About Akka

Mariano Basile, Angelo Buono, Corrado De Sio

Contents

1 What is Akka?

Akka is an open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM. Akka supports multiple programming models for concurrency, but it emphasizes actor-based concurrency, with inspiration drawn from Erlang.

Language bindings exist for both Java and Scala. Akka is written in Scala and Akka's actor implementation is included as part of the Scala standard library.

1.1 Why Akka?

Write correct concurrent, fault-tolerant and scalable applications is too hard. Most of the time it's because we are using the wrong tools and the wrong level of abstraction. Akka is here to change that. Using the Actor Model we raise the abstraction level and provide a better platform to build scalable, resilient and responsive applications.

1.2 Features

1.2.0.1 Actors Paradigm

- ▷ Simple and high-level abstractions for concurrency and parallelism.
- ▷ Asynchronous, non-blocking and highly performant event-driven programming model.
- ▷ Very lightweight event-driven processes (several million actors per GB of heap memory)

1.2.0.2 Fault Tolerance

- ▷ Supervisor hierarchies with "let-it-crash" semantics.
- ▷ Supervisor hierarchies can span over multiple JVMs to provide truly fault-tolerant systems.
- ▷ Excellent for writing highly fault-tolerant systems that self-heal and never stop

1.2.0.3 Location Transparency

- ▷ Everything in Akka is designed to work in a distributed environment: all interactions of actors use pure message passing and everything is asynchronous.

Persistence

- ▷ Messages received by an actor can optionally be persisted and replayed when the actor is started or restarted.
- ▷ This allows actors to recover their state, even after JVM crashes or when being migrated to another node.

2 More about Akka and Actor System

2.1 Actors Model

Akka's unit of code organization is called an Actor. The Actor model adopts the philosophy that everything is an actor. This is similar to the everything is an object philosophy used by some object-oriented programming languages, but differs in that object-oriented software is typically executed sequentially, while the Actor model is inherently concurrent.

An actor is a computational entity triggered by messages it receives. In the Actor model implemented by Akka there is not shared state, state visibility, threads, locks, concurrent collections, thread notifications. All this problems are solved by the Actor model using a message passing paradigm. Akka's Actor model is designed to be completely transparent and distributable.

2.2 Hello Actor!

Let'see a simple programm exploiting the concept exposed above.

2.2.1 Messages

```
1 public class Greeting implements Serializable {
2         public final String who;
3         public Greeting(String who) { this.who = who; }
4         }
```

Each Actor is triggered by messages. Messages are standard classes that implements Serializable interface. The Message *Greeting* has only one field *who*.

2.2.2 Actors

```
1 public class GreetingActor extends UntypedActor {
2         LoggingAdapter log = Logging.getLogger(getContext().system(), this);
3
4
5         public void onReceive(Object message) throws Exception {
6                 if (message instanceof Greeting)
7                         log.info("Hello " + ((Greeting) message).who);
8         }
9                 }
```

Each Akka's Actors extend the class *UntypedActor*.

GreetingActor has no fields and inherites *onReceive* method by *UntypedActor*.

Each Actor has only one *onReceive* method triggered by any messages send it by someone (we will go deeper about send messages soon).

onReceive's parameter is a generic *Object*, (so when you need to operate on it you have to perform casting) and its body define the Actor's behaviour.

In our example when a instance of *GreetingActor* receive a *Greeting* object it prints a string on log.

```
ActorSystem system = ActorSystem.create("MySystem");

ActorRef greeter = system.actorOf(new Props(GreetingActor.class), "greeter");
```

2.2.3 ActorSystem and Actor Configuration

Akka's Actors are implemented with a hierarchical structure. Root of this structure is the Actor System. Actors are created with a factory method provided by Actor System or by the UnypedActorRef, a reference for the actor. This reference is available in the 'getContext()' method in the UntypedActor.

The factory method *actorOf* has a mandatory parameter *Props* initialized with the .class of the Actor I need to create. In our example the second parameter is the name of new Actor.

2.2.4 Triggering an Actor

```
greeter.tell(new Greeting("Charlie_Parker"))
```

tell method of the ActorRef is what allow us to send a message to an Actor. The *tell* is asynchronous and non-blocking, the Actor is passive until a message is sent to it, which triggers something within. Anyone can operate a *tell* method on an Actor and evreything is lockless and asynchronus.

2.3 Hierarchies

Actors' hierarachy is an Akka's strong feature. The hierarchial system provide an easy way to manage crash and to specify univocally an Actor.

Actor Systems and Actors can create other Actors. This new Actors are *child* w.r.t. their creator and are ordered and can be identified like in classic

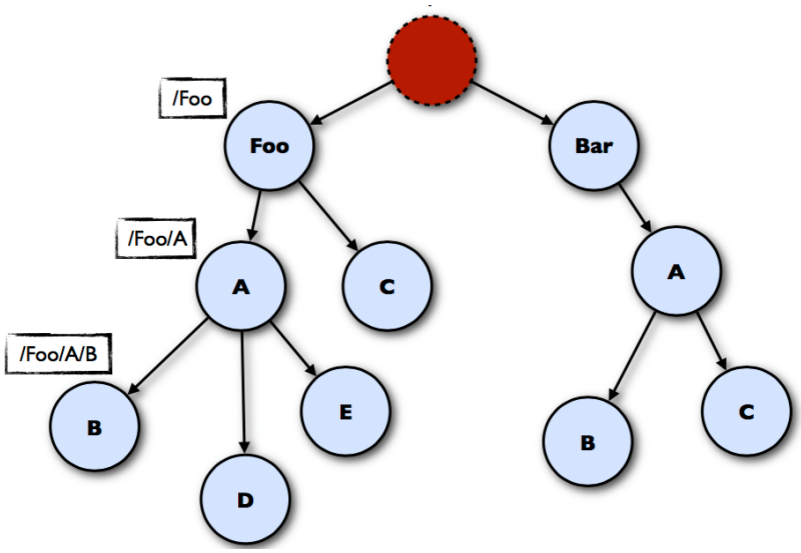

Figure 1: Akka Hierarchy

2.3.1 Failure Recovery

A *father* Actor is a supervisor for its *child* and if a *child* crash a notification will be sent to his supervisor, who can react upon the failure. This is a strong feature that allow to manage crash and errors in a solid way demanding jobs

that was assigned at the crashed actors to others. A *father* Actor who receives a crash notification can change its behaviour to replace crashed Actor or recovery the crashed Actor or manage the crash how the programmer wish.

2.4 Distribuited Actors

To move our code in a distribuited system is easy. The code we showed above works in a distribuited architecture too . I don't need to care about where my Actors are, I only need an *ActorRef* to send messages. All is done in a transparent way. I just need to create a configuration file for each Actor. In this file I choose the port where my Actors are listening.

```
akka {
  actor { provider = "akka.remote.RemoteActorRefProvider" }
  remote { netty.tcp { hostname = "127.0.0.1" port=2553 } }

}
```

The general pattern used to find an actor on a remote node is the following:

```
akka.<protocol>://<actorsystemname>@<hostname>:<port>/<actor path>
```

To obtain the *ActorRef* of the Remote Actor I have to pass this string as a parameter of the constructor of the Client Actor to allow to manage the connection between the two Actors. After this step I can use my *ActorRef* in the same way of the local one.

3 Starting with Akka

Akka requires that you have Java 8 or later installed on your machine.

Typesafe provides a commercial build of Akka and related projects such as Scala or Play as part of the Reactive Platform. This is useful and reccomanded to start because the programmer don't have to worry about dependancies problem that could be an issue for beginners.

3.1 Typesafe Activator

3.1.1 Getting Started

There are several ways to download Akka. The best way to start learning Akka is to download Typesafe Activator. It is a tool for getting started with Typesafe platform and it's the easiest way to do so. There are many templates showing how to do simple and complex task and it is also a learning tool.

3.1.2 Start Activator's UI

Once installed you can run Typesafe Activator's web interface either from a file browser or from a command line. From a file browser, double-click on either the activator (for Mac or Linux) or *activator.bat* (for Windows) script. From a command line, run the script with a ui argument: *activator ui*. This will start a local web server that can be reached at: *http://localhost:8888*.

3.1.3 Create An Application Using the Web Interface

Inside your browser you can now create a new application based on one of the 389 templates. Both Typesafe and the community have contributed templates to Activator.

Once you have selected a template and optionally entered a name and location, select the "Create" button to have your new application created.

3.1.4 Open an Existing Application Using the Web Interface

Existing applications can be opened by running *activator* or *activator.bat* from a project's root directory. If the Activator UI is already running, then open *http://localhost:8888/home* in your browser and either select a known existing app, or folder icon next to Open existing app to browse to an existing app.

3.1.5 Working with Applications in the Activator UI

Once you have created or opened an application in the Activator UI you can do much with it:

1. Read the Tutorial
2. Browse & edit the code (select Code)
3. See the compile output (select Compile)
4. Test the application (select Test)
5. Run the application (select Run)
6. Inspect the application (select Inspect)
7. Whenever you save changes to a source file in the application, the changes will be recompiled, the application will be re-run, and the tests will be re-run.

The UI is intuitive and easy to use but you can still write your code with the IDE you prefer and then compile it and run it at command line.

3.1.6 Developer License ID

Your Developer License ID is your key to using Reactive Platform with your existing Scala, Java 8, Akka and Play projects. The Reactive Platform is a set of JARs that integrates simply into your project build process. It's available free for development and requires a Subscription for production usage.

3.1.7 Activator Command Line

Activator can be used at command line too in the directory of the project. Useful commands are *activator run* to execute the project and *activator compile* to compile it. More can be learned with *activator help*.

A Programming in Akka: Online Chat

Introduction

The idea is to write an Online Chat in Java using Akka where:

- ▷ Client-Server paradigm is used.
- ▷ Clients are many and run on many machines.
- ▷ Servers is one and is located.
- ▷ Each Client choose an username and ask to the Server the allow to partecipate with that (unique) name in the chat.
- ▷ Each user can read messages sent by users clients.
- ▷ Each user can start a Chat 1-to-1 with another user.
- ▷ Users can leave the chat.
- ▷ Client crash are detected.

A.1 Remote Actors and Actor Systems

A.1.1 Setting ActorSystems

Our project is composed of two actor systems:

ChatSystem listens on port 2552 and starts one actor, the remoteActor that provides a service for the chat/chatroom.

ClientChatSystem listens on port 2553 and starts one actor, the Communicator that represents one generic user.

The two actor systems use different configuration, which is where the listen port is defined. The ChatSystem uses chat.conf and the ClientChatSystem uses remotelookup.conf. The two configuration files define on which port to listen.

The configuration files also import the common.conf. This enables the remoting by installing the RemoteActorRefProvider and chooses the default remote transport.

```
1 //common.conf
2
3 akka {
4 actor { provider = "akka.remote.RemoteActorRefProvider" }
5 remote { netty.tcp { hostname = "127.0.0.1" } }
6 }
```

There are two things that are needed for remote deploying:

- ▷ *Add host name*

  The machine you want to run the actor system on; this host name is exactly what is passed to remote systems in order to identify this system and consequently used for connecting back to this system if need be, hence set it to a reachable IP address or resolvable name in case you want to communicate across the network

- ▷ *Change provider*

  From akka.actor.LocalActorRefProvider to akka.remote.RemoteActorRefProvider;

A.1.2 Lookup remote actors

The Communicator takes a String path as constructor parameter. This is the full path, including the remote address of the chat/chatroom service. The general pattern used to find an actor on a remote node is the following:

- ▷ akka.<protocol>://<actorsystemname>@<hostname>:<port>/<actor path>.

The actor system name of the path matches the remote system's name, as do IP and port number.Top-level actors are always created below the "/user" guardian, which supervises them. In our case it is:

- ▷ akka.tcp://ChatSystem@127.0.0.1:2552/user/remoteActor

The communicator first sends an Identify message to the actor selection of the path. The remote actor will reply with ActorIdentity containing its ActorRef. Identify is a built-in message that all Actors will understand and automatically reply to with a ActorIdentity.If the identification fails it will be retried after the scheduled timeout by the Communicator.Once it has the ActorRef of the remote service it can watch it. The remote system might be shutdown and later started up again, then Terminated is received on the watching side and it can retry the identification to establish a connection to the new remote system.

A.1.3 Communicator and RemoteActor

The Communicator is the actor created for handling the communication between the user and all the other users in the chatroom. The communication between users relies into a communication between the communicator and the remote actor on the server machine. The communicator has also another job: create subactors, also known as subCommunicator, to which delegate the communication between one user and another in case of chat1to1.

A.1.3.1 SubCommunicators and SubSevers To istanciate a new chat with a given user,identified by the name provided after the */query* message written in the inputTextArea, the communicator which represents,let's say *client x*,creates a subCommunicator. If the user specified is present, let's call it *user y*, the subCommunicator that has been just created receives the actorRef of the subActor created by the server for the goal of handling the communication between these two clients.This lets *user x* talks with *user y*. At this time we say that a "channel" between *user x* and *user y* has been created.(Of course, on the other side, when the other communicator receives the request for the chat1to1 it instanciates a new child, a subCommunicator, which job is to handle the communication *with user* x.)

A.2 Behaviour

We show two scenario to clarify how the system works.

A.2.1 Login, Talk, Logout

1. Client sends a LoginRequest to the Server
2. Server handles the LoginRequest (ack/reject)
3. Client sends a message (to the Server) in the chatroom
4. Server replays to all the available Clients
5. Client Sends a LogoutRequest (disconnect/exit)
6. Server handles the LogoutRequest

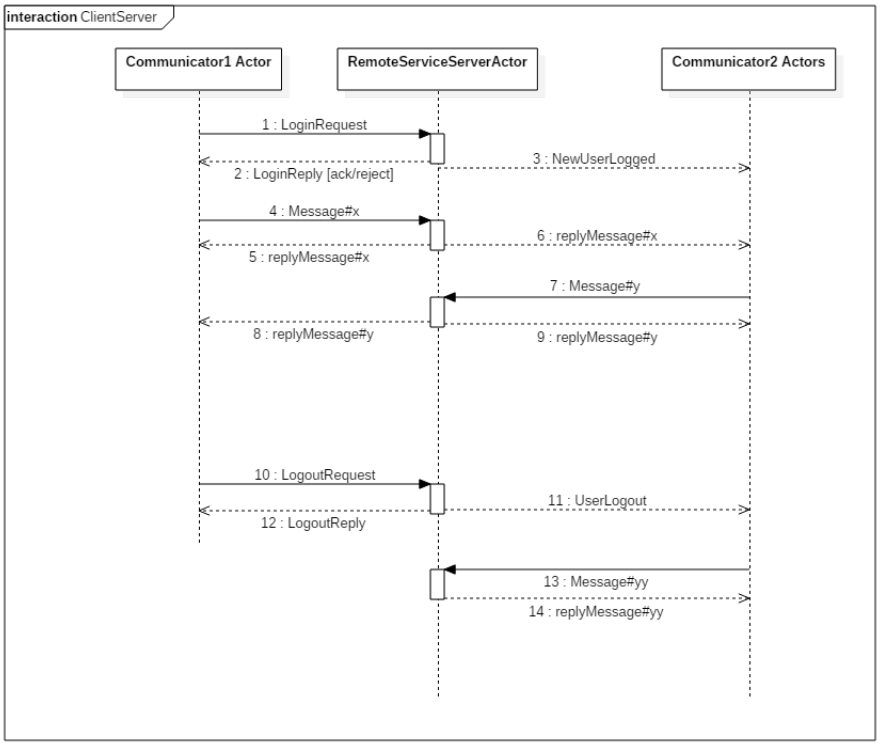

Figure 2: Exchange messages during a chatroom.

A.2.2 Chat 1to1

Using the keywords /query nickname it's possible to start a chat 1to1 with another Client Client.

1. Communicator1 creates a SubCommunicator and sends a RequestChat1to1 to the server.
2. Server creates a SubServer that will handle the communication and sends a Request to the Communicator2.
3. Communicator2 creates a SubCommunicator to support the Chat .
4. User1 and User2 can chat in a private room.

5. When a Client Logout all the SubActor related to that Client are killed.

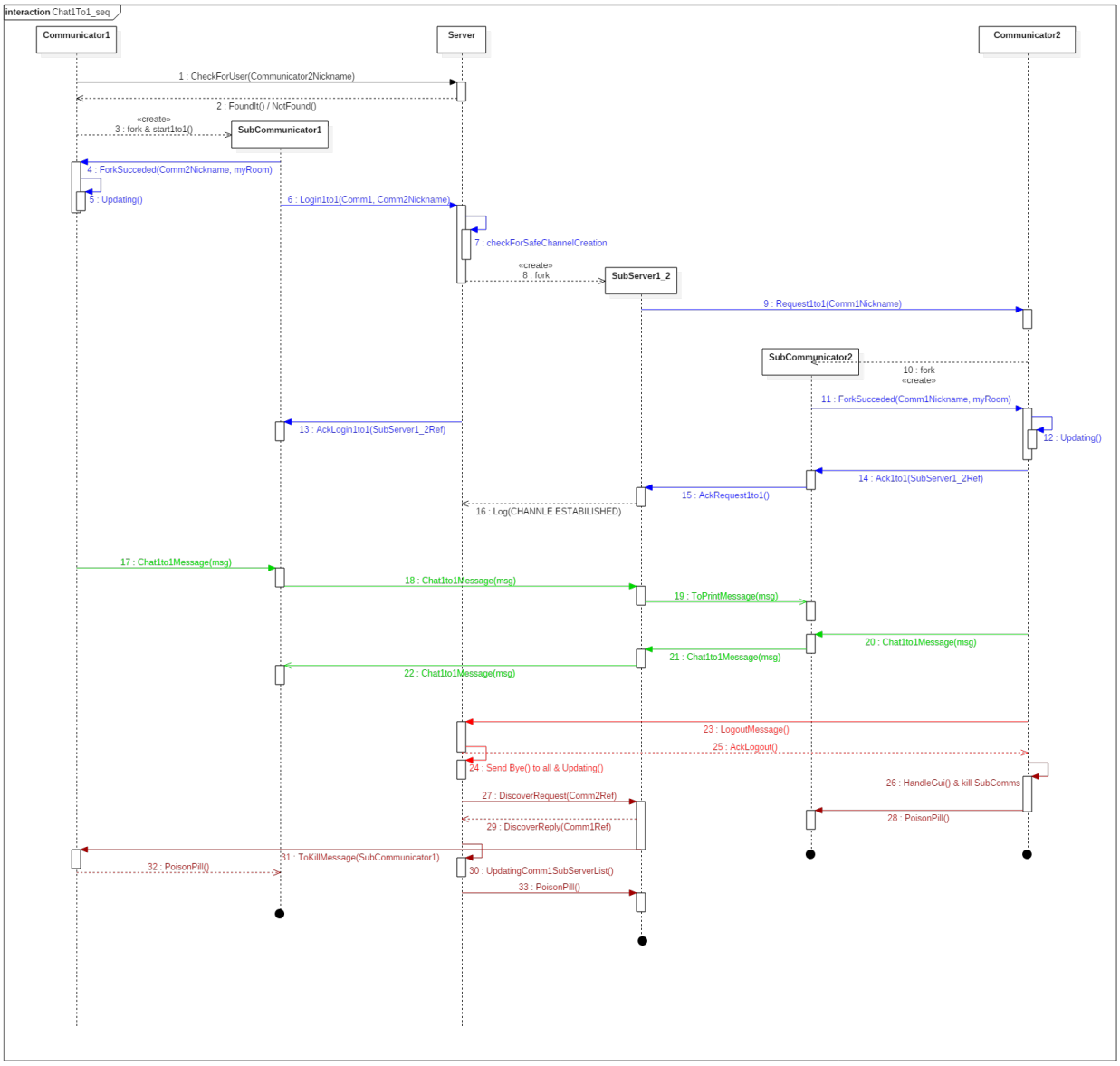

Figure 3: Exchange messages during private chat 1-to-1.

A.3 Messages

Each component in the our project has a set of messages to handle.

A.3.1 Server

The Server side evolves with the following messages:

- ▷ Login Message: (from a Communicator)

- It is sent by a Client that want to join the chatroom. The Server will store the information related to the Client.

▷ ChatMessage: (from a Communicator)

- The Server forwards this Message to all the communicators online.

▷ CheckForUser: (from the Communicator that wants start the chat 1to1)

- When a Client wants to start a communication 1to1 (before to create a SubCommunicator) sends a request to the server to see if the user is logged. The server replies with an ack or a reject.

▷ Login1to1: (from the SubCommunicator that wants to start (and handle) a chat 1to1)

- When an ack is received from the CheckForUser request, a SubCommunicator is created. The new SubCommunicator sends a request to the Server to start the chat 1to1. The server creates a SubServer providing the informations related to che two Communicators and SubCommunicator.

▷ LogoutMessage: (from a Communicator)

- The server updates its data structures and notify all the other Client that a logout has occurred.

▷ DiscoverReply: (from a SubServer)

- This is a service message used to update the data structures that store information related to the channels (SubServers) of a Client that goes offline.

▷ Pong: (from a Communicator)

- This is a service message implemented to detect a Client crash.

▷ LogMessage: (from a SubServer)

- This is a service message that a SubServer uses to write on the Log Area in the server.

A.3.2 Communicator

The Communicator) handle the following messages:

▷ LoginMessage: (from the GUI)

- The Communicator will send a connection request to the Server.

▷ RejectLogin: (from the Server)

- Server notifies that the login request has been rejected (g.e. username is taken)

▷ AckLogin: (from the Server)

- Server accepts the login request. The connection is established.

▷ UpdateUserList: (from the Server)

- Client has to update the List of Available Users.

▷ TrunkedUsername: (from the Server)

- Connection succeded, but the username is trunked because longer than the max length.

- ▷ Query: (from the GUI)
  - When a Client types "/query username", the Communicator sends a Login1to1 request to the server.
- ▷ NotFound: (from the Server)
  - If the username specified in the Login1to1 request is not online the connection 1to1 is not created.
- ▷ FoundIt: (from the Server)
  - If the username specified in the Login1to1 request is found the Communicator (that is requiring the chat 1to1) creates the SubCommunicator.
- ▷ ForkSuccessed: (from the SubCommunicator)
  - When a SubCommunicator is created it notifies the Communicator that the creation has been finalized.
- ▷ Request1to1: (from the SubServer)
  - The Communicator (that is requiring the chat1to1) receive the Chat1to1 request from a SubServer, and creates the SubCommunicator to support it.
- ▷ RoutingMessage: (from the GUI)
  - This is a service message to handle the correct textarea associated to the message.
- ▷ ChatMessage: (from the GUI)
  - The Communicator forward the message to the Server.
- ▷ ToPrintMessage: (from the Server)
  - The Communicator will write the message in the chatroom.
- ▷ ToPrintWarningMessage: (from the GUI)
  - This is a service message of warning (g.e. the user tried to start a chat with itself).
- ▷ LogoutMessage: (from the GUI)
  - The Communicator notifies the Server that it is going offline.
- ▷ ToKillMessage: (from the Server)
  - The Server notifies the Communicator to terminate the SubCommunicator related to a chat1to1 because the Client is going offline.
- ▷ AckLogout: (from the Server)
  - The Server acknowledges the logout.
- ▷ Bye: (from the Server)
  - The Server acknowledges that a user is now offline (a message will be written in the chatroom).
- ▷ Ping: (from the Server)
  - This is a service message to detect crashes.
- ▷ Terminated: (from the ServerSystem)
  - This is a System message sent when the Server goes down. The Client will retry to estabilish the connection.
- ▷ ReceiveTimeout: (from the System)
  - Is used to check if the timeout is elapsed when a message is sent to the Server.

A.3.3 SubCommunicator

The SubCommunicator evolves handling the following messages:

- ▷ Start1to1: (from the Communicator (client1))
  - The client which wants to start the communication with another client sends a request to the Server.
- ▷ AckLogin1to1: (from the Server)
  - The client that starts the communication receive the acknowledgment for the channel, and the ActorRef of the SubServer that handles the chat1to1.
- ▷ AckComm1to1: (from the Communicator (client2))
  - The client that receives the request sends the ack to the SubServer (so that the SubServer creates the channel).
- ▷ Chat1to1Message: (from a Communicator)
  - It forwards the message written in the textArea to the SubServer.
- ▷ ToPrint1to1Message: (from the SubServer)
  - It receives the message to print in the chat.
- ▷ ReceiveTimeout: (from the System)
  - It's used to check if the timeout is elapsed when a message is sent to the Server.
- ▷ PosisonPill: (from the Communicator)
  - A system message to kill its child actor (the SubCommunicator).

A.3.4 SubServer

The SubServer evolves handling the following messages:

- ▷ AckRequest1to1: (from the SubCommunicator of the Client that receive the chat1to1 request)
  - It stores the ActorRef related to the SubCommunicator that has sent the message.
- ▷ Chat1to1Message: (from a SubCommunicator)
  - It contains the message for the other user.
- ▷ DiscoverRequest: (from the Server)
  - It returns to the Server the ActorRef of the SubCommunicator that was chatting with a client that is now offline.
- ▷ PoisonPill: (from the Server)
  - A system message to kill the the SubServer.