi, A.: HDS: a Software Framework for the Realization of Pervasive Applications. WSEAS Trans. on Computers 10(9), 1149-1159 (2010)
19. Poggi, A.: ASiDE - A Software Framework for Complex and Distributed Systems. In: 16th WSEAS International Conference on Computers, pp. 353-358, Kos, Greece (2012)
20. Poggi, A., Tomaiuolo M., Turci, P.: Extending JADE for agent grid applications. In: 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2004), pp. 352-357, Modena, Italy (2004)
21. Poggi, A., Tomaiuolo M., Turci, P.: An Agent-Based Service Oriented Architecture. In: WOA 2007, pp. 157-165, Genova, Italy (2007)
22. Poggi, A.: Developing ontology based applications with O3L. WSEAS Trans. on Computers, 8(8), 1286-1295 (20099
23. Poggi, A., Tomaiuolo M., Vitaglione, G.: 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, pp. 162-179, Springer, Berlin, Germany (2005)
24. Tomaiuolo, M.: dDelega: Trust Management for Web Services. International Journal of Information Security and Privacy 7(3), 53-67 (2013)
25. Bergenti, F., Poggi, A., Somacher, M.: A collaborative platform for fixed and mobile networks. Communications of the ACM, 45(11), 39-44 (2002)
26. Bergenti, F., Poggi, A.: Ubiquitous Information Agents. International Journal on Cooperative Information Systems 11(3-4), 231-244 (2002)