Distributed Programming with Types and OTP Design Principles

Jiansen HE

Aug 13, 2012

1 Project Motivation

In 1996, the Erlang/OTP (Open Telecom Platform) library[**?**] was released for writing Erlang code using the Actor model[**?**] and principles derived from 10 years successful experience. Those principles are later known as OTP principles. Although Erlang is an untyped language, with the help of testing tools and OTP principles, successful OTP applications has achieved a high reliability.[**?**]

Since May 2011, the Akka team has been working on porting OTP principles into Scala, a typed language that mixes functional and objected-oriented programming paradigms. The Akka library[**?**, **?**] demonstrated the possibility of using OTP principles in a typed setting; however, messages to Akka actors are dynamically typed and some Akka APIs may deserve more specific types. It remains to be seen whether a strongly typed platform helps the construction of reliable distributed applications.

2 Project Objectives

The aim of this project is to *study how types and OTP principles help the construction of distributed systems*. This project intends to explore factors that encourage programmers to employ types and good programming principles. To this end, following iterative and incremental tasks are required.

1. To identify some medium-sized applications implemented in Erlang or Akka using OTP principles. Those examples will be served as references to evaluate frameworks that support distributed programming. Candidate examples shall cover a wide range of aspects in distributed programming, including but not limited to (a) transmitting messages and potentially computation closures, (b) modular composition of distributed components, (c) default and extensible failure recovery mechanism, (d) eventually consistency model for shared states, and (e) dynamic topology and hot code swapping.

2. To implement a library that supports OTP design principles in a strongly typed setting. Basic components of the library may be build on top of Akka[**?**] in the case that repeated implementation could be avoided. Comparing to the Akka library, the new library shall be able to prevent more forms of type errors.

3. To re-implement identified applications using the library developed in task 2. The primitive purpose of this task is to be aware of prominent features and limitations of existing and the proposed library. As a research which contains certain level of overlaps with other existing and evolving frameworks, this research should distinguish itself from others by devised solutions to some problems which are difficult to be solved by alternatives.

4. To evaluate how the proposed library meets the demands of distributed programming. At this stage, experience gained in task 3 will be systematically analysed. To do so, a novel framework for comparing distributed comparing platforms is likely required. The evaluation phase does not suggest the end of this project. The library and its evaluation will be published to programming communities for feedbacks before we conclude.

3 Progress Overview

At the time of this writing, key components of a typed actor library, TAkka, have been implemented. The TAkka library provides type-parametrised actor and Akka-style APIs. The TAkka library has been tested against examples ported from Erlang and Akka examples. Experience shows that type-parametrised actor works well with OTP principles and provides clearer communication interface and better guarantee of type safety.

Aside from type-parametrised actor, I have implemented a typed nameserver which maps each type symbol to a value of corresponding type. The typed nameserver is immediately used in the TAkka library for looking up actor references.

4 The TAkka Library

4.1 Type Parametrised Actor

Every TAkka actor takes a type parameter specifying the type of its expecting messages. Only messages of the right type can be sent to TAkka actors. Therefore, receivers do not need to worry about unexpected messages while senders can ensure that messages will be understood and processed, as long as the message is delivered.

```scala
import akka.actor._

class MyAkkaActor extends Actor{
  def receive = {
    case m:String =>
     println("received message: "+m)
  }
}

object AkkaActorTest extends App {
  val system = ActorSystem("MySystem")
  val myActor = system.actorOf(
        Props(new MyAkkaActor),"myactor")
  myActor ! "hello"
  myActor ! 3
 //no compile error

}
```

```scala
import takka.actor._

class MyTAkkaActor extends Actor[String] {
  def typedReceive = {
    case m =>
      println("received message: "+m)
  }
}

object TAkkaActorTest extends App {
  val system = ActorSystem("MySystem")
  val myActor = system.actorOf(
          Props[String](new MyTAkkaActor),"myactor")
  myActor ! "hello"
  //myActor ! 3
  //compile error: type mismatch; found : Int(3)
  // required: String
}
```

Table 1: A Simple Actor in Akka and TAkka

Table-?? shows how to define and use a simple string processing actor in Akka and TAkka. Both actors define a message handler that only intends to processing strings. At line 15, sending a message of non-string type to the string processing actor is prohibited in TAkka but allowed in Akka[1].

4.2 Supervising Type Parametrised Actors

System reliability will be improved if failed components are restarted in time. The OTP Supervision Principle suggests that actors should be organised in a tree structure so that failed actors could be properly restarted by its supervisor. The Akka library enforces all user-created actors to be supervised by a system guardian actor or another user-created actor. The TAkka library derives the above good design. To be embedded into a supervision tree, every TAkka actor, Actor[M], defines two message handlers: one for messages of type M and another for messages for supervision purpose.

4.3 Wadler's Type Pollution Problem

The Walder's type pollution problem is first found in layered systems, where each layer provides services to the layer immediately above it, and implements those services using services in the layer immediately below it. If a

[1]In versions prior to Akka 1.3.1, receiving a message that does not match any pattern will cause an exception. Since Akka 2.0, unhandled messages are sent to an event stream. Both strategies are different from the Erlang design where unhandled messages are remain in the actor's message box for later processing.

layer is implemented using an actor, then the actor will receive both messages form the layer above and the layer below via its only channel. This means that the higher layer could send a message that is not expected from it, so as the lower layer.

Generally speaking, type pollution may occur when an actor expects messages of different types from different users. Another example of type pollution is implementing the controller in the MVC model using an actor.

One solution to the type pollution problem is using type parametrised actors and sub-typing. Let a layer has type Actor[T], denoting that it is an actor that expects message of type T. With sub-typing, one could publish the layer as Actor[A] to one user (e.g. the layer above or the Model) while publish the same layer as Actor[B] to another user (e.g. the layer below or the Viewer). The system could check if both A and B are subtypes of T.

4.4 Code Evolution

Same as in the Akka design, an actor could hot swap its message handler by calling the *becomes* method of its context, the actor's view of its outside world. Different from the Akka design, code evolution in TAkka has a restricted direction, that is, an actor must be evolve to a version that is able to handle the same amount or more number of patterns. The decision is made so that a service published to users would not become unavailable later. The related code in the ActorContext trait in presented below. Notice that both static and dynamic type checking are used in the implementation.

```
1 trait ActorContext[M] {
2  implicit var mt:Manifest[_] = manifest[M]
3
4  def become[SupM >: M](behavior: SupM => Unit,
       possibleHamfulHandler:akka.actor.PossiblyHarmful =>
       Unit)(implicit smt:Manifest[SupM]):ActorRef[SupM] = {
5      if (!(smt >:> mt))
6        throw BehaviorUpdateException(smt, mt)
7
8      mt = smt
9      // other code
10  }
11 }
```

Table 2: Code Evolution in TAkka

4.5 A comparison with Akka Typed Actor

In the Akka library, there is a special class called TypedActor, which contains an internal actor and could be supervised. Users of typed actor invoke

a service by calling a method instead of sending messages. The typed actor prevents some type errors but has some limitations. For one thing, typed actor does not permit code evolution. For the other, typed actor cannot solve the type pollution problem.

4.6 Typed Nameserver

Traditional nameserver associate each name with a value. Users of a traditional name server can hardly tell the type of request resource. To overcome the above limitation, we designed a typed nameserver which map each typed name to a value of corresponding type, and to look up a value by giving a typed name.

4.6.1 Scala Manifest as Type Descriptor

Comparing type information among distributed components requires a notion of type descriptor that is a first class value. In Scala, a descriptor for type T could be constructed as manifest[T], which is a value of type Manifest[T]. Furthermore, with *implicit parameter* (e.g. line 4 below) and *context bound* (line 10 below), library designer could present users concise APIs. For example, in a Scala interpreter, one could obtain following results:

```
1  scala> manifest[Int => String]
2  res0: Manifest[Int => String] = scala.Function1[Int,
       java.lang.String]
3
4  scala> def name[T](implicit m: scala.reflect.Manifest[T]):String
       = m.toString
5  name: [T](implicit m: scala.reflect.Manifest[T])String
6
7  scala> name[Int => String]
8  res1: String = scala.Function1[Int, java.lang.String]
9
10 scala> def name2[T:Manifest]:String = {manifest[T].toString}
11 name2: [T](implicit evidence$1: Manifest[T])String
12
13 scala> name2[Int => String]
14 res2: String = scala.Function1[Int, java.lang.String]
```

4.6.2 Typed Name and Typed NameServer

A typed name, TSymbol, is a unique string symbol shipped with a manifest. In TAkka, TSymbol is defined as follows:

```
1  case class TSymbol[T](val symbol:Symbol)(implicit val
       t:Manifest[T]) {
2    override def hashCode():Int = symbol.hashCode() }
```

The three operations provided by typed nameserver are:

- set[T:Manifest](name:TSymbol[T], value:T):Unit

  Register a typed name with a value of corresponding type. A 'NamesHasBeenRegisteredException' will be thrown if the name has been used by the name server.

- unset[T](name:TSymbol[T]):Unit

  Cancel the entry *name* if (i) the symbol representation of *name* is registered, and (ii) the manifest of *name* (i.e. the intention type) is a super type of the registered type.

- get[T] (name: TSymbol[T]): Option[T]

  Return Some(v:T) if (i) *name* is associated with a value and (ii) T is a supertype of the registered type; otherwise return None.

Notice that *unset* and *get* operations permit polymorphism. For this reason, the hashcode of TSymbol does not take manifest into account. The typed nameserver also prevents users from registering two names with the same symbol but different manifests, in which case if one is a supertype of another, the return value of *get* will be non-deterministic.

5 Performance Evaluation

The three dimensions of library evaluation are (i) scalability (ii) reliability or availability (iii) reasonable efficient. Standard methodology for evaluating performance has been found in literature. Evaluation methodology for the other two aspects, unfortunately, requires more studies.

The performance evaluation methodology presented in this section is derived from Hennessy and Patterson's book [**?**], which provides quantitative approaches used in the design and analysis of hardware development. I found the presented approach is also applicable to software performance evaluation.

In hardware evaluation, benchmark suites are used to measure performance. Unfortunately, by the time of this writing, there is no standard benchmark suite for Akka or other distributed programming frameworks. Therefore, a selection of benchmarking examples is required for assessing the overall performance of the TAkka library. To avoid unintentional optimization in TAkka implementation, benchmarks should be selected from existing applications written in other platforms and modifications should be made at the minimal level.

Once the suite of benchmarks is decided, we could use geometric mean of all performance measures to summarize the overall performance. In addition, to assess the variability of the chosen suite, the geometric standard deviations could be used as an indicator to ́decide whether the mean is

likely to be a a good predictor: Hennessy and Patterson[**?**] further suggest employing statistic tools to analysis results.

6 Future Work

The work for the next year will be divided into three parts. Firstly, I will continue the work of porting existing examples to understand the demands of real distributed applications. With experience gained in previous work, I should be able to attempt bigger examples such as Play2 and Riak. Secondly, evaluation components for other aspects of distributed programming should be formalised. Those aspects include scalability and reliability. Finally, small examples that demonstrate the advantage of TAkka library will be constructed.

A Examples

This appendix lists examples ported from other projects. When applicable, for each example, the number of modified lines and the total number of lines in the Akka version are given in the brackets following the example name.

A.1 Examples Ported from Quviq Training Examples

Following two examples are ported from examples of a Quviq course. Quviq is a quickcheck tool for Erlang programs. The Erlang version of following two examples are commercial products that shall not be published. To avoid unintentional leak of their Erlang design. Akka and TAkka implementation for those examples are not published to public repositories.

ATM simulator (199/1148): A GUI simulator of a bank ATM terminal.

Elevator Controller (172/1850): A system that monitors and schedules a number of elevators.

A.2 Examples from Akka documentation

Following examples are selected to check that the TAkka library could provide Akka equivalent services. Those examples are:

Ping Pong (13/67): A simple message passing application.

Dining Philosophers (23/189): A application that simulate the dining philosophers problem using Finite State Machine (FSM) model.

Distributed Calculator (43/250): An application that examines distribution and hot code swap.

Fault Tolerance (69/274): An application that demonstrate how system responses to component failures.

A.3 Other Examples from Open Source Akka Applications

Following examples are selected to the TAkka library meets the demand of real applications building on top of Akka. Those examples are:

Barber Shop (104 / 754): The implementation of famous Barber Shop example in Akka.

EnMAS (213/1916): An environment and simulation framework for multi-agent and team-based artificial intelligence research

Socko Web Server: “ lightweight and embeddable Scala web server that can serve static files and support RESTful APIs to our business logic implemented in Akka.”

Gatling: A stress testing tool. Most Gatling components are implemented using its own DSL. Therefore, changes are only made to 27 core files, where 111 out of 1635 lines of code are modified.

B The TAkka Library

An *actor* is a lightweight process that responses to *message*s according to its *behavior*. In a fault tolerant system, related actors are supervised by their *supervisor*s and form a tree structure. To better benefit from the functional nature of the actor model, the Akka framework [**?**, **?**] decides to shield actors from the outside using *actor reference*s, which can be freely passed across applications and distributed nodes.

Following core operations are required by the actor model:

- **create**: create an actor and return an actor reference to the actor created;
- **send**: send a message to an actor via its actor reference;
- **become**: update the *behavior* of an actor.

Following sub-sections will explain how the TAkka library supports the actor model described above. APIs of this library is largely influenced by the Akka framework [**?**]. Key differences between this library and Akka will be explained in related sub-sections. Full TAkka APIs are given at `http://homepages.inf.ed.ac.uk/s1024484/takka/`.

B.1 Actor

```
1 package actor
2 abstract class Actor[Msg:Manifest]{
3   protected def typedReceive:PartialFunction[Msg, Unit]
4   protected def possiblyHarmfulHandler:akka.actor.PossiblyHarmful
        => Unit
5
6   protected[actor] implicit val typedContext:ActorContext[Msg]
7   implicit final val typedSelf:ActorRef[Msg]
8   final lazy val typedRemoteSelf:ActorRef[Msg]
9
10  def preStart():Unit
11  def postStop():Unit
12  def preRestart(reason:Throwable, message:Option[Msg]):Unit
13  def postRestart(reason:Throwable):Unit
14 }
```

The **Actor** class provides essential constructs for defining an actor. The *behavior* of an actor is defined by the *typedReceive* method and the *possiblyHarmfulHandler* method. Users must implement *typedReceive* which will be used as the handler for receiving messages of type **Msg**. General users do not have to override the default implementation of *possiblyHarmfulHandler*, which reacts to system messages. This library uses the same

system messages as the akka library [**?**]. Selected system messages will be explained in related sections. In the case that there is an intersection of **M** and **PossiblyHarmful**, message of the intersect type will be handled by *typedReceive*.

An **Actor** instance has fields representing its actor reference(§**??**) and actor context(§**??**). The *typedSelf* field, initialised at the same time as the actor, is an actor reference that enables local communication to this actor. Using the value of *typedSelf* at a remote site will usually fail to delivery messages, unless the actor is deployed as a remote actor (§**??**). On the other side, value of the *typedRemoteSelf*, if it exists, can be passed around local and remote sites and behave as expected. Notice that *typedRemoteSelf* is a lazy initialised field. When *typedRemoteSelf* is first called, a **NotRemoteSystemException** may raise if the actor is not located in an actor system that supports remote communication. The use of actor context will be explained in §**??**.

Moreover, users can specify procedures which will be called (i) before the actor is started; (ii) after the actor is terminated; (iii) before the actor is restarted due to an Error or Exception raised when handling a particular message; and (iv) after the actor is successfully restarted.

Finally, unlike the akka design, this library deprecates the *sender* field which represents the sender of the last receiving message. The *sender* field is deprecated for several reasons. Firstly, the type of the *sender* field should be a super type of the union of types of all possible senders expecting a reply. Therefore, it is still possible to reply a sender with messages of wrong types. Secondly, without due care, using the shared variable *sender* may introduce bugs in concurrent processes. Thirdly, experience shows that using *sender* contributes to the difficulty of tracing dataflow in debugging process. Lastly, to reduce side-effects of message processing, the library designer argues that resources that may effect the message processing, including the sender of a synchronous request, should be part of the message.

B.2 Actor Reference

```
1 @serializable
2 @SerialVersionUID("ActorRef-v-0-1")
3 abstract class ActorRef[-Msg : Manifest]{
4   def ![M](message: Msg):Unit
5   final def tell(msg: Msg): Unit
6
7   def isTerminated : Boolean
8   def path : akka.actor.ActorPath
9   final def compareTo (other: ActorRef[_]):Int
10 }
```

As mentioned earlier, same as in the akka library, actors are shielded from the outside using actor references. Users send messages to an actor via

calling the *tell* method or the ! method of corresponding actor references.

Same as in the akka API, the *isTerminated* test of an actor reference checks the liveness of its representing actor. The *isTerminated* test returns true only if the representing actor is completely shut down by its actor system; temporary actor failure will not change the test result from true because actor is always be supervised and could be restarted after the failure.

In addition, users could enquiry and compare the paths of actor references. Actor path in this library is not type parametrised because it is not directly related to message sending.

The two serialization annotations before the class declaration ensures that an actor reference could be serialized and deserialized consistently.

B.3 Actor Reference Configuration

```
object Props{
  def apply[T, A<:Actor[T]] (implicit arg: ClassManifest[A],
      t:Manifest[T]): Props[T]
  def apply[T:Manifest](actorClass: Class[_ <: Actor[T]]): Props[T]
  def apply[T:Manifest](creator: => Actor[T]): Props[T]
}
case class Props[-T:Manifest] (props: akka.actor.Props)
```

Trying to be consistent with the akka library, this library also provides a Props class, whose instance represents the configuration of actor creation, used in *ActorSystem.actorOf* and *ActorContext.actorOf*. This section will only look at ways of creating an instance of Props. Using a Props to create an actor and actor reference will be explained the the next two sections.

```
class StringActor extends Actor[String] {
  // class implementation
}
```

Suppose an actor based string processor, **StringActor**, is implemented as above, users can then obtain a corresponding Props of **StringActor** using one of the following idioms:

```
  val props = Props[String, StringActor]
  val props = Props[String](StringActorClass)
  //where StringActorClass is a class object for StringActor
  val props = Props[String](new StringActor)
```

Omitting the String parameter in above examples will give an actor creation configuration of type Props[_], whose corresponding actor reference will not accept message of any type.

Though it is more flexible to create a Props by providing a type parameter and an akka Props instance, props object created in this way looses the guarantee of having a correct type parameter captures the type of expecting

messages.

B.4 Actor Context

```
abstract class ActorContext[M:Manifest] {
  val props:Props[M]
  def typedSelf : ActorRef[M]
  implicit def system : ActorSystem

  def receiveTimeout : Option[Duration]
  def resetReceiveTimeout(): Unit
  def setReceiveTimeout (timeout: Duration): Unit

  def actorOf[Msg:Manifest](props:Props[Msg],
      name:String):ActorRef[Msg]
  def actorOf[Msg:Manifest](props:Props[Msg]):ActorRef[Msg]
  def remoteActorOf[Msg:Manifest](props:Props[Msg]):ActorRef[Msg]
  def remoteActorOf[Msg:Manifest](props:Props[Msg],
      name:String):ActorRef[Msg]
  def actorFor[E:Manifest](actorPath: String): ActorRef[E]

  def watch[M](subject: ActorRef[M]): ActorRef[M]
  def unwatch[M](subject: ActorRef[M]): ActorRef[M]

  def become[SupM >: M](behavior: SupM => Unit,
              possibleHamfulHandler:akka.actor.PossiblyHarmful =>
                  Unit)
              (implicit arg:Manifest[SupM]):ActorRef[SupM]
}
```

An actor context provides contextual information for an actor.[?] By accessing an actor context, users could retrieve the props that created the actor, the actor reference that pointed to the actor, the actor system that the actor located in, and the timeout of receiving the first message. If a timeout is set for the first message, then users should handle the **ReceiveTimeout** message inside the *possiblyHarmful* method.

Two main usages of actor context are creating and looking for a child actor. Actor is created by calling the *actorOf* method or the *remoteActorOf* method. The actor reference returned by the *actorOf* method may not contain a global aware path. The actor reference returned by the *remoteActorOf* method, on the other hand, can be used at a remote site if the actor system supports remote communication. Actor created by either methods is supervised by the actor captured by this actor context. Like in akka, an actor will be given a name at the time of its creation. If the user does not provide a name, a system generated name will be provided; if the user provides a name which has been used by a sibling of the creating actor, an **InvalidActorNameException** will be thrown. Once a valid name is associated with the created actor, a new actor path is generated with the

name appending to the actor captured by the actor context. In another word, a child actor is created. The *actorFor* method returns an actor reference associated with the provided actor path and the expecting message type. The returned actor reference will point to a **DeadLetters**, which is the sink of messages to dead actors, in the case that the actor path is not associated with an actor, or the actor at the actor path has an incompatible type parameter to the provided type parameter.

Like in OTP[**?**] and akka[**?**], an actor could linked to another actor, watching for its death message, or unlinked from a linked actor. The death message, **Terminated (actor: ActorRef)**, is handled by the *possiblyHarmful* method of an actor.

Unlike in the akka library, the actor context in this library does not manage information about the current processing message such as its sender. Any information that the message receiver needs to consider should be part of the message itself.

Finally, this library provides code swap on the actor behaviour using *become* in the same fashion as in akka. The current API is different from the akka version in three aspects. Firstly, the *become* method in this library requires a handler for system messages. Fortunately, in most cases, users use actor context inside an actor definition; therefore, users could pass the system message handler retrieved from the actor definition, if the handler does not need to be changed. Secondly, the behavior parameter has type **SupM** $\Rightarrow$ **Unit**, where **SupM** is usually not **Any**. Thirdly, this library deprecated the *unbecome* method to avoid potential problems related to type evolution.

B.5 Actor System

```
object ActorSystem {
  def apply():ActorSystem
  def apply(name: String): ActorSystem
  def apply(name: String, config: Config): ActorSystem
}

abstract class ActorSystem {
  // members with type parameter
  def actorOf[Msg:Manifest](props:Props[Msg]):ActorRef[Msg]
  def actorOf[Msg:Manifest](props:Props[Msg], name:String):
  def remoteActorOf[Msg:Manifest](props:Props[Msg]):ActorRef[Msg]
  def remoteActorOf[Msg:Manifest](props:Props[Msg],
      name:String):ActorRef[Msg]

  def actorFor[Msg:Manifest](actorPath: String): ActorRef[Msg]
  def actorFor[Msg:Manifest](actorPath: akka.actor.ActorPath):
      ActorRef[Msg]

  def deadLetters : ActorRef[Any]
```

```
18
19  // members that same as akka version
20  def awaitTermination (): Unit
21  def awaitTermination (timeout: Duration): Unit
22  def eventStream : akka.event.EventStream
23  def extension [T <: Extension] (ext: ExtensionId[T]): T
24  def isTerminated : Boolean
25  def log : LoggingAdapter
26  def logConfiguration (): Unit
27  def name : String
28  def registerExtension [T <: Extension] (ext: ExtensionId[T]): T
29  def registerOnTermination (code: Runnable): Unit
30  def registerOnTermination [T] (code: => T): Unit
31  def scheduler : akka.actor.Scheduler
32  def settings : akka.actor.ActorSystem.Settings
33  def shutdown (): Unit
34  override def toString():String
35  def uptime : Long = system.uptime
36
37  // addtional members for constructing remote ActorRef
38  def isLocalSystem():Boolean
39
40  @throws(classOf[NotRemoteSystemException])
41  def host:String
42
43  @throws(classOf[NotRemoteSystemException])
44  def port:Int
45 }
46
47 case class NotRemoteSystemException(system:ActorSystem) extends
      Exception("ActorSystem: "+system.name+" does not support
      remoting")
```

An actor system manages resources to run its containing actors. Same as the akka library, an actor system is initialised by calling one of the *ActorSystem.apply* methods. Users could optionally provide a name and a configuration when creating an actor system, otherwise a default name and a default configuration will be provided. Actor system created with the default configuration can only provide actor references for local communication. In a user defined configuration, remote communication and other functionalities may be enabled. For more information about the **com.typesafe.config.Config** object and actor system configuration, users may refer to the akka API[**?**] and the akka documentation[**?**]. A later version of the type-parametrised actor library may provide a convenient API for writing actor system configuration or enable remote communication by default.

§**??** explains how to created an actor supervised by another actor. To construct a complete supervision tree inside an actor system, a top-level actor needs to be created. The akka library [**?**] provides an guardian ac-

tor, *user*, for all user created top-level actors. This library follows the same design and put actors created using *ActorSystem.actorOf* and *ActorSystem.remoteActorOf* at the next level of the *user* actor.

This library also provides a typed version of *deadLetters*, which is the receiver of messages to stopped or non-exist actors. The last two methods in the **ActorSystem** class are implemented to help constructing actor references that can be used at remote sites. The isLocalSystem returns false if the actor system is registered to a host name and port number, or returns true otherwise. The rest methods have the same functionality as their corresponds in the akka library[**?**].

B.6 FSM

```
1 object FSM {
2   case class CurrentState[S, E](fsmRef: ActorRef[E], state: S)
3   case class Transition[S, E](fsmRef: ActorRef[E], from: S, to: S)
4   case class SubscribeTransitionCallBack[E](actorRef: ActorRef[E])
5   case class UnsubscribeTransitionCallBack[E](actorRef: ActorRef[E])
6   case class Timer[E](name: String, msg: E, repeat: Boolean,
        generation: Int)
7                      (implicit system: ActorSystem)
8 }
9
10 trait FSM[S, D, E] extends akka.routing.Listeners {
11   this: Actor[E] =>
12
13   type State = FSM.State[S, D]
14   type StateFunction = scala.PartialFunction[Either[Event[E],
        StateTimeout.type], State]
15   type EventFunction = scala.PartialFunction[Event[E], State]
16   type TimeoutFunction = scala.PartialFunction[StateTimeout.type,
        State]
17
18   protected[actor] def setTimer(name: String, msg: E, timeout:
        Duration, repeat: Boolean): State
19   protected final def when(stateName: S, stateTimeout: Duration =
        null)(eventFunction: EventFunction): Unit
20   protected final def whenStateTimeout(stateName: S, stateTimeout:
        Duration = null)(timeoutFunction: => State): Unit
21   case class Event[E](event: E, stateData: D)
22 }
23
24 trait LoggingFSM[S, D, E] extends FSM[S, D, E]
```

Whenever possible, the library designer suggests to avoid using mutable actor state so that the actor behavior will be more functional and more likely to be effect free. In some cases; however, if a stateful actor better captures a problem, the user may consider encode the actor as a finite state machine

(FSM) using the typed **FSM** trait.

Like what has been done in the Actor class, a generic type denoting the type of events is added to the FSM trait comparing to the akka version. To save the space, the above list only gives public members with restricted type. The rest of members have the same signatures as members in the akka version.