ActoDeS - A Software Development System for Complex and Distributed Applications

Agostino Poggi
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma

ABSTRACT

The current evolution of computing architectures increases the importance of the use of appropriate distributed and concurrent programming models, languages and software frameworks for the development of systems. The actor model and the derived programming languages and development frameworks seem to offer the suitable features for the development of concurrent and distributed systems. ActoDeS is a software environment that has the goal of simplifying the development of distributed and complex systems by using a customizable actor model implementation. This technical report has the goal of giving an overview of the software environment together with a short introduction about the steps that a developer should follow for developing applications.

Key Words: actor model, concurrent programming, distributed systems, Java.

Document version: May 28, 2020

Correspondence address: Prof. Agostino Poggi
Dipartimento di Ingegneria e Architettura
Università degli Studi di Parma
Parco Area delle Scienze, 181A, I-43100, Parma, Italy

This software environment has been developed thanks to the contribution of the students of the distributed systems course of the Computer Engineering second cycle degree.

Introduction

Distributed and concurrent programming models, languages and software frameworks (Philippsen, 2000; Leopold, 2001) are assuming always more importance because the diffusion of the use of distributed computing architectures. The actor model (Hewitt, 1977; Agha, 1986; Agha et al., 1997) and the derived programming languages and development frameworks (see, for example, Varela & Agha, 2001; Srinivasan & Mycroft, 2008; Haller & Odersky, 2009) seem to offer the suitable features for the development of concurrent and distributed systems. However, the implementations of such a model had to choose between two goals, i.e., either to make simple the writing of the code (by using the thread based programming model) or to allow the development of efficient large systems (by using the event based programming model).

An implementation of the actor model that achieves both the previous two goals may derive from the AmbientTalk language (Dedecker et al., 2006). In fact, it represents an actor as an event loop that perpetually processes events from its event queue by invoking a specific event handler. Moreover, each event handler cannot stop itself for waiting other events. Therefore, the development of the code of event handlers should not entail big difficulties and the possibility of developing efficient large systems should mainly depend on the implementation of the event loop and of the code supporting the exchange of events among the actors. ActoDeS (Actor Development Environment) is an actor based software environment that at the same way of AmbientTalk delegates the reception of messages and the execution of the corresponding handlers to its execution environment. Moreover, it allows the implementation of efficient large systems by supporting the sharing of threads among the actors of the systems (i.e., systems can scale with large numbers of actors).

This technical report presents the ActoDeS software environment and introduces the basic steps and the essential code to developing applications.

ActoDeS Programming Model

In ActoDeS a system is based on a set of interacting actors that perform tasks concurrently. An actor is an autonomous concurrent object, which interacts with other actors by exchanging asynchronous messages. In response to a message, an actor can send messages to other actors or to itself, create new actors, update its local state, change its behavior, kill itself and be ready to process the next message.

Each actor has a system-wide unique identifier called its address. This address allows an actor to be referenced in a location transparent way. An actor can send messages only to the actors of which it knows the address, that is, the actors it created and the actors of which it received their addresses from other actors.

After its creation, an actor can change several times its behavior until it kills itself. Each behavior has the main duty of processing incoming messages through some message handlers. Each message handler can process only the messages that match a specific message pattern represented by an object that can apply a combination of constraints on the value of all the fields of a message. Therefore, if an unexpected message arrives, then the actor mailbox maintains it until a next behavior will be able to process it.

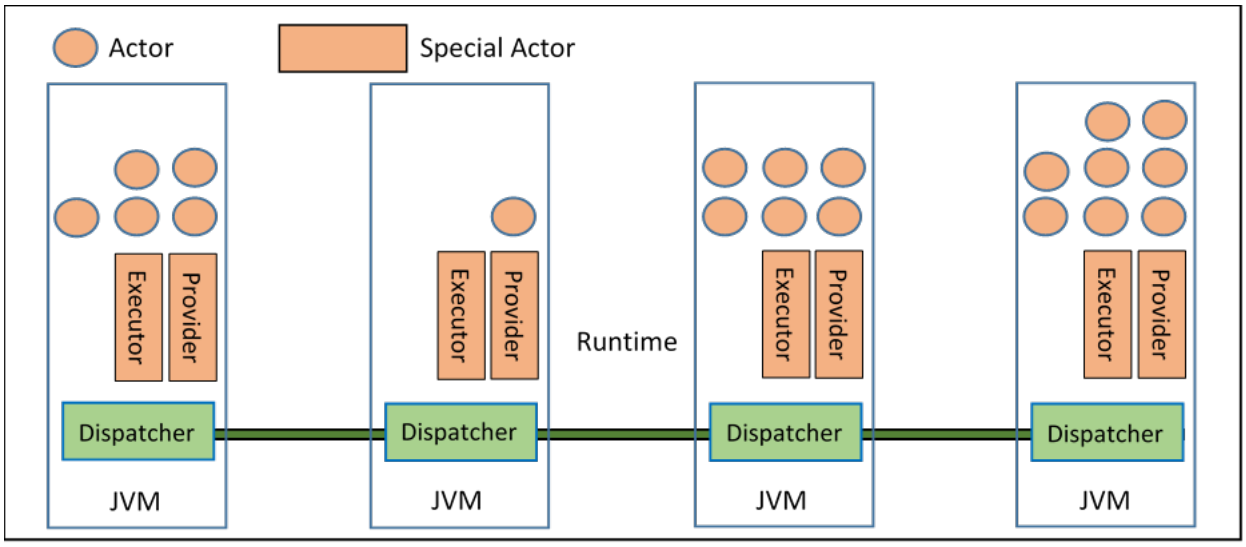

Figure 1. ActoDeS distributed application architecture.

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. An actor space contains a set of actors (application actors) that perform the specific tasks of the current application and two actors (runtime actors) that support the execution of the application actors. These two last actors are called executor and the service provider. The executor 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 1 shows a graphical representation of the architecture of an ActoDeS distributed application.

ActoDeS Implementation

ActoDeS is a software environment implemented by using the Java language and takes advantage of preexistent Java software libraries and solutions for supporting concurrency and distribution. The development of a standalone or distributed application consists in the definition of the code of the behaviors assumed by the different actors of the application and the definition of the configuration of the application. In particular, ActoDeS allows the configuration of an application with different implementations of actors and runtime components for improving some or other its attributes (e.g., performance, reliability, and scalability). Moreover, the deployment of an application on a distributed architecture is simplified because an actor can transparently communicate with remote actors.

Actor

An actor is an instance of the `Actor` class. An actor can be viewed as a logical thread that implements an event loop [11, 21]. This event loop perpetually processes incoming messages. In fact, when an actor receives a message, then it looks for the suitable message handler for the processing of the message and, if it exists, it processes the message. The execution of the message handler is also the means for changing its way of acting. In fact, the actor uses the return value of its message handlers for deciding to remain in the current behavior, to move to a new behavior or to kill itself.

An actor performs its duties taking advantage of four main components: a reference, a mailbox, a behavior and a state. Figure **2** shows a graphical representation of the architecture of an actor.

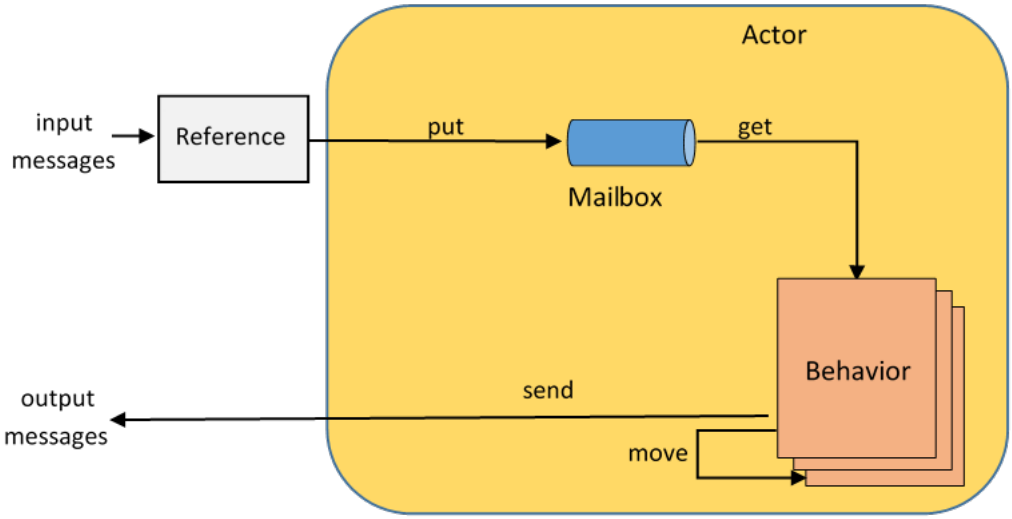

Figure 2. Actor architecture.

Reference

A reference is an instance of the `Reference` class. This class defines a runtime component that acts as proxy of a specific actor with the goal of supporting the sending of messages to the actor it represents. Therefore, an actor needs to have the reference of an actor for sending it a message. In particular, an actor has two main ways for obtaining the references of the other actors of the application:

- By creating new actors (in fact, the creation method returns the reference of the new actor).
- By receiving a message from another actor (in fact, each message contains the reference of the sender and its content can contain the reference of one or more other actors).

Moreover, an actor can bind its reference to a name. In this case, if another actor knows the name of such an actor, then it can get its reference performing a lookup action.

However, an actor has a priori knowledge of four references of the actor space identifying: the service provider, the executor, the local broadcasting service and the global broadcasting service. Usually, actors do not receive and send messages from/to the executor of the actor space, but its reference is useful to support the collaboration among the executors of a distributed application (i.e., the executors collaborate for guarantying a good distribution of workloads among the actor spaces). From an implementation point of view, such a priori knowledge is possible because the code of actor behaviors can access such references through their `PROVIDER`, `EXECUTOR`, `SPACE` and `APP` class variables.

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

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

Mailbox

A mailer maintains the messages sent to its actor until it processes them. As introduced above, a behavior can process a set of specific messages leaving in the mailbox the messages that is not able to process. Such messages remain into the mailbox until a new behavior is able to process them and if there is not such a behavior they remain into the queue for all the life of the actor. A mailbox has

not an explicit limit on the number of messages that can maintain. However, it is clear that the (permanent) deposit of large numbers of messages in the mailboxes of the actors may reduce the performances of applications and cause in some circumstances their failure.

Message, Message Pattern and Message Handler

A message is an instance of the `Message` class. This class defines a set of fields maintaining the typical header information of a message and its content (Table 1 introduces a short description of the fields of a message). Moreover, each message is different from any other one. In fact, messages of the same sender have a different identifier and messages of different senders have a different sender reference.

An actor has not direct access to the local state of the other actors and can share data with them through the exchange of messages. However, if an actor creates other actors, then it can pass some data to them through the creation operation. 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 done by a majority 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 developers to use immutable objects, by implementing all the predefined message content objects as immutable objects.

Field name	**Description**
Identifier	Message identifier
Sender	Sender reference
Receiver	Receiver reference
Content	Message content
Time	Delivery time
Type	Message type: either one-way or two-way
inReplyTo	Identifier of the replied message

Table 1. Message fields.

A message pattern is an instance of the `MessagePattern` class. This class applies a set of constraints to the fields of messages and each constraint is an instance of a class that implements the `Constraint` interface. Table 2 provides a short description of the set of predefined constraints.

This kind of representation of a message pattern allows a very sophisticate filtering of messages. Moreover, the use of the `Matches` constraint allows a specialization of the filtering for all the message fields and in particular for the content of the message. For example, ActoDeS provides an additional pattern, implemented by the `RegexPattern` class. This pattern allows the filtering of the messages by matching the string representation of the value of a field with a specific regular expression. Moreover, the `Behavior` abstract class provides a set of predefined message patterns through some class variables. Table 3 provides a short description of these predefined message patterns.

Developers can add new types of pattern and constraint. Of course, it is necessary to develop the classes representing such patterns and/or constraints, but is also necessary to implement the component that applies the pattern to a specific type of object. This component will be an implementation of the `Matcher` interface.

A message handler implements the `MessageHandler` functional interface and, in particular, its `process` method that has the duty of processing the messages that satisfy a specific message pattern. The implementation of such a method takes advantage of a set of other methods, defined by

`Behavior` class. These methods allow an actor to get its address and the address of its creator, to get and set the state of the behavior, to create other actors, to perform naming operations and to send messages. Table 4 provides a short description of the methods that implement the basic actions of an actor.

Constraint	**Description**
All(constraint, objects)	True if all the elements of objects satisfy the constraint
And(constraints, object)	True if object satisfies all the constraints
Contains(objects1, objects2)	True if objects1 contains all the elements of objects2
IsDifferent(object1, object2)	True if object1 is different from object2
IsEqual(object1, object2)	True if object1 is equal to object2
IsHigher(object1, object2)	True if the object2 is greater than object1
IsInstance(type, object)	True if object is an instance of type
IsLower(object1, object2)	True if object2 is less than object1
IsNull(object)	True if object is null
IsOneOf(objects, object)	True if object is one of the elements of objects
Matches(pattern, object)	True if object matches the pattern
Not(constraint, object)	True if object does not satisfy the constraint
Or(constraints, object)	True if object satisfies at least one of the constraints
Some(constraint, objects)	True if at least one of the elements of objects satisfies the constraint

Table 2. Message pattern constraints.

Message pattern class variable name	**Description**
START	Matches the message that start the behavior activity
TIMEOUT	Matches the messages that notify the firing of a timeout
KILL	Matches the messages that notify a request to kill itself
CYCLE	Matches the messages that notify a request the starting of an execution step of a passive actor (see the section “Actor and Executor Implementation”)
ACCEPTALL	Matches all the messages

Table 3. Predefined message patterns.

Behavior

The original actor model associates a behavior with the task of messages processing. In ActoDeS, a behavior does not direct1y process messages, but it delegates such a task to some message handlers that have the goal of processing the messages that match a specific message pattern. Moreover, a behavior can act as a finite state machine where in each state are active a subset of its message pattern – message handler pairs.

A behavior extends the `Behavior` abstract class and must implement the `cases` method (the term “case” is usually used in the actor languages to identify a message pattern – message handler pair). This method performs defines the message pattern – message handler pairs driving the life of the behavior through an instance of that implements the `CaseFactory` interface. This interface provides two methods that must be used for the definition of the message pattern – message handler pairs of the behavior. The execution of such two methods has effect only during the creation of the behavior instance and so such methods have effect only if they are executed inside the `cases`

method. Table 5 provides a short description of the methods the `CaseFactory` interface used for defining the message pattern – message handler pairs.

Method name	**Description**
getReference() : Reference	Gets the reference of the actor
getParent() : Reference	Gets the reference of the parent actor
getState() : BehaviorState	Gets the actor behavior state
setState(BehaviorState s) : void	Sets the actor behavior state
actor(Behavior b) : Reference	Creates a new actor
send(Reference r, Object o) : void	Sends a message without requiring a reply
send(List<Reference> l, Object o) : void	Sends a message without requiring a reply
send(Message m, Object o) : void	Sends a reply without requiring another reply
future(Reference r, Object o, MessageHandler h) : void	Sends a message needing a reply and defines its processing message handler
future(Reference r, Object o, Long t, MessageHandler h) : void	Sends a message needing a reply, defines its processing message handler and sets a timeout for waiting for the reply
future(Message m, Object o, MessageHandler h) :void	Sends a reply needing another reply and defines its processing message handler
future(Message m, Object o, long t, MessageHandler h) :void	Sends a reply needing another reply, defines its processing message handler and sets a timeout for waiting the reply
onReceive(MessagePattern p, MessageHandler h) :void	Defines a message pattern – handler pair used for managing an expected incoming message
onReceive(MessagePattern p, long t, MessageHandler h) :void	Defines a message pattern – handler pair used for managing an expected incoming message and sets a timeout for waiting its arrival

Table 4. Methods implementing the behavior actions.

Method name	**Description**
define(MessagePattern p, MessageHandler h) : void	Defines a message pattern – message handler pair
define(MessagePattern p, MessageHandler h, BehaviorState … s) : void	Defines a message pattern – message handler pair for a set of behavior states

Table 5. Methods for defining the message pattern – message handler pairs.

The execution of a behavior always starts with the execution of one of its message handlers and so with the reception of a message. Usually such a message is received from another actor. However, an actor sends a “start” message to itself during the initialization of the current behavior; therefore, such a behavior can react to its reception with some initialization activities (i.e., it can send messages to other actors).

However, some messages can be managed through some temporary message pattern – message handler pairs (i.e., they can manage a single message). In particular, they can be automatically created by a message handler when either it sends a message that requires a reply (i.e., using a `future` method) or when the message handler needs to do something when it receives a specific type of message or when a timeout fires (i.e., using an `onReceive` method).

As introduced above, an actor can change its behavior and kills itself. An actor can decide to change its behavior after the processing of any message. In fact, the decision is done by using the return value of the message handler (i.e., of its `process` method) that processed the message. If the return value is a behavior class instance, then the actor moves to such a behavior else (i.e., the return value is `null`) the actor maintains the current behavior. Moreover, an actor kills itself either when its current behavior does not define any message pattern – message handler pair or when a `process` method returns a `Shutdown` behavior instance.

As introduced above, a behavior can act as a finite state machine where in each state are active a subset of its message pattern – message handler pairs. The use of behaviors with or without states is only an implementation choice. In fact, a behavior with states corresponds to a set of behaviors without states where each behavior without states uses the message pattern – message handler pairs associated with a state of the behavior with states. The only difference is that, while the message handlers of the behavior with states change its current state, the ones of a behavior without states move the actor to another behavior.

However, an implementation based on behaviors with states reduces the length of the code of the actor and improves the performances because there is not the cost of the creation and initialization of other behaviors. Nevertheless, from the software engineering perspective, an implementation based on behaviors without states is preferred because it offers a clear separation among the implementations of the tasks performed by the different behaviors. Therefore, always from the software engineering perspective, developers should use behaviors with states only when they manage a small, focused set of responsibilities.

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 three main runtime components, (i.e., controller, dispatcher and registry) and of two runtime actors (i.e., the executor and the service provider).

Runtime Components

The controller is a runtime component, defined by the `Controller` c1ass, which configures the actor space and then manages its activities until it ends the execution of the actor space. In particular, it has the duty of initializing the other runtime components and the special actors of the actor space. See the section "Application Configuration" for information on the configuration of an actor space.

The dispatcher is a runtime component defined by the `Dispatcher` class that 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.

The dispatcher works in collaboration with another runtime component defined by the `Connector` class. This component has the duty of opening and maintaining a connection toward all the other actor spaces of the application. In particular, the connectors of one of the actor spaces of an application plays the role of communication broker having the additional duty of maintaining the information necessary for a new actor space for creating connections towards the other actor spaces of the application. Table 6 introduces a short description of the predefined connector implementations.

Class name	Description
`ActiveMqConnector`	Supports the communication among the actor spaces of an application through the ActiveMQ JMS implementation (Snyder et al., 2011)
`JeroMqConnector`	Supports the communication among the actor spaces of an application through a Java implementation of the ZeromMQ (Pronschinske, 2013)
`MinaConnector`	Supports the communication among the actor spaces of an application through the MINA socket library (Apache Software Foundation, 2013)
`RmiConnector`	Supports the communication among the actor spaces of an application through Java RMI (Pitt and McNiff, 2001)

Table 6. Connector implementations.

However, to allow the development of a distributed application the actors of an actor space need to have information about the other actor spaces of the application. In fact, actors need the references of the service providers (e.g., for creating actors in some other actor spaces); moreover, runtime actors may need the references of the other runtime actors (e.g., for providing a global broadcast and coordinating the execution of the different actor spaces of the application). ActoDeS offers a singleton object, defined by the `INFO` c1ass variable of the `SpaceInfo` class, which maintains the relevant information about the current actor space and the information useful for the runtime and application actors to interact with the actors of the other actor spaces. Table 7 introduces a short description of the methods offered by such a class.

Method name	Description
getConfiguration() : Configuration	Gets the information about the configuration of the actor space
getBroadcast(Reference c) : Reference	Gets the local broadcast reference of an actor space giving the reference of an actor of the actor space
getBroadcasts() : Set<Reference>	Gets the local broadcast references of the connected actor spaces
getBroker() : Reference	Gets the reference of the service provider of the actor space acting as communication broker
getPopulation() : int	Gets the number of actors running in the actor space
getProvider(Reference c) : Reference	Gets the reference of the service provider of an actor space giving the reference of an actor of the actor space
getProviders() : Set<Reference>	Gets the references of the service providers of the other actor spaces
getExecutor(Reference c) : Reference	Gets the reference of the executor provider of an actor space giving the reference of an actor of the actor space
getExecutors() : Set<Reference>	Gets the references of the executors of the connected actor spaces
getServices() : Set<String>	Gets the class qualified names of the local actor space services

Table 7. SpaceInfo class methods.

The registry is a runtime component defined by a concrete class that extends the `Registry` abstract class. This component supports the creation of actors and the work of the dispatcher and provides a name service. In particular, it creates the references of the new actors and supports the delivery of the messages coming from remote actors by proving the reference of the final destination to the dispatcher. In fact, as introduced in a previous section an actor can send a message to another actor only if it has its reference (or its name). However, while the reference of a local actor allows the direct delivery of messages, the reference of a remote actor delegates the delivery to the dispatchers of the local and remote actor spaces. Figure 3 shows the delivery of messages between local and remote actors. Table 8 introduces a short description of the predefined registry implementations.

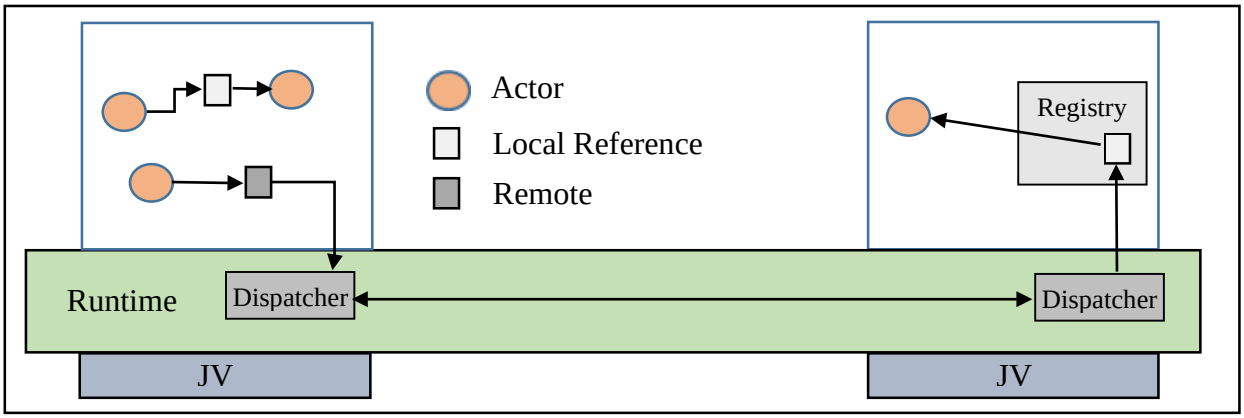

Figure 3. Message dispatching.

Class name	Description
`DefaultRegistry`	Assigns a reference to a new actor up to the number of created actors reaches a value defined in the configuration of the actor space
`PersistentRegistry`	Assigns a reference to a new actor up to the number of created actors reaches a value defined in the configuration of the actor space. Moreover, it stores inactive actors in a persistent storage and reloads them when they receive a new message
`TemporaryRegistry`	Assigns a reference to a new actor up to the number of created actors reaches a value defined in the configuration of the actor space. Moreover, it stores inactive actors in the memory and reloads them when they receive a new message

Table 8. Registry implementations.

Class name	Description	Mandatory
`Broadcaster`	Broadcasts messages	Yes
`Creator`	Creates new actors	No
`FileStorer`	Provides a file-based persistent storage for actors and checkpoints of actor spaces.	Yes
`Grouper`	Manages communication groups	No
`Logger`	Logs information about the execution of actors and runtime components	Yes
`Mobiler`	Creates mobile actors and moves them between actor spaces	No
`Namer`	Binds names to actors and allows to use such names for getting their references	NO

Table 9. Actor space services.

Service Provider

The service provider is a runtime actor that offers the possibility of performing new kinds of action to the actors of an application by proving them a set of services. In fact, the actors of an application can require the execution of such services by sending a message to a service provider (i.e., an actor

can ask a service to a service provider of any actor space of the application). Moreover, each actor space of an application can provide a different set of services. Table 9 introduces a short description of the predefined actor space services.

Developers can add new kinds of service. Such an operation requires work at the application and runtime layers. At the application layer, it is necessary to develop the classes representing the service requests that actors need to send to the actor spaces. At the runtime layer, it is necessary to develop the software components able to perform the services.

Actor Space Services

As introduced above, an actor usually needs to send a message to the service provider for taking advantage of its services. ActoDeS provides some services (e.g., the communication services) that an actor can use by simply knowing the associated references and others, called runtime services, that it uses without the need of performing actions (e.g., the `FileStorer`).

The broadcasting service is a mandatory service of an actor space and so all the service providers of an application provide it. This service allows the delivery of messages to all the actors either of the local actor space or of the application. As introduced in a previous section, actors can use such service by addressing the messages to the references bound to their `SPACE` and `APP` class variables.

The creation service is an optional service that supports the creation of actors. However, given that an actor can directly create other actors in its actor space, only remote actors should use it. Such a service accepts some `Create` requests and answers with messages containing the references of the new actors.

The communication group (from here simply named group) service is an optional service that supports the management of the exchange of messages among a set of actors. A group is identifiable by a name and takes advantage of a multicast reference to support the exchange of messages.

In particular, an actor can register a group, (using the `Register` request) deciding if the group is open or closed. This actor becomes the owner of the group and can add or remove itself and other actors to/from the group (using the `Signup` and the `Cancel` requests) and, finally, can close the group (using the `Deregister` request). If the group is open, then all the actors can add or remove themselves to/from the group. If the group is closed, then all the actors can only remove themselves from the group. An actor can have different roles in the group: i) subscriber, i.e., the actor receives the messages exchanged in the group, ii) publisher, i.e., the actor can send messages to the group and iii) exchanger: i.e., the actor acts as both the subscriber and the publisher roles. Of course, actors of different actor spaces of an application can belong to the same group.

The mobility service is an optional service that supports the creation of mobile actors and their movement between actor spaces. This service does not require the use of special messages, but only the use of the `Mobile` and the `Mover` behaviors. In fact, an actor can create mobile actors if it uses a behavior that extends the `Mobile` behavior. Moreover, a mobile actor can move to another actor space by changing its current behavior to the `Mover` behavior. This behavior performs the migration of its actor by interacting with the mobility services of the local and remote actor spaces. A mobile actor has a special reference that allows the reception of messages without the need that the sender knows its location. In fact, this reference delivers messages by using a home base approach (i.e., messages are delivered to the home location of the mobile actor and then to its real location). Of course, a mobile actor can only travel between the actor spaces that provide the mobility service.

The naming service is an optional service that supports the binding (`Bind` request) and unbinding (`Unbind` request) between a name and the reference of an actor, and that allows the use of this name for getting the reference of the related actor (using the `Lookup` request). However, an actor can directly manage the binding of its reference and getting the reference of another actor from its name

in its local actor space; therefore, an actor should only use the naming service of some remote actor spaces for simplifying the interaction with the actors of such actor spaces.

The logger is a mandatory runtime service that supports the logging of an application. In particular, this service is defined by the `Logger` c1ass and allows the writing on a stream some information on the relevant events that happen during the execution of an actor space (e.g., creation and deletion of actors, sending of messages, processing of messages, and initialization of behaviors). Such a service takes advantage of the Java Logging API (Gilstrap, 2002) and provides information in both the textual and binary formats. Moreover, it is possible to decide the types of event to log, where export them (by using some writer components) and defines the format of log messages for the writer components (by using some formatter components). In particular, the types of event to log is defined by combining some of a set of filter constant fields, defined in the Logger c1ass, through the Java bitwise or operator (|). Table 10 introduces a short description of the scope of such filter constant fields. Moreover, it is possible to refine the selection of the events to be logged (i. e., it allows the selection of the logging events of a subset of actors) by using an object filter whose class implements the `Constraint` interface.

Of course, such an information can be very useful for understanding the activities of the application and for diagnosing execution problems. Moreover, the binary format of such information contains real copies of the application objects (e.g., messages and actor data) and so some support tools of the software framework can use such an information for performing different kinds of task (e.g., monitoring and simulation tools).

Field name	**Description**
ACTORCREATION	Logs the creation of actors
BEHAVIORINITIALIZING	Logs the starting of the initialization of behaviors
BEHAVIORINITIALIZED	Logs the end of the initialization of behaviors
OUTPUTMESSAGE	Logs the sending of messages
MESSAGEPROCESSING	Logs the starting of the processing of messages
MESSAGEPROCESSED	Logs the end of the processing of input messages
MESSAGENOMATCHED	Logs the input messages that do not match any message pattern
ACTORSHUTDOWN	Logs the shutdown of actors
ACTIONS	Logs all the events corresponding to the above types of logging
CONFIGURATION	Logs the information about the actor space configuration
EXECUTION	Logs the execution time and the number of created actors
STEP	Logs steps information for a passive executor
DATALOADING	Logs information about the resource from which data are loaded
DATASAVED	Logs information about the resource where data were saved
REGISTRY	Logs the executions of the registry methods
DISPATCHER	Logs the executions of the dispatcher methods
STORER	Logs the executions of the persistent storage manager methods

Table 10. Logger filtering fields.

The persistent storage service is a mandatory runtime service that provides a persistent storage where maintains both actors and checkpoints of actor spaces. This service is defined by the `Storer` interface and the current release of ActoDeS provides a file-based implementation of the service

defined by the `FileStorer` c1ass. Note that this service is a runtime service used by the controller for saving checkpoints of actor spaces and by some executors for saving idle actors.

Class name	Description
`ActiveActor`	Implements active actors. It performs all the actions of the actor until it kills itself
`CycleActor`	Implements passive actors. It provides a `step` method that processes the messages received by the actors in the last scheduling cycle
`OldActor`	Implements passive actors. It provides a `step` method that processes the messages received by the actors before the execution of such a method
`OneActor`	Implements passive actors. It provides a `step` method that processes the first oldest message in its queue
`SavableActor`	Implements passive actors. It provides a `step` method that processes the messages received by the actors in the last scheduling cycle. Moreover, it provides a method that gives information about the current number of inactivity cycles of the actor
`SharedActor`	Implements passive actors. It uses a mailbox that gets the messages from a queue shared with the other actors of the actor space. Moreover, it provides a `step` method that processes the messages received in the last scheduling cycle

Table 11. Actor implementations.

Class name	Description
`CycleScheduler`	Works in the actor spaces containing only `CycleActor` instances. Runs the actors by cyclically calling their `step` method
`OldScheduler`	Works in the actor spaces containing only `OldActor` instances. Runs the actors by cyclically calling their `step` method
`OneScheduler`	Works in the actor spaces containing only `OneActor` instances. Runs the actors by cyclically calling their `step` method
`PersistentScheduler`	Works in the actor spaces containing only `SavableActor` instances. Runs the actors by cyclically calling their `step` method. Moreover, it asks the `PersistentRegistry` to store idle actors in a persistent storage and to reload them in the scheduler when they receive a new message
`PoolCoordinator`	Works in the actor spaces containing only `ActiveActor` instances. Runs the actors by using a Java thread pool
`SharedScheduler`	Works in the actor spaces containing only `SharedActor` instances. Runs the actors by cyclically calling their `step` method
`TemporaryScheduler`	Works in the actor spaces containing only `SavableActor` instances. Runs the actors by cyclically calling their `step` method. Moreover, it asks the `TemporaryRegistry` to maintains idle actors in memory and to reload them in the scheduler when they receive a new message
`ThreadCoordinator`	Works in the actor spaces containing only `ActiveActor` actors. Runs the actors by creating a number of Java threads equal to their number

Table 12. Executor implementations.

Executor

An executor is a runtime actor that defines the implementation and manages the execution of the actors of an actor space, and may create the initial set of actors. Of course, the duties of an executor depend on the type of implementation of its actors and, in particular, on the type of threading solutions used in their implementation. In fact, an actor can have its own thread (from here named active actors) or can share a single thread with the other actors of the actor space (from here named passive actors).

Hence, while the executors of active actors (called coordinators) delegate the large part of their work to the Java runtime environment, the executors of passive actors (called schedulers) must completely manage their execution by initializing them and cyclically calling their methods that process the messages in their mailboxes. As introduced above, an executor has the duty of creating the initial set of actors of an actor space; to do it, the constructor of an executor has as argument either an instance of an actor behavior, when the initial set contains a single actor that has the duty of creating the other actors of the actor space, or an instance of a c1ass which extends the `Builder` c1ass and whose code must create the initial set of actors.

Actor and Executor Implementations

However, the implementations of an actor and of an executor do not only depend on the threading solution, but on other implementation solutions (e.g., example, the use of different mailbox implementations and the use of different rules for processing messages) that allow an improvement of the efficiency of specific applications. Table 11 and Table 12 introduces a short description of the predefined implementations of actors and executors. In particular, the “cycle” and “shared” actor and executor implementations for the development of simulation applications. In particular, such actor implementations differ from the other because they can set the length of the execution and of timeouts for waiting for a new message both in milliseconds and in scheduling cycles.

Package name	**Description**
`broadcasting`	Shows the use of the broadcasting service
`buffer`	Is a starting level example that shows how to create an application, how to build multi-behavior actors, and how to implement communication between actors
`fibonacci`	Is a starting level example that shows how to create a hierarchy of actors build multi-behavior actors, and how to implement communication between actors
`persistence`	Shows how the use of a specific actor executor allows to store/load inactive actors to/from a persistent storage through a simple master - workers application.
`messaging`	Shows how to create a distributed application and how to use the creation service
`mobile`	Shows how to create a distributed application involving mobile actors
`naming`	Shows the use of the naming service and how it can be used for synchronizing some actors
`pubsub`	Shows the use of the group communication service for implementing a topic based publish – subscribe application
`simulation`	Shows how to build a simulation application of the classical dining philosophers concurrency problem
`unreliability`	Shows the use of future messages with timeouts for manages unreliable messages

Table 13. Examples.

Application Development and Execution

Using ActoDeS is possible to build both standalone and distributed applications. A standalone application deploys a single actor space created with a set of initial actors that will start the execution of the application. A distributed application involves more than one actor space. Each actor space can have set of initial actors, but usually a distributed application starts with a set of empty actor spaces and an "initiator" actor space with a set of initial actors that have also the duty of creating the initial set of actors of the other actor spaces. After the identification of the actors involved in the application and the definition of the deployment of such actors on one or more actor spaces, the two main steps for obtaining an executable code of the application are:

- The design and coding of all the classes defining the different behaviors of the actors.
- The configuration of the actor spaces.

Moreover, for some applications is necessary to work on the design and coding of all the classes defining: i) the virtual and real entities defining the environment where actors act and ii) the content of the messages supporting the interaction among actors.

ActoDeS provides a set of examples that try to emphasize the main aspects of the framework. Table 13 introduces a short description of such examples.

```
public abstract class Buffer extends Behavior
{
  private static final long serialVersionUID = 1L;

  protected static final MessagePattern GETPATTERN =
      MessagePattern.contentPattern(new IsInstance(Get.class));
  protected static final MessagePattern PUTPATTERN =
      MessagePattern.contentPattern(new IsInstance(Put.class));

  protected BufferQueue queue;

  protected MessageHandler getCase;
  protected MessageHandler putCase;

  public Buffer(final BufferQueue q)
  {
    this.queue = q;

    this.getCase = (m) -> {
      send(m, this.queue.remove());

      if (this.queue.size() == 0)
      {
        return new EmptyBuffer(this.queue);
      }
      else if (getClass().getName().equals(FullBuffer.class.getName()))
      {
        return new PartialBuffer(this.queue);
      }

      return null;
    };

    this.putCase = (m) -> {
      this.queue.add(((Put) m.getContent()).getElement());
      send(m, Done.DONE);

      if (this.queue.size() == this.queue.getCapacity())
      {
        return new FullBuffer(this.queue);
      }
      else if (getClass().getName().equals(EmptyBuffer.class.getName()))
      {
        return new PartialBuffer(this.queue);
      }

      return null;
    };
  }
}
```

Figure 4. Buffer class code.

```
public final class EmptyBuffer extends Buffer
{
  private static final long serialVersionUID = 1L;

  public EmptyBuffer(final BufferQueue q)
  {
    super(q);
  }

  public void cases(final CaseFactory c)
  {
    c.define(PUTPATTERN, this.putCase);
    c.define(KILL, DESTROYER);
  }
}
public final class PartialBuffer extends Buffer
{
  private static final long serialVersionUID = 1L;

  public PartialBuffer(final BufferQueue q)
  {
    super(q);
  }

  public void cases(final CaseFactory c)
  {
    c.define(GETPATTERN, this.getCase);
    c.define(PUTPATTERN, this.putCase);
    c.define(KILL, DESTROYER);
  }
}
public final class FullBuffer extends Buffer
{
  private static final long serialVersionUID = 1L;

  public FullBuffer(final BufferQueue q)
  {
    super(q);
  }

  public void cases(final CaseFactory c)
  {
    c.define(GETPATTERN, this.getCase);
    c.define(KILL, DESTROYER);
  }
}
```

Figure 5. EmptyBuffer, PartialBuffer and FullBuffer class code.

```
public final class StateBuffer extends Behavior
{
  private static final long serialVersionUID = 1L;

  private static final MessagePattern GETPATTERN =
      MessagePattern.contentPattern(new IsInstance(Get.class));
  private static final MessagePattern PUTPATTERN =
      MessagePattern.contentPattern(new IsInstance(Put.class));
  private final Queue<Integer> queue;
  private int capacity;

  private enum BufferState implements BehaviorState {
    EMPTY, PARTIAL, FULL;
  }

  public StateBuffer(final int c) {
    super(BufferState.EMPTY, BufferState.values());
    this.queue    = new LinkedList<Integer>();
    this.capacity = c;
  }

  public void cases(final CaseFactory c) {
    MessageHandler h = (m) -> {
      send(m, this.queue.remove());

      if (this.queue.size() == 0)
      {
        setState(BufferState.EMPTY);
      }
      else if (getState() == BufferState.FULL)
      {
        setState(BufferState.PARTIAL);
      }

      return null;
    };

    c.define(GETPATTERN, h, BufferState.FULL, BufferState.PARTIAL);

    h = (m) -> {
      this.queue.add(((Put) m.getContent()).getElement());
      send(m, Done.DONE);

      if (this.queue.size() == this.capacity)
      {
        setState(BufferState.FULL);
      }
      else if (getState() == BufferState.EMPTY)
      {
        setState(BufferState.PARTIAL);
      }

      return null;
    };

    c.define(PUTPATTERN, h, BufferState.EMPTY, BufferState.PARTIAL);
    c.define(KILL, DESTROYER);
  }
}
```

Figure 6. StateBuffer class code.

Behavior Definition

A behavior extends the `Behavior` abstract class. This class provides a partial implementation of a behavior relaying to the concrete classes the definition of its message pattern – message handler pairs.

In the following the definition of a behavior is described through the presentation of some code of two different implementations of a buffer application: the first implementation does not use the behavior states (see Figure 4 and Figure 5), the second implementation uses the behavior states (see Figure 6). Note that such buffer implementations are included in the ActoDeS software distribution (see the package it.unipr.sowide.actodes.examples.buffer).

Message Content Definition

Even if ActoDeS does not constrain actors to exchange immutable data, a good rule is to provide an immutable implementation of the content of the messages when there are not advantages in using a mutable implementation (e.g., reuse of preexistent software). ActoDeS provides a small set of immutable message contents that support the basic interactions between actors and with a service provider. Table 14 introduces a short description of the predefined message contents. Figure 7 shows the `Put` class; this class defines an immutable implementation of the storage requests, used in the buffer application described in the previous section.

Class name	Description
`Become`	Requests to an actor to change its behavior
`Create`	Requests the creation of a new actor
`Done`	Informs an actor about the successful processing of an its message
`Error`	Informs an actor about an delivery or processing error of an its message
`Forward`	Forwards a message to another actor
`Kill`	Requests to an actor to kill itself
`Lock`	Requests to an actor to start its exclusive service
`Start`	Requests to an actor to start itself
`Status`	Informs an actor about the fact that the sender is alive, is stopping itself, or is killing itself
`Stop`	Requests to an actor to stop itself
`Unlock`	Requests to an actor to end its exclusive service

Table 14. Predefined message content classes.

```
public final class Put implements Request
{
  private static final long serialVersionUID = 1L;

  private final int element;

  public Put(final int e)
  {
    this.element = e;
  }

  public int getElement()
  {
    return this.element;
  }
}
```

Figure 7. Put class code

```
public final class Initiator extends Behavior
{
  private static final long serialVersionUID = 1L;

  // Default configuration values.
  private static final long DURATION = 1000;
  private static final int CAPACITY  = 100;
  private static final int PRODUCERS = 10;
  private static final int CONSUMERS = 10;

  private final long duration;
  private final int capacity;
  private final int nProducers;
  private final int nConsumers;
  private final Behavior behavior;

  private HashSet<Reference> rProducers;
  private HashSet<Reference> rConsumers;

  public Initiator(final boolean f, final long t,
      final int s, final int p, final int c)
  {
    this.duration = t;
    this.capacity = c;

    this.nProducers = p;
    this.nConsumers = c;

    this.rProducers = new HashSet<>();
    this.rConsumers = new HashSet<>();

    if (f)
    {
      this.behavior = new EmptyBuffer(new BufferQueue(c));
    }
    else
    {
      this.behavior = new StateBuffer(this.capacity);
    }
  }

  /** {@inheritDoc} **/
  @Override
  public void cases(final CaseFactory c)
  {
    // cases code ...
  }

  public static void main(final String[] v)
  {
    // main code ...
  }
}
```

Figure 8. Initialization class code of the buffer example.

```
public void cases(final CaseFactory c)
{
  MessageHandler h = (m) -> {
    if ((this.duration > 0) && (this.capacity > 0))
    {
      MessageHandler t = (n) -> {
        this.rProducers.forEach(r -> send(r, Kill.KILL));
        this.rConsumers.forEach(r -> send(r, Kill.KILL));
        send(this.behavior.getReference(), Kill.KILL);

        return Shutdown.SHUTDOWN;
      };

      Reference r = actor(this.behavior);

      for (int i = 0; i < this.nProducers; i++)
      {
        this.rProducers.add(actor(new Producer(r)));
      }

      for (int i = 0; i < this.nConsumers; i++)
      {
        this.rConsumers.add(actor(new Consumer(r)));
      }

      onReceive(ACCEPTALL, this.duration, t);
    }

    return null;
  };

  c.define(START, h);
}
```

Figure 9. Initialization class code of the buffer example: the cases method.

Applications Initialization and Configururation

Often an ActoDeS application is started by a specific initialization behavior that has the duty of setting the configuration parameters and creating the initial set of actor (Figure 8, Figure 9 and Figure 10 show the code of the behavior that starts both the buffer implementations). An ActoDeS application fails during its startup when it is not able to get the configuration information or when it has an incomplete or incorrect set of configuration information. In particular, an object, defined by the `Configuration` c1ass, has the duty of maintaining such information. Configuration information depends on the type of application and on its deployment. Moreover, different implementations are possible for the actors and the runtime components of a specific application. Finally, some information can be absent because there are optional components or because some components (i.e., the mandatory services) take advantage of a default configuration if their configuration information is absent. In particular, the configuration:

- Must contain the definition of the executor.
- Must contain the definition of the connector if the application is distributed.

- May contain an alternative implementation of the other runtime components.
- May contain the definition of a set of actor space optional services.
- May contains some sets of property name – value pairs used for specific application configurations.

```
public static void main(final String[] v)
{
  Configuration c =  SpaceInfo.INFO.getConfiguration();

  boolean f = c.load(v);

  final long duration = f ? c.getLong("duration") : DURATION;
  final int capacity  = f ? c.getInt("sixe") : CAPACITY;
  final int producers = f ? c.getInt("producers") : PRODUCERS;
  final int consumers = f ? c.getInt("consumers") : CONSUMERS;

  c.setFilter(Logger.ACTIONS);
  c.setLogFilter(new NoCycleProcessing());

  c.addWriter(new ConsoleWriter());

  //c.addWriter(new BinaryWriter("examples/buffer"));

  Scanner scanner = new Scanner(System.in);

  System.out.println("Enter:");
  System.out.println(" b for the behavior change implementation");
  System.out.println(" s for the state change implementation");

  String s = scanner.next();

  scanner.close();

  switch (s)
  {
    case "b":
      c.setExecutor(new OldScheduler(new Initiator(true, duration,
          capacity, producers, consumers)));
      break;
    case "s":
      c.setExecutor(new PoolCoordinator(new Initiator(false, duration,
          capacity, producers, consumers)));
      break;
    default:
      return;
  }

  c.start();
}
```

Figure 10. Initialization class code of the buffer example: the main method.

In particular, Figure 10 shows the main method starting the buffer application. This code enables: i) the loading of a configuration file (used for providing alternative configuration values), ii) the refinement of the filtering of the events to be logged by using a constraint defined by the `NoCycleProcessing` class (this class avoids the logging of the events corresponding to the processing of the cycle messages sent by the executor to all the actors of the application), iii) the

running of one of the two implementations of the buffer example, and iv) the instantiation of one implementation of the executor passing the `Initiator` behavior to its constructor.

Execution

As introduced above, both configuration and starting code of an actor space can be located in a `main` method of a class. Moreover, if the actor space has an initial actor, then the best place for such a main method is the class defining the initial behavior of the class. If the actor space has not an initial actor (i.e., the actor space belongs to a distributed application and its actors are created by actors of other actor spaces), then the main method can be placed in an empty class. Usually the initial configuration of a distributed application creates one or more actors on an actor space and then such actors have the duty of creating some actors on the other actor spaces. In this case, the startup of the actor spaces needs to begin from an empty actor space (called broker) that acts as communication broker, then the other empty actor spaces (simply called nodes) and finally the actor space (called initiator) with the initial set of actors. Figure 11 how to launch an application (i.e., the buffer example provided in ActoDeS distribution) assuming that the jar file, containing the ActoDeS class files, are in the current directory. In particular, the second example shows how to pass a properties file for defining a new application configuration. Of course, the use of an IDE tool can simplify the launching of an application.

```
java –cp actodes-1.2.jar it.unipr.sowide.actodes.examples.buffer.Initiator

java –cp actodes-1.2.jar it.unipr.sowide.actodes.examples.buffer.Initiator
     -cfg ./src/main/resources/buffer-config.properties
```

Figure 11. Examples of the use of the java tool for launching an application.

Termination

The default termination condition of a standalone application is reached when there are not running actors in its actor spaces. Moreover, an actor can ask the termination of the actor space, and, if this actor acts as broker of the application, then all the actor spaces of the application terminate. Finally, an executor can also ask the termination of an actor space by sending a `Kill` message to the service provider. In particular, it happens when the termination condition, defined by the executor configuration information, becomes true.

References

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

Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L. (1997). A Foundation for Actor Computation. Journal of Functional Programming, 7(1):1-69.

Apache Software Foundation. (2013). Apache Mina Framework. http://mina.apache.org.

Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W. (2006). Ambient-oriented programming in ambienttalk. In ECOOP 2006 - Object-Oriented Programming, pp. 230-254. Springer: Berlin, Germany.

Gilstrap, B.R., (2002). An introduction to the java logging API. http://homepages.inf.ed.ac.uk/stg/teaching/ec/handouts/logging.pdf.

Haller, P., Odersky, M. (2009). Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science, 410(2-3):202–220.

Hewitt, C.E. (1977). Viewing controll structures as patterns of passing messages. Artificial Intelligence, 8(3):323–364.

Hintjens, P. (2013). ZeroMQ: Messaging for Many Applications. O'Reilly, O'Reilly, Sebastopol, CA.

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

Pronschinske, M. (2013). JeroMQ: Pure Java ZeroMQ. DZone. http://java.dzone.com/articles/jeromq-pure-java-zeromq.

Philippsen, M. (2000). A survey of concurrent object-oriented languages. Concurrency: Practice and Experience, 12(10):917-980.

Pitt, E., McNiff, K. (2001). Java.rmi: the Remote Method Invocation Guide. Addison-Wesley: Boston, MA, USA.

Snyder, B., Bosnanac, D., Davies, R. (2011). ActiveMQ in action. Manning: Westampton, NJ, USA.

Srinivasan, S., Mycroft, (2008). A. Kilim: Isolation-Typed Actors for Java. In J. Vitek ed. ECOOP 2008 – Object-Oriented Programming, Lecture Notes in Computer Science, 5142, pp. 104-128, Springer: Berlin, Germany.

Varela, C., Agha, G.A. (2001). Programming dynamically reconfigurable open systems with SALSA. SIGPLAN Notices, 36(12):20-34.

Appendix – Installing the software and managing external dependencies

ActoDeS is distributed with a compressed Eclipse project folder including the source code and the Javadoc documentation (see the "target/site/apidocs" project directory). Therefore, its installation is very simple: you need only to extract all the files from the compressed folder. If you will use it through the Eclipse IDE, then you need to create a new Eclipse Java project located in the ActoDeS directory that has been created in the file system of your computer during the extraction of the files from the compressed Eclipse project folder. Moreover, given that last version of ActoDeS uses constructs provided by JDK 12 , then its necessary to have JDK 12 (or higher) installed and to configure Eclipse to use it. Figure 13 shows how to create an Eclipse project for the ActoDeS software and show where to set the Java Runtime Environment.

The final step of the installation is to resolve the dependencies with some external libraries. It can be done in two ways:

1) If your IDE provides the Maven tool, it is only necessary to select the pom.xml file and update the project dependencies (Figure 12 show how to do it with Eclipse).
2) Otherwise, you need to add manually all the jar files in the ActoDeS "lib" directory to the project property "Java build path (Figure 13 show how to do it with Eclipse).

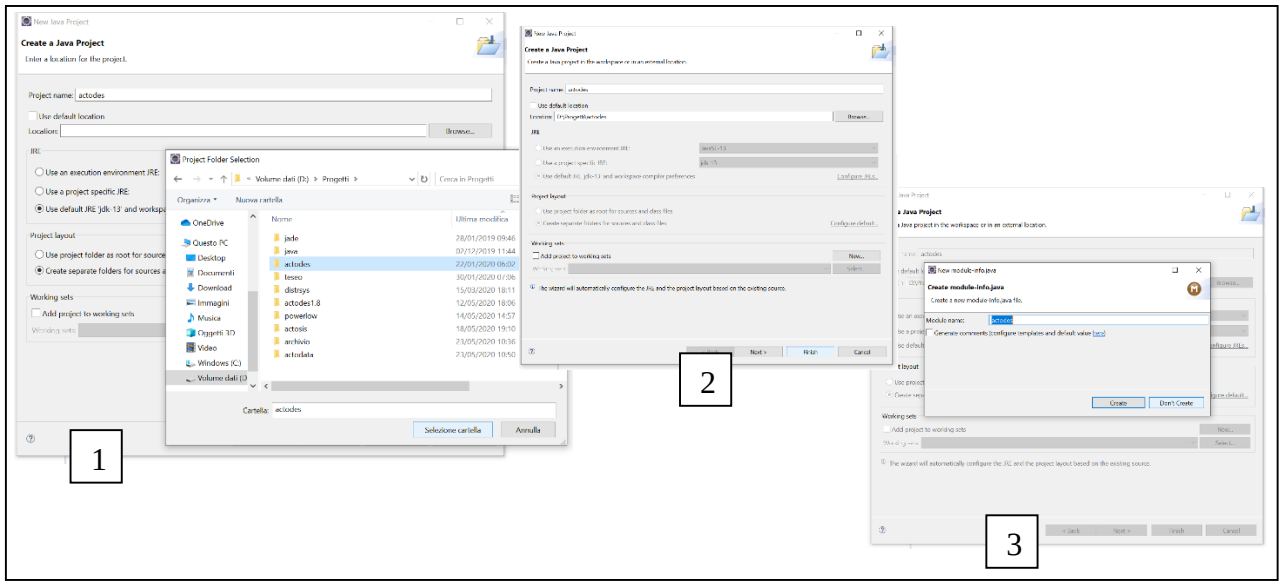

Figure 12. Installing ActoDeS as Eclipse project.

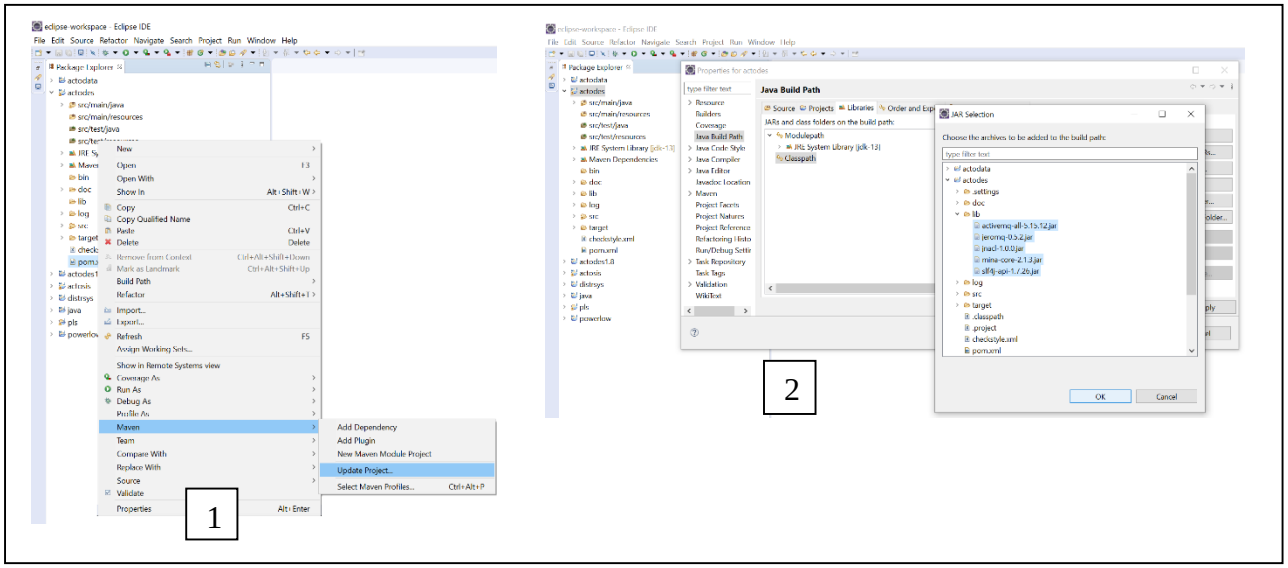

Figure 13. The two possible ways for resolving ActoDeS dependencies with external libraries.