AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Informatyki, Elektroniki i Telekomunikacji

Katedra Informatyki

PRACA DYPLOMOWA MAGISTERSKA

TRACING OF LARGE-SCALE ACTOR SYSTEMS

ŚLEDZENIE SYSTEMÓW AKTOROWYCH DUŻEJ SKALI

Autor:	Michał Ciołczyk
Kierunek studiów:	Informatyka
Promotor:	dr hab. inż. Maciej Malawski

Kraków, 2017

OŚWIADCZENIE AUTORA PRACY

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej „sądem koleżeńskim”", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście, samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

. . . . . .

PODPIS

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Maciej Malawski, for his invaluable support, guidance, patience and many ideas that helped me while working on this thesis.

Similarly, I would like to thank Mariusz Wojakowski for many discussions on the proposed tool, his invaluable suggestions and code review.

I would like to express my gratitude to Michał Janczykowski for many discussions as well as permission to use the car traffic simulation as a real application in the evaluation of the library being the subject of this thesis.

I would like to also thank GitHub and Amazon for the opportunity to participate in the GitHub Student Developer Pack program and AWS Educate program, which enabled to access the AWS computing cloud and conduct library performance tests at much reduced costs.

Finally, I would like to thank all my friends and family for their support while I was writing this thesis.

Abstract

Nowadays many systems are being modelled using actors. The nature of these systems varies from purely computational (which are mainly used in scientific computations) to purely business (which are used primarily as backend services in companies).

In the case of actor systems, there are often situations that processing within one of the actors fails. It is rarely due to a failure on the actor itself, as in many cases it is due to the fact that some error occurred in previous actor, due to network errors or some external service outage.

In such cases, it is very difficult to trace down the origin of the failure and fix the mistake. Existing monitoring tools for actor systems only provides ways to collect metrics and statistical information about the actor system execution. The error could be easily detected if we had a history of messages that were passed between actors – a trace of the processing in actor system. While it can be quite easily done if the system was deployed on one machine and the number of messages is relatively small. Unfortunately, the actor systems that are used now are of large scale – they often process a lot of messages (often the numbers exceed thousands per second) and are deployed into a cluster of machines.

In this thesis, the author provides a tool for *tracing* distributed actor systems – *Akka Tracing Tool* – a library that allows users to generate a traces graph during actor system execution. Due to the distributed environment, an efficient data collection mechanism needed to be used. The proposed solution uses the one-way replication technique implemented in one of the popular document databases – CouchDB.

The tool was evaluated in a distributed environment (Amazon Web Services (AWS) computing cloud) on a real application (car traffic simulation). The simulation was run on up to 50 nodes (the largest city). When the library was included in the system, the frames per second (FPS) of the test application suffered 68% to 70% loss on average. The overhead is quite high, but the tests show that despite the FPS loss, the library has still been proven to be scalable with respect to the number of nodes in the actor system and to be user-friendly. Due to these facts, the author expects that by using his tool, finding errors in actor systems would be much less burdensome process, which would enable to speed up the development process of actor systems.

Keywords: actor systems, actor computing model, tracing, monitoring, Akka, scalability.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1. Introduction

This Chapter lays out the main idea of the thesis as well as provides a theoretical background for tracing actor systems.

The Chapter is structured as follows: in Section 1.1 the motivation of tracing actor system is discussed, followed by the problem statement (Section 1.2), the definition of the project goals (Section 1.3) and the layout of the the thesis (Section 1.4).

1.1 Motivation

Nowadays many systems are being modelled using actors. The nature of these systems are varying from purely computational (which are mainly used in scientific computations) to purely business (which are used primarily as backend services in companies).

The examples of services modelled as actor systems contain:

- Serving HTTP requests on servers; these systems often contact external services, databases, etc.
- Simulations, e.g. car traffic simulations, fluid flow simulations.
- Processing of streaming data, which is often the case in many real-time data processing systems.
- Some longer computations that need to take place on the servers, these can be e.g. calculating statistics of system usage, performing some actions which need to query databases which contain lots of data, actions that need to perform many queries to database or external services.

In the case of actor systems, there are often situations that processing on some of the actor fails. It is rarely due to a failure on the actor itself, in many cases it is due to the fact that some error occurred in previous actor, as is shown in Fig. 1.

In the case presented in this figure, there was either a network failure while transmitting the message 2 from actor 2 to actor 3, or some error in processing message 1 in actor 2.

While actor's behaviour is easy to reason about (as actors process messages sequentially), it is very difficult to find such error which originated in faulty message that was processed in two actors before it actually manifested itself as an exception.

This could be done if we had a history of messages that were passed between actors – a trace of the processing in actor system. It can be quite easily done if the system was deployed on one machine and the number of messages is relatively small. Unfortunately, the actor systems that are used now are ones of large scale – they often process a lot of

messages (often the numbers exceed thousands per second) and are deployed into a cluster of machines, not on one.

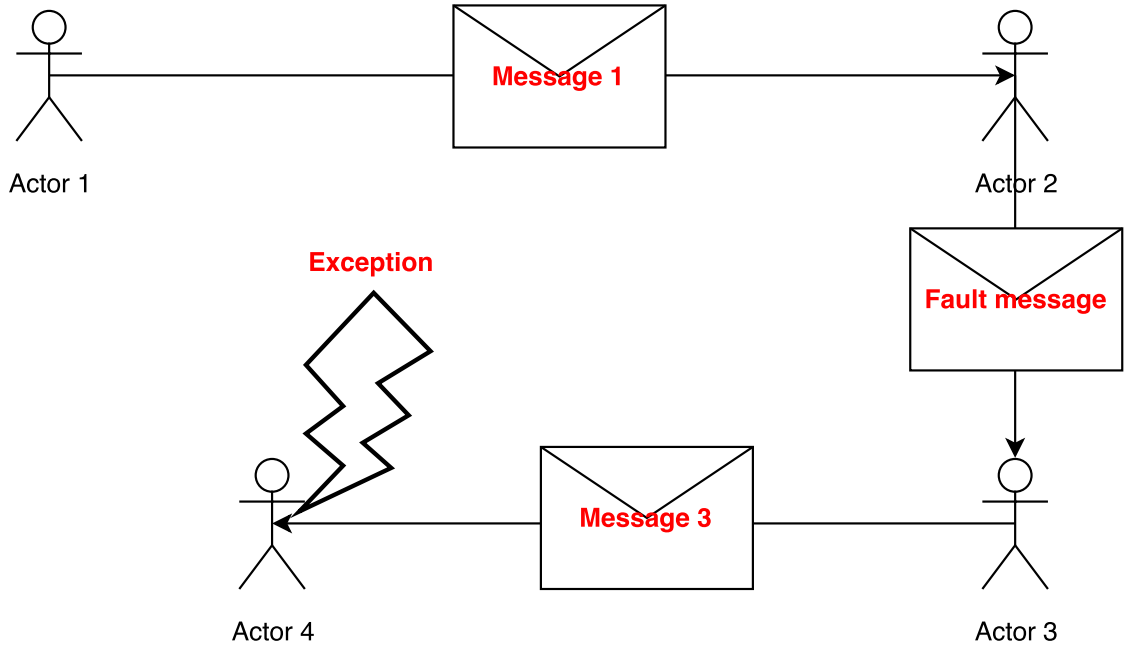

Figure 1: An example of actor processing with failure which is not the fault of the actor itself, but either network error or processing error in some previous actor.

What is more, tracing can help not only in finding errors in actor system. It can be used in many other ways, for example to analyze the performance of the system or to gather statistics about actor execution.

This leads to the conclusion that tracing large scale actor systems in scalable and user-friendly way is an important challenge in development of distributed actor systems.

1.2 Formal definition of the problem

The problem of tracing actor systems this MSc thesis addresses can be formalized as follows:

> Prepare a mechanism which enables to create a directed multi-graph $G = (V, E)$ during the execution of an actor system, such that:
>
> - The vertices are:
>
> – traced actors,
>
> – vertices that represent the exterior of the system that communicates with the actor system being traced.

- The edge (u, v) represents sending message from actor u to actor v.
- Any path in this graph is called *a trace*.

In Fig. 2 there is an example of a traces graph collected using the library.

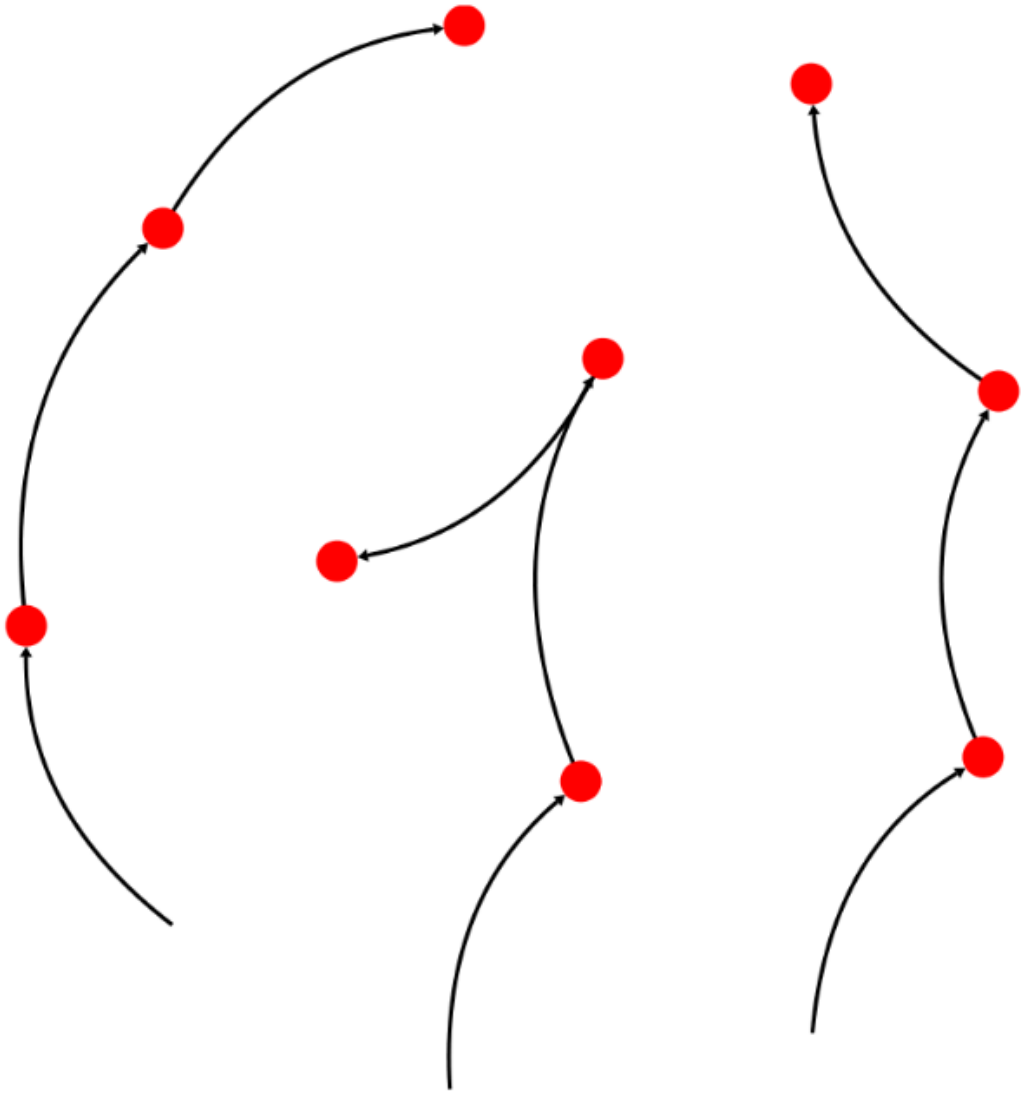

Figure 2: An example of the traces graph collected by the library.

1.3 Goals of the thesis

As the result of this MSc thesis the following goals should be achieved:

- Create a *proof-of-concept* of a library written in Scala language [41, 43] which enables creating the traces graph in a scalable way for applications written in Akka framework [2, 25, 47].
- Prepare an efficient mechanism able to collect data about the traces from a distributed actor system.
- Conduct large-scale tests of the library to measure the impact on the performance of the traced system. These tests should be conducted using a real application on many nodes located on the cloud environment (using AWS [9]), which is often the production environment of the actor systems.

1.4 Thesis layout

The thesis is structured as follows: Chapter 1 describes the main idea of the thesis. Chapter 2 provides background on two main concepts: actor systems and tracing. These topics are used later through the thesis and it is important to understand them to grasp the ideas discussed in this thesis. Chapter 3 provides the throughout description of the proposed solution – *Akka Tracing Tool*, all of its parts and the mechanisms used under the hood. In Chapter 4, the author evaluates the solution, mainly focusing on the performance impact on the traced system, but also comparing the *user-friendliness* of the library to other related libraries. Finally, the Chapter 5 provides the summary of the thesis. It begins with conclusions from the performed tests, then proposes future work that can be done to improve the solution and lays out ideas for further research.

Chapter 2. Background – actor systems and tracing

This Chapter focuses on the background of the thesis, mainly on description of actor systems and tracing. These two areas are essential topics that the thesis is based on and that is why it is necessary to understand them to grasp the main idea of this thesis.

The Chapter is structured as follows: Section 2.1 gives description of actor systems, followed by the discussion on actor systems implementations, mainly written in Scala language [41, 43] and Akka framework [2, 25, 47] (Section 2.2) and tracing, which also consists of the *state of the art* of this problem in context of tracing actor systems (Section 2.3).

2.1 Actor systems

The thesis focuses on actor computing model, a formal model for distributed computations proposed by Hewitt [28] in 1973. This model has been adopted by Akka framework [2, 25, 47]. The model is based on passing messages [26], a paradigm used for example by Message Passing Interface (MPI) [37].

Actor model proposes to decompose the computation into *actors*, the basic units of computation. Actors themselves process messages sequentially. In addition to this, they can:

- send (asynchronously) a finite number of messages to itself or another actors,
- spawn a finite number of actors,
- decide on their behaviour (they can change the behaviour that they use for processing messages).

An actor has *a mailbox*, a queue that stores messages which were received by the actor, but not yet processed. In this way, even if the actor is still processing previous message, new messages will still be processed by it in the future.

In Fig. 3, an example actor system is shown with 3 actors. The messages being sent to actors are put into their mailboxes and then actors process them.

Actors are often deployed to a cluster, often onto a computing cloud. This means that actor systems libraries must include mechanisms for remote actor deployment (as actors can spawn new actors), messages serialization (often it is in the scope of the users of the library, sometimes language mechanisms are used – for example, in Akka, standard Java object serialization is frequently used) and location transparency (often implemented by actor refs; an actor is encapsulated in actor reference and therefore users send messages to actor refs instead of actors themselves, so if the actor is on another machine, message is still sent to the actor ref, an abstraction that knows that the actor is on a remote computer and the user does not need to know it beforehand).

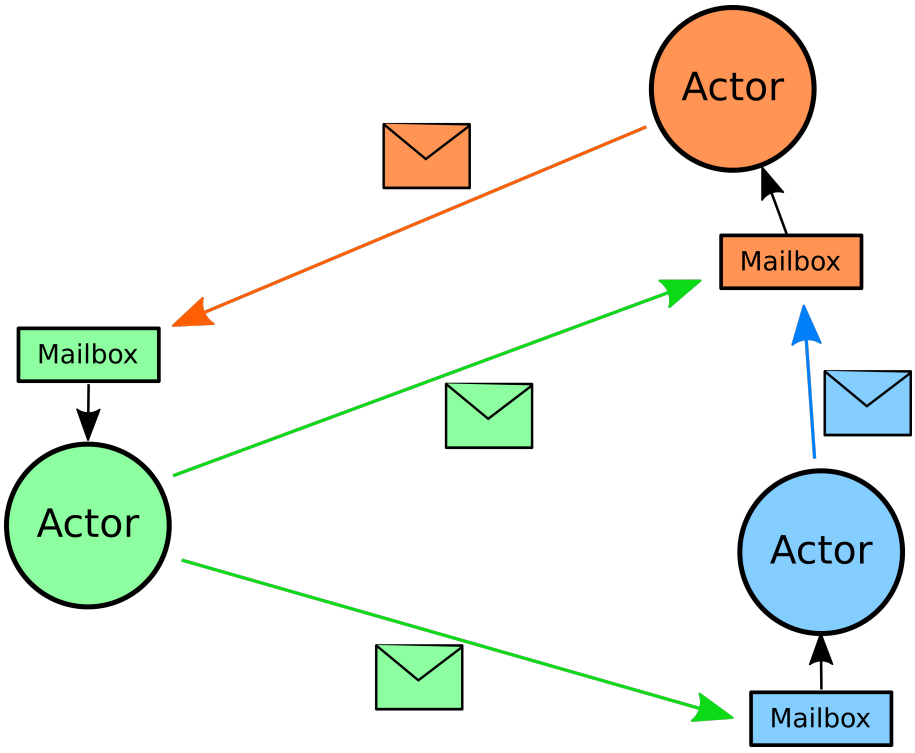

Figure 3: An example of an actor system

Actor systems are commonly used in many large companies (examples include Walmart, Amazon, PayPal). It is due to the fact, that these systems are very fault-tolerant. This tolerance is achieved by the supervision mechanism. Actor systems are designed in what can be called *parent-children* relation – the structure of such systems is often that of a tree (see Fig. 4) – the parents are supervising their children, delegating tasks to them and returning the results to the system's users. In this way, if something happens to the child (which is executing some task) and it terminates, not completing the task it should complete e.g. due to an exception, unavailability of the external system, network connection issues, etc., the supervisor can decide what to do:

- resume the subordinate, keeping its accumulated internal state,
- restart the subordinate, clearing out its accumulated internal state,
- stop the subordinate permanently,
- escalate the failure, thereby failing itself.

In this way, the failure in an actor system can be handled *gracefully*, not causing the termination of the whole system, but only one part of it. This failure handling model is often called *let it fail* – the system does not need to know how to recover from a failure, instead, the subsystem where the failure occurred is restarted and the system operates normally again.

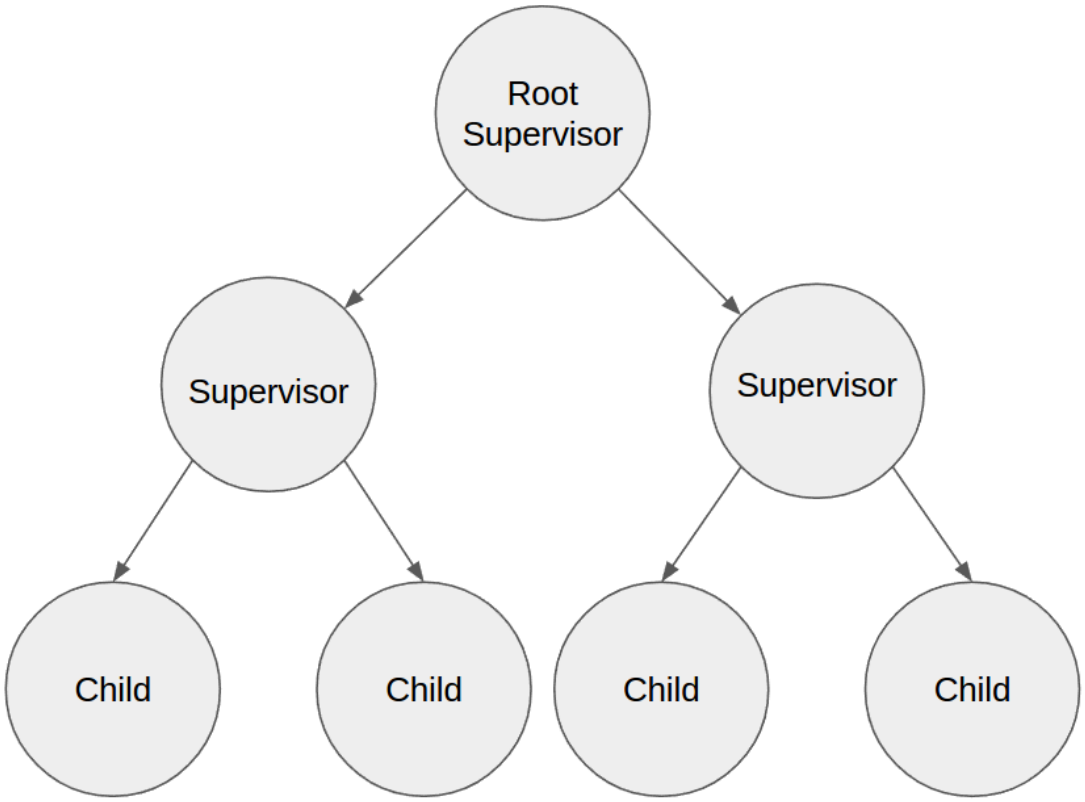

Figure 4: An example of a supervision tree in actor system

2.2 Implementations of actor systems

As mentioned in Section 2.1, the actor computing model was proposed by Hewitt [28] in 1973. It was then thoroughly researched resulting in many publications. Some of the important ones are [1, 10, 13, 24, 27] which provide mathematical deductions about proposed model. It was then popularized when Erlang language was published [7, 18] which used actor model to implement concurrency mechanisms. Erlang was then used at Ericsson with great success to build highly concurrent and reliable telecom systems. To this day, Erlang is widely used in telecommunications systems.

Inspired by Erlang's success, many other libraries and frameworks were created to adopt actor model in other languages. Examples include Pykka [42] written in Python, Akka.NET [3] for .NET technologies and Akka [2,25,47] for Java Virtual Machine (JVM) languages (especially for Scala [41, 43] and Java [23, 30]), which will be the main focus in this Section as the library that is the subject of this thesis was designed to work with Akka.

Akka since the first stable version in 2009 was warmly welcomed by the JVM community, which led then of many usages in production systems of such companies as Walmart, PayPal or Amazon. Akka is also increasingly used in scientific applications, examples contain workflow systems and simulations.

Akka is written in Scala – a programming language which is very commonly used to provide scalable solutions in both high performance computing (HPC) and production systems, with many successful results (for example, Apache Spark [46, 49] is written in Scala language and it is used in many HPC applications and in many companies).

2.3 Related work on tracing actor systems

While the systems created with Akka under the hood used in large companies are fault-tolerant, they still need monitoring so that their availability is not compromised. There are many publications regarding monitoring distributed systems, especially based on message-passing paradigm. Examples of such research include [31, 35, 38–40, 44, 50]. Unfortunately, most of the systems considered in these articles monitor MPI-based systems, which are not based on actor model. Actor systems are a bit more specific than ordinary message-passing based systems and that is why most of the methods considered in these publications cannot be directly used, although of course they can be adopted to the actor systems. What is more, the publications often focus on collecting statistics from the execution of the systems being monitored and do not try to create a graph of messages being passed between processes. This means that the methods discussed in them are not usable to the problem of this thesis.

This thesis proposes *tracing* as a method to not only monitor but also debug or reason about actor systems. This approach is not very popular as there are not many tools that can be used to achieve this in actor systems. However, there are some tools available that, although focus on monitoring, do a very similar thing – instrument into the Akka code and collect information about actor execution and passed messages. They do not construct the messages graph, but the statistics collected by them are very useful for both monitoring and reasoning about the system (from statistics point of view) purposes.

One of such tools is Kamon [32]. It offers users an ability to collect various metrics and is designed to work with many JVM libraries, including Akka. However, it focuses solely on metrics and does not focus on messages graph. Therefore, it is lacking the ability to provide information about the message path that for example caused an exception in an actor. Kamon is also a bit hard to configure, which can discourage users from using the library. It also focuses solely on the capturing statistics – Kamon itself does not include any visualization tool, but can be plugged into many metrics collection tools, enabling easy integration between Akka application and already used metrics visualization tool.

Another tool that can be used is Akka Tracing Library [33]. It again offers metrics collection and visualization on Gant diagrams using Twitter's Zipkin [51]. While this is very useful for analysis of the timing of the actors execution, it still does not provide way to detect a fault message. It is also quite difficult to use, as it requires users to make a lot of changes in their systems.

There is one tool that can be used for displaying the trace graph – Erlang Performance Lab toolkit [19], although it is designed to work with Erlang applications. The toolkit provides many tools, including metrics collection and trace graph visualization. Unfortunately, it does not work with Akka aplications, so it cannot be used to trace actor systems written on the JVM.

As we can see, there is no tool for creating a trace graph on the JVM technological stack and the existing tools are lacking the *user-friendliness* and it often discourage users from using such libraries. Therefore, during the BSc thesis [12] and this thesis, the author proposed a solution for the problem stated in Section 1.2 – a library that allows users to generate a traces graph during actor system execution. This library, the *Akka Tracing Tool*, is thoroughly discussed in Chapter 3 and evaluated in Chapter 4.

Chapter 3. Solution – *Akka Tracing Tool*

The *Akka Tracing Tool* [4] is a *proof-of-concept* of a library capturing traces in actor systems written in Scala language [41,43] with Akka framework [2,25,47].

The library itself is written in Scala and uses as few dependencies as possible. During designing how the library should work, the user was a main focus – it was decided that it must be as user-friendly as can be done without loosing too much on performance. Therefore, user must perform very few steps to include the library and collect traces from an actor system.

The development of the library began with the BSc thesis [12] which is co-authored by Mariusz Wojakowski. The library after the BSc thesis provided very basic functionality of collecting traces but could not be run in a distributed environment. During MSc thesis, the library was refactored and new functionalities were added, such as filtering the traces and possibility to trace actor systems deployed in a distributed environment.

The rest of this Chapter is structured as follows: first, an overall principle of operation and library architecture is discussed, following by a description of all of its components and code instrumentation. Then proceeds the description of the visualization tool and the mechanism of data collection in a distributed environment.

3.1 High-level description of the principle of operation of the library

In Fig. 5 there is a diagram that shows how the library works. It provides a high-level description and does not get into much details, which are discussed in later sections.

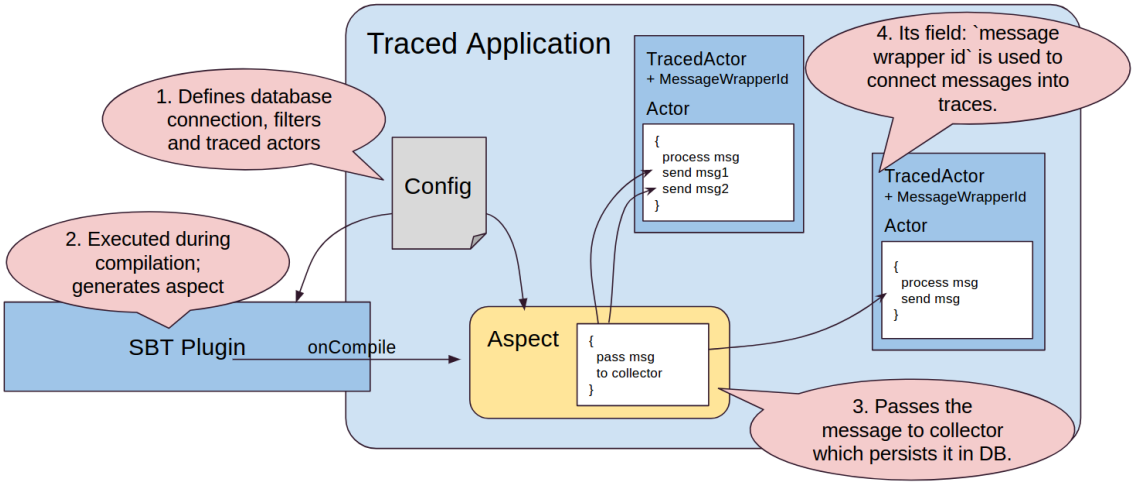

Figure 5: High-level description of the principle of operation of *Akka Tracing Tool*

1. User provides **a configuration file** to the library. It contains information on both database connection, filters and traced actors. Based on this file other parts of the

library will be generating and instrumenting proper code that will be used to collect traces.

2. **The plugin** is executed during the compilation of user's project. It generates an AspectJ aspect that will be collecting messages passed between actors in the traced system. The plugin also includes other parts of the library to the user's project. Therefore only the plugin is directly included to the project as a dependency.

3. **The aspect** captures and sends messages passed in the traced system to the collector. The collector itself is started as a separate actor system. Therefore, the internal messages are not captured. The collector persists messages into the database that user specified in the configuration file.

4. **The `TracedActor`** trait is mixed into user's actors. Its "message wrapper id" field is used to connect the messages into traces. The message succession relation is also stored into the database so the traces can be later retrieved from it.

3.2 Architecture

The library was designed to be easily extendable. Where applicable, it uses widely known design patterns, both OOP (*Object-Oriented Programming*) and functional. Code quality was assured by doing code review using GitHub [22] and continuous integration service Travis CI [48].

The library is split into three main parts:

- **Core** – contains classes to parse configuration file and create appropriate tracing filters (including filters defined by user).

- **Collector** – provides a way to persist collected traces to database. There were three standard collectors implemented:

  - **RelationalDBCollector** which provides a way to persist data in most relational databases.
  - **CouchDBCollector** which persists data in a CouchDB database
  - **NoOpCollector** – a fallback collector which does nothing.

- **SBT plugin** – provides automatic and user-friendly way to generate aspect which instruments the code of the library.

The overall library architecture can be seen in Fig. 6. User includes SBT plugin which in turn generates aspect and includes library core to the traced application. Aspect instantiates collector and put traces into it. Collector saves traces to the database (in figure it is

a CouchDB database [5, 15] with replication to the central database). Collected traces can be visualized using visualization tool (see Section 3.6).

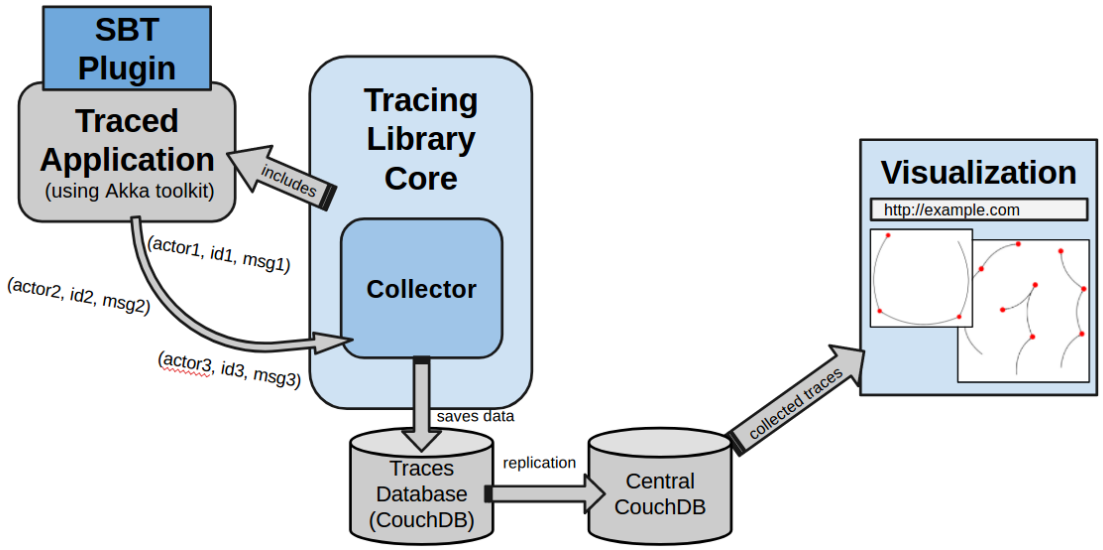

Figure 6: Overall architecture of *Akka Tracing Tool*

3.3 Library Core

The library core contains essential elements of the whole library. Its main focus is to provide all necessary elements for other parts of the library as well as classes used by users to capture (`TracedActor` trait) and filter traces (various filters operating both on messages and actors). It also contains a configuration parser that is used by the library to read configuration file and create collector and filters.

The overall class diagram can be seen in Fig. 7. As we can clearly see, it is split into five parts:

- **tracing tools** – used by library to keep track of the messages ids to save the succession relation to the database. It consists of two classes:
  - **`TracedActor` trait** – needs to be mixed into user's actors. Its purpose is to keep previous message id in the actor's state so we can save the succession relation to the database.
  - **`MessageWrapper` class** – wraps the message with its id before it's send to another actor. Therefore we can use the id to create the succession relation in the receiver actor.
- **`Collector` and `DataSource` traits** – used to persist (`Collector`) and read (`DataSource`) traces to/from database. There are fallback implementations of `NoOpCollector` and `NoOpDataSource` that do nothing or return no traces.

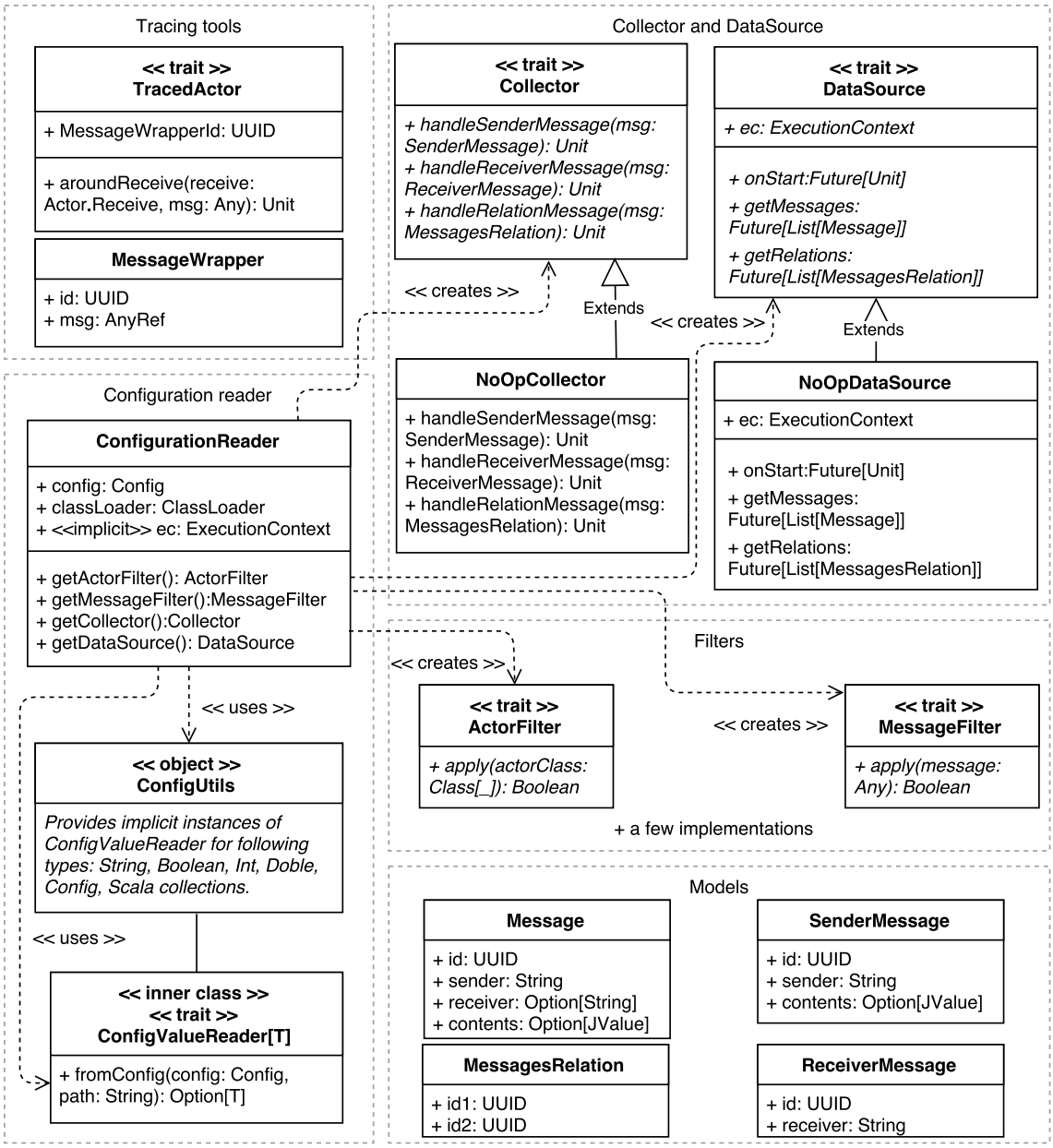

Figure 7: The class diagram of the library core. A few classes have been skipped to preserve the clarity of the figure

- **actors and messages filters** – defines classes used for filtering the collected traces. More detailed class diagram can be seen in Fig. 8. This part consists of two main traits (with standard implementations):
  - **`MessageFilter` trait** – defines filter for messages. The following standard filters were implemented:
    - ⋆ **`ByClassesAllowMessageFilter`** – filter that acts as an whitelist filter – it permits tracing only passed classes of messages.
    - ⋆ **`ByClassesDenyMessageFilter`** – filter that acts as an blacklist filter – it permits tracing all but passed classes of messages.

        - ⋆ **`StackedConjunctionMessageFilter`** – filter that acts as a conjunction of filters – it permits tracing messages that passes all filters passed to this filter.
        - ⋆ **`StackedDisjunctionMessageFilter`** – filter that acts as a disjunction of filters – it permits tracing messages that passes any of the filters passed to this filter.
        - ⋆ **`ProbabilityMessageSampler`** – filter that acts as a sampler – it permits the message with given probability.
        - ⋆ **`NoOpMessageFilter`** – a fallback filter that permits tracing any message.
    - **`ActorFilter` trait** – defines filter for actors. The following standard filters were implemented:
        - ⋆ **`ByClassesAllowActorFilter`** – filter that acts as an whitelist filter – it permits tracing only passed classes of actors.
        - ⋆ **`ByClassesDenyActorFilter`** – filter that acts as an blacklist filter – it permits tracing all but passed classes of actors.
        - ⋆ **`StackedConjunctionActorFilter`** – filter that acts as a conjunction of filters – it permits tracing actors that passes all filters passed to this filter.
        - ⋆ **`StackedDisjunctionActorFilter`** – filter that acts as a disjunction of filters – it permits tracing actors that passes any of the filters passed to this filter.
        - ⋆ **`NoOpActorFilter`** – a fallback filter that permits tracing any actor.

- **models** – define classes used throughout all library. It consists of four classes:
    - **`Message`** – read model for messages. It contains:
        - ⋆ **`id`** – UUID used to identify the message in the system.
        - ⋆ **`sender`** – actor that sent the message.
        - ⋆ **`receiver`** – *(optional)* actor that received the message.
        - ⋆ **`contents`** – *(optional)* contents of the message (serialized to JSON).
    - **`SenderMessage`** – write model for messages. It contains all the information in message class that can be provided by sender actor only (so no `receiver` field).
    - **`ReceiverMessage`** – write model for messages. It contains all the information in message class that can be provided by receiver actor only (`receiver` field (and `id` for identifying messages only)).

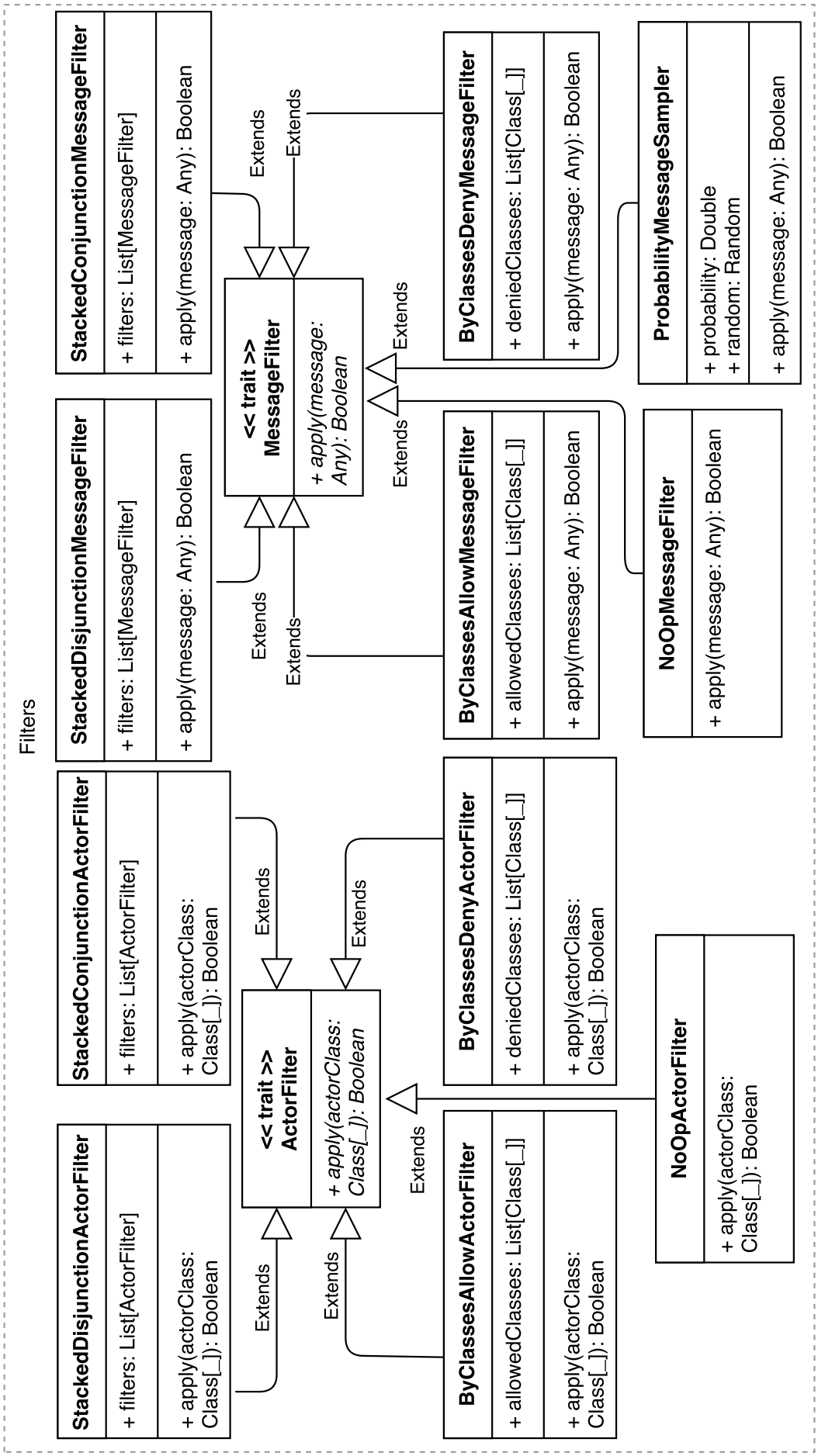

Figure 8: The class diagram of the filters

  - **`MessagesRelation`** – read/write model for messages succession relation. It contains:
    - ⋆ **`id1`** – UUID used to identify the predecessor message in the system.
    - ⋆ **`id2`** – UUID used to identify the successor message in the system.

- **configuration reader** – a few classes used for reading configuration file, parse it and provide the necessary components (actor and message filters, collector, data source) to other parts of the library. The main class used is `ConfigurationReader`.

3.4 SBT Plugin and code instrumentation

The SBT plugin is used to automatically generate aspect used for code instrumentation and include the core library and collector into user's actor system. Its main purpose is to provide automation to the user's build. The plugin provides also tasks for creating and cleaning database.

The main purpose of the plugin is to create an aspect, which instruments the library code into user's actor system. The aspect plugs into the internal Akka code, which enables wrapping/unwrapping messages into our `MessageWrapper` and sending messages and messages succession relation into the collector which persists it to the database. The aspect also includes filters defined by user to save only traces that user wants to save.

The aspects are written in AspectJ [17, 34]. It is one of the most popular Aspect-Oriented Programming (AOP) technology used on Java Virtual Machine (JVM). This method of code instrumentation was chosen, as it is widely used – e.g. by Kamon [32] and other tracing tools, such as ExplorViz [20, 21].

In Fig. 9 we can see class diagram of the plugin. As we can clearly see, it has four main components:

- **`AkkaTracingPlugin`** – this is the main class of the plugin. It provides the following tasks or setting keys for users:
  - `configurationFile` – enables users to provide their own name of the configuration file. This value is passed to the Lightbend's Config library [14]. Default: `akka_tracing.conf`.
  - `collector` – enables users to specify which collector they want to use. Defaults to `None`. If it's set to `None`, it'll be read from configuration.

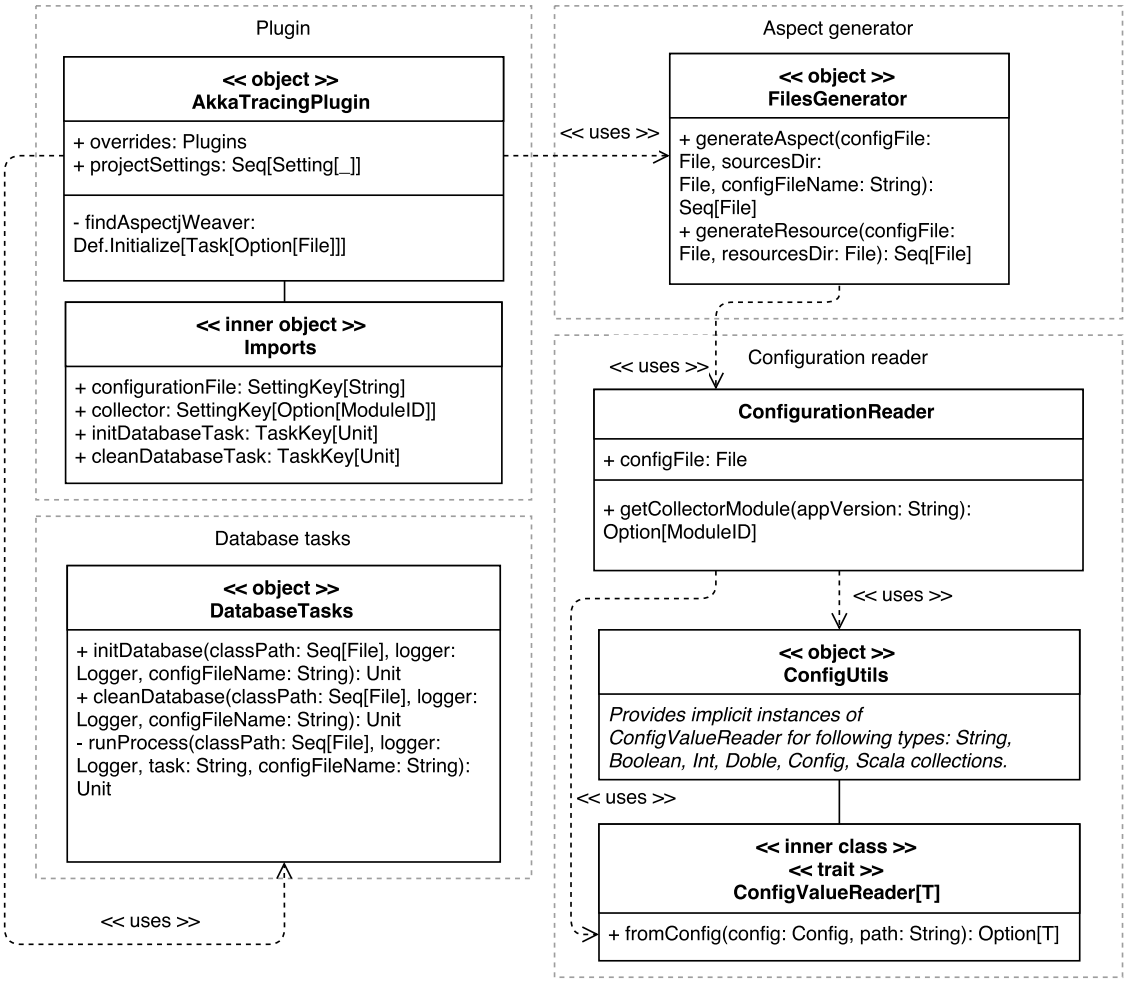

Figure 9: Class diagram of SBT plugin

  - `initDatabase` and `cleanDatabase` – provide database utilities tasks to initialize and clean database.

- **Configuration tools** – due to the fact that in SBT 0.13 you can write code only in Scala 2.10 and Akka 2.4 is only available for Scala 2.11 and 2.12, the code in the core of the library cannot be used inside the plugin. Therefore, there was a need for code duplication and a lot of the configuration from core is also defined here.

- **`DatabaseTasks`** – utilities to run database tasks. They actually create a new Java process which runs a class named `PluginOperations` which is defined in every collector (besides the `NoOpCollector`), passing the name of the configuration file as well as whether to initialize or clean the database.

- **`FilesGenerator`** – provides automatic aspect generation with `aop.xml` configuration file.

3.5 Collectors

There are two collector implementations provided by *Akka Tracing Tool*:

- **`RelationalDbCollector`** – provides way to persist data into most popular relational databases. It uses Slick 3.2.0 [45] under the hood, a Scala library very commonly used for data persistence and therefore supports databases that Slick itself is supporting:
  - DB2,
  - Derby/JavaDB,
  - H2,
  - HSQLDB (HyperSQL),
  - Microsoft SQL Server,
  - MySQL,
  - Oracle,
  - PostgreSQL,
  - SQLite.
- **`CouchDBCollector`** – provides way to persist data into CouchDB [5, 15], a document database with very fast replication protocol. It also supports performing replication (it is being set up in the database task `initDatabase`.

These collectors provide a quick way to just plug in the library for most use cases and start tracing an actor system. However, if user wants to persist data to other database, which is not officially supported, it can be done by simply extending the `Collector` trait and provide the `CollectorConstructor` so the library knows how to parse configuration and create a collector.

3.6 Visualization tool

During early stages of library development, a simple visualization tool was created to prove that the library is indeed working as expected. While not being a main focus of the development, it fulfilled its role and can be used when tracing smaller systems to visualize the collected data. In Fig. 10 the screenshot of the visualization application is shown after collecting data in a very basic example of an actor system. The tool provides a graphical view of the succession relation – the nodes represent actors and the edges represent messages. It can also show message contents, if it was saved during the sending of the message.

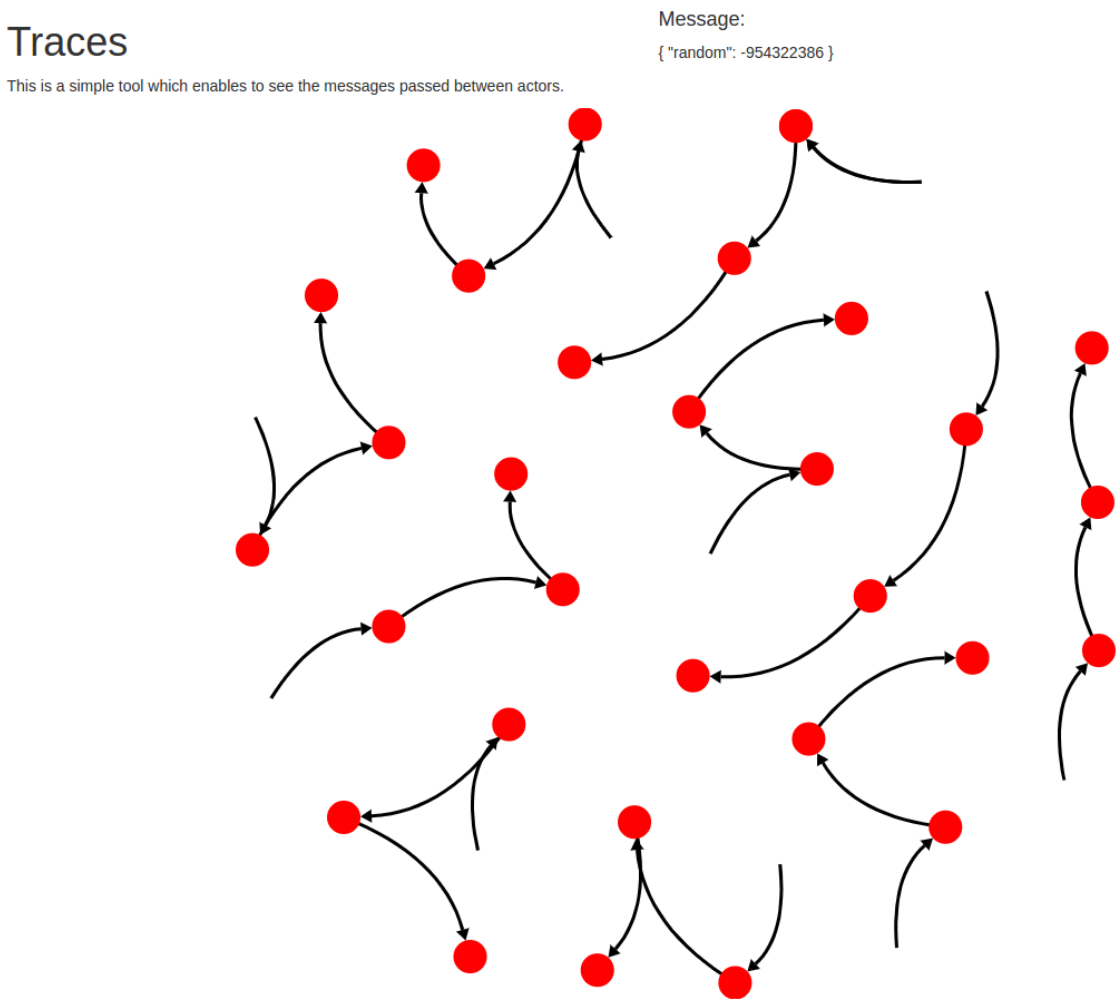

Figure 10: Visualization tool – provides a simple way to see collected traces

3.7 Configuration

The library is highly configurable. In the configuration file user can define:

- **filters** – they define which messages and actors are going to be traced. The filters available to users have been discussed in Section 3.3.
- **collector** – it defines which collector is going to be instantiated. Inside this key user has to also define:
  - **collector constructor** – it defines how to parse the configuration at this key to create the collector,
  - **database connection** – it tells collector how to connect to database.
- **packages** – it tells the AspectJ weaver which packages should be considered to weave the aspect in, so it also defines which packages will have instrumented code of the library.

The configuration file is read by the aspect which then constructs collector and filters based on the parsed configuration.

3.8 Data collection in distributed environment

In any monitoring solution for distributed systems, efficient data collection is a very important challenge to overcome. If data is collected inefficiently, the performance of the system being monitored will decrease, which is a very unwanted side-effect. On the other hand, if the data is distributed on the nodes, it would be very difficult to reason about the whole system status just looking on one node. Therefore, queries would be performed on the whole cluster, leaving the monitoring system inefficient in generating alarms if, for example, some component is failing or is overloaded.

Tracing is no different from monitoring – for efficient operation of the large-scale actor systems, a scalable way to collect traces is needed in distributed environment. During this thesis, such method for collecting data in large-scale actor systems has been proposed. As can be seen in Fig. 11, the mechanism proposed contains of a local CouchDB database into which the library saves the data. The traces are then replicated into the central CouchDB databse, which can also be clustered.

The CouchDB has been chosen as it has a well-defined replication mechanism with several properties that are beneficial for the needed solution:

- the replication is performed only in one-way, into the central server, therefore saving the network traffic compared to the classic two-way replication,
- the CouchDB replication mechanism is very efficient and requires very little configuration.

These properties satisfy the requirements of the efficient data collection mechanism, as the library can simply put collected data into the local database (the operation is quick as it connects to the database on the local node) and the database would perform replication (which is efficient) to the central database (or central cluster of databases). The one-way property of the replication is another benefit – it saves network bandwidth as the data are not transferred both ways in the system.

This architecture has been successfully used to perform various tests on the Amazon Web Services (AWS) [9] infrastructure. The tests are described in Chapter 4.

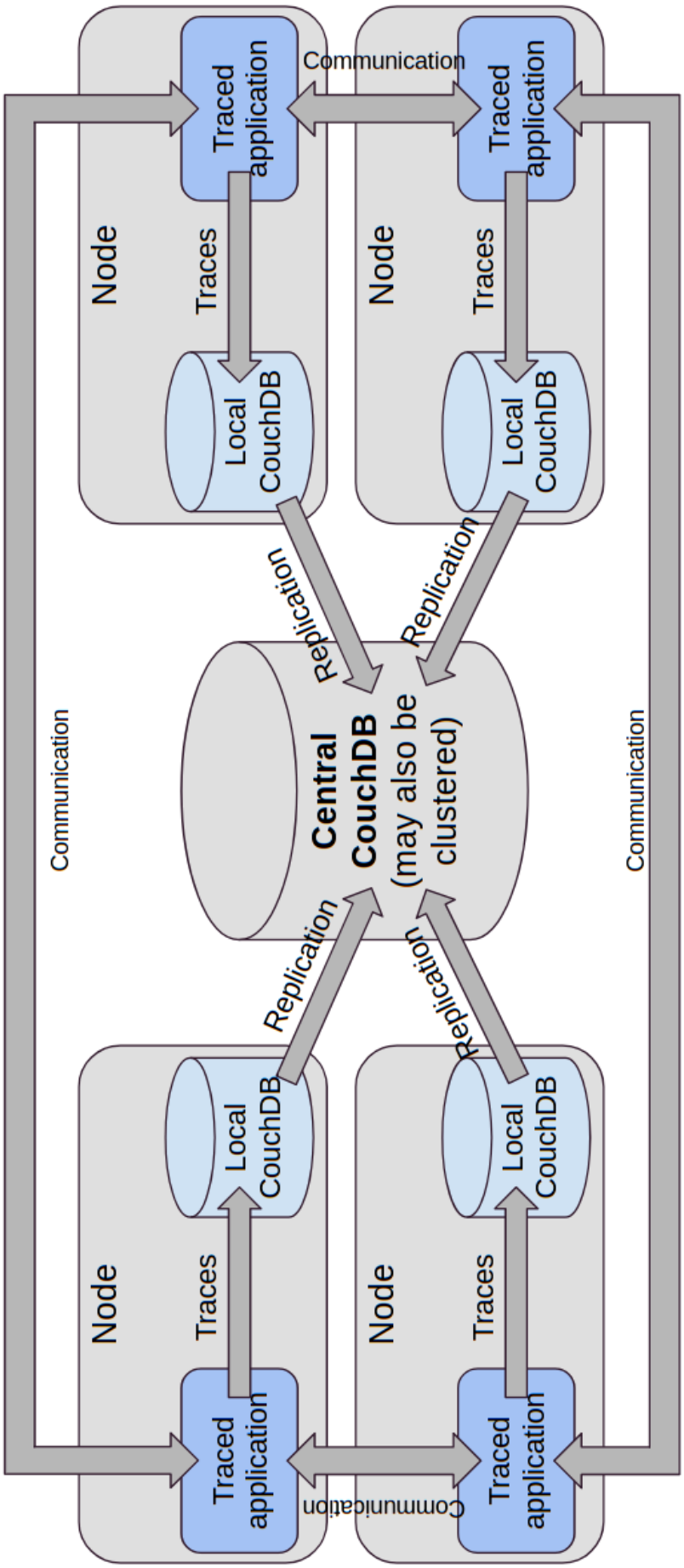

Figure 11: Mechanism for efficient data collection in distributed actor systems

Chapter 4. Evaluation of the solution

In this chapter, the derived solution is being thoroughly evaluated. The evaluation is mainly focused on the impact on the traced application's performance, as it is the most important aspect of monitoring tools. It also includes the information on the evaluation of the *user-friendliness* of the solution.

The Chapter is structured as follows: Section 4.1 describes how the tests were designed, Section 4.2 contains description of the application being used for tests, followed by the discussion of the test environment (Section 4.3). Then, the results are described: the performance is discussed in sections 4.4 and 4.5 and the *user-friendliness* is evaluated in Section 4.6. Finally, the results are summarized in Section 4.7.

4.1 Tests methodology

The impact on performance of the traced application is not an easy thing to measure. On one hand, someone can measure how much time does it take to process a single message. The problem with this approach is that different messages can take different time to process, even the same message can be handled quicker or slower, depending on overall system load and the performance and load of the external systems. Of course, mean value of all of the messages can be calculated, but it can lead to the situation when a lot of the operations in the code will be getting the timestamp of the start and end of the processing of the messages. This can affect the system itself as the operations that are so frequent (every message there are two invocations of time measurement) can affect the performance of the traced system.

Therefore, if one can measure the performance of the application in another way, it would provide a better approach to measure the library performance impact. A lot of metrics that are very often calculated or measured anyways can be used, for example:

- in simulations, frames per second (FPS) can be used, as it is very often calculated anyways by the simulation software,
- in HTTP servers, number of requests per second can be used – it is often being monitored,
- in data stream processing – the average time to process a single chunk of data.

These metrics can be used to perform the measurements of the tracing tool's impact on the performance application and what is more, they are very often calculated in modern systems anyway so we can just use the existing monitoring software to measure the performance and not add another piece of code to do it ourselves. This creates a better approach as the environment of the system is as close to the real one as possible, so the measurements are more precise.

This approach has been used to measure the performance impact of the *Akka Tracing Tool* on the system being traced. As the measure, FPS has been chosen, as the application is a simulation (see more about the application itself in the Section 4.2).

The test environment was designed to be as close as possible to the environment which many actor systems uses – a computing cloud. This means that the measurements are as accurate as possible, as the environment is very similar to the one used in production systems. For more information about used environment, see Section 4.3.

To assess the library, 3 tests were prepared. The first test was to measure the impact of the library on the performance of the traced system. The FPS is measured when application is running without the library and then with the library. Therefore we can calculate how the tracing tool is impacting the simulation, how much less FPS were achieved when the tracing was enabled.

The goal of the second test was to measure if the impact of the library is proportional to the number of messages being traced. In this test, the messages were sampled with given probability to reduce the amount of the messages being actually processed by the library while the simulations were conducted with the same set of parameters. More information about this test can be read in Section 4.5.

Lastly, an assessment was conducted about how much work the users need to perform to include the library itself to their project. This measures the *user-friendliness* of the library. The evaluation provides a list of the actions that users need to perform to include the library to their project. These lists are compared between Akka Tracing Library [33], Kamon [32] and the proposed solution. The assessment can be found in Section 4.6.

4.2 Test application – car traffic simulation

The car traffic simulation [29] was chosen as the application that will be traced by the tool. The application has several properties that are useful for performing tests of the proposed solution:

- it is producing a very large amount of messages,
- it has been proven to be scalable, so the application itself is not impacting the scalability of the tracing tool,
- it measures the FPS, which can be then used to measure the impact on the performance of the tracing,
- the communication between actors is a very large part of the simulation – it takes most of the simulation's time if run with not large traffic density.

These properties enable to use the FPS as the main metric to measure the performance of the application and the impact on the performance that the tracing library creates when tracing is enabled.

The simulation has been also found to be scalable (see [29]) – so this means that the application being traced is not affecting the library on its scalability.

The simulation has following assumptions/properties:

- the city is made with the orthogonal roads (as in Manhattan) – it is easy to divide such city between actors,
- the city is divided in areas of the same size, each area is being processed as a separate actor,
- each actor communicates only with its neighbours – it passes information about cars leaving the area and available space on the input roads,
- the city is a torus – that means that a car that is travelling left leaving the first area on the left will appear on the first area on the right,
- each car is simulated independently, that means that this is a microscopic simulation,
- each road has three lanes and is divided into *cells* – parts of road that corresponds to about 5m in real life.

The simulation has following configurable parameters:

- **rows** and **cols** – amount of the rows and cols in the city, the city is divided into a table with `rows` rows and `cols`, so it contains `rows`$\times$`cols` areas,
- **apn** – the amount of areas per node – this means how many actors are being simulated on a single node,
- **traffic density** – at the start of the simulation, traffic is randomly put onto the roads. This parameter controls how many cars are being simulated (what is the probability that the cell contains a car),
- **road cells** – how long is the road between the intersections,
- **area size** – how many inersections are on the area, this parameter must be a square of a natural number (e.g. 1, 4, 9, 16, 25, ...).

In Fig. 12 we can see a screenshot from the visualization of the simulation.

Figure 12: Visualization of the car traffic simulation – in this case it has 4 areas (4 actors)

4.3 Test environment – AWS cloud

The tests were performed on a cloud infrastructure. It is an environment very commonly used to run actor systems, therefore using it to perform tests means that we are running evaluation on the environment that is very likely to be used for running production systems. As the cloud provider, Amazon Web Services (AWS) [9] was chosen – one of the biggest and most popular platform as a service (PaaS) provider. Its service – Amazon Elastic Compute Cloud (Amazon EC2) – is used by many companies all over the world, many of which are running actor systems on the machines that the service provides.

AWS machines are called *instances*. Amazon provides many different type of instances, both general purposes types (*t2*, *m3* and *m4* instances) and optimized instances for different purposes. Optimized instances with Linux operating system includes instances:

- for computation (*c3* and *c4* instances),
- for storage (*i3* and *d2* instances),
- for memory (*x1*, *r3* and *r4* instances),
- with GPUs (*p2* and *g2* instances).

The instances have different amount of virtual CPUs, memory and storage. For test purposes, we used *t2.micro* instance (for running the supervisor) and *c4.large* instances (for running workers). The hardware configuration of these instances are wrapped up in tables 1 (*t2.micro*) and 2 (*c4.large*).

Table 1: Hardware configuration of AWS EC2 *t2.micro* instances

Component	**Available hardware**
vCPU(s)	1 (CPU type not listed)
Memory	1 GB
Storage	Elastic Block Storage (EBS) (attached via network)

Table 2: Hardware configuration of AWS EC2 *c4.large* instances

Component	**Available hardware**
vCPU(s)	2 × Intel Xeon E5-2666 v3: 2.9 GHz, 10 cores, 20 threads, 64 bit
Memory	3.75 GB
Storage	EBS (attached via network)

While the workers of the simulation do need a bit of computing power (hence the usage of the *c4.large* instances), the supervisor practically only sets up the simulation and does nothing during the computation (that is why it uses cheaper, less powerful *t2.micro* instance).

The software being used while running tests is presented in Table 3. Please note that the CouchDB is running on the Docker container as it was much less complicated to install in this way than downloading from sources as Ubuntu currently does not have an official package to install the most recent version of the database.

Table 3: Software being used on test environment

Software name	Version
Operating System	Ubuntu 16.04.2 LTS 64 bit (Linux kernel: 4.4.0-1017-aws)
Java (JDK)	1.8.131 (Oracle)
Scala	2.11.11
SBT	0.13.9
Akka	2.4.17
AspectJ	1.8.10
CouchDB (runs on Docker)	2.0.0
Akka Tracing Tool	0.1
Python	2.7
Boto	2.43.0
Docker	17.05.0-ce

For tests automation, scripts written in Python language were used. They use Boto library to gather information about running instances and then connects to them via SSH and runs commands which enable to run the simulation. The main output of these commands are saved onto the supervisor's disk.

During tests, maximum number of instances that were used did not exceed 51 (50 workers + supervisor). Of course, more instances could be used, but these tests were enough to check whether the library scales linearly and it enabled cost reductions (the *c4.large* instances are not cheap).

4.4 The impact on performance of the traced application

The first kind of tests performed was to check whether the library does scale linearly while increasing the amount of nodes being used for simulation. The impact on the simulation's FPS was calculated while increasing the size of the city and adding more nodes to the simulation. There were 2 tests conducted of this kind, they differ on the value of the APN parameter. The Table 4 contains parameters used for this experiment.

Table 4: Simulation parameters used during the performance impact tests

Parameter name	Rows/cols	APN	Traffic density	Road cells	Area size
Test 1	Varying	8	8%	100	4
Test 2	Varying	16	8%	100	4

4.4.1 Test 1 – APN: 8

These tests were conducted with 8 actors per single worker node. That means that the average amount of actors did not exceed half of the number of available cores. This situation is not common during normal actor systems operations – normally the number of actors is very high, but the actors do not perform such time consuming operations as in simulations. Therefore it is a good model of an actor system that is not very loaded. However, the actors can still produce and process a very high number of messages (even hundreds of thousands per minute).

The simulation was run for each city size 10 times, then the FPS average and standard deviation were calculated and chart showing the relationship of FPS from nodes was drawn. The repetitions were necessary as the cloud environment can change a little at any point of time (for example, a virtual machine can be run on a different physical machine).

The results are shown in Fig. 13 and in Table 5. The performance of the application does decrease significantly when the library is turned on, but the performance loss (approximately 70% FPS loss on average) remains more or less constant while we are increasing number of nodes. The small decrease in application's performance seen at about 13-16 nodes is mirrored by the library.

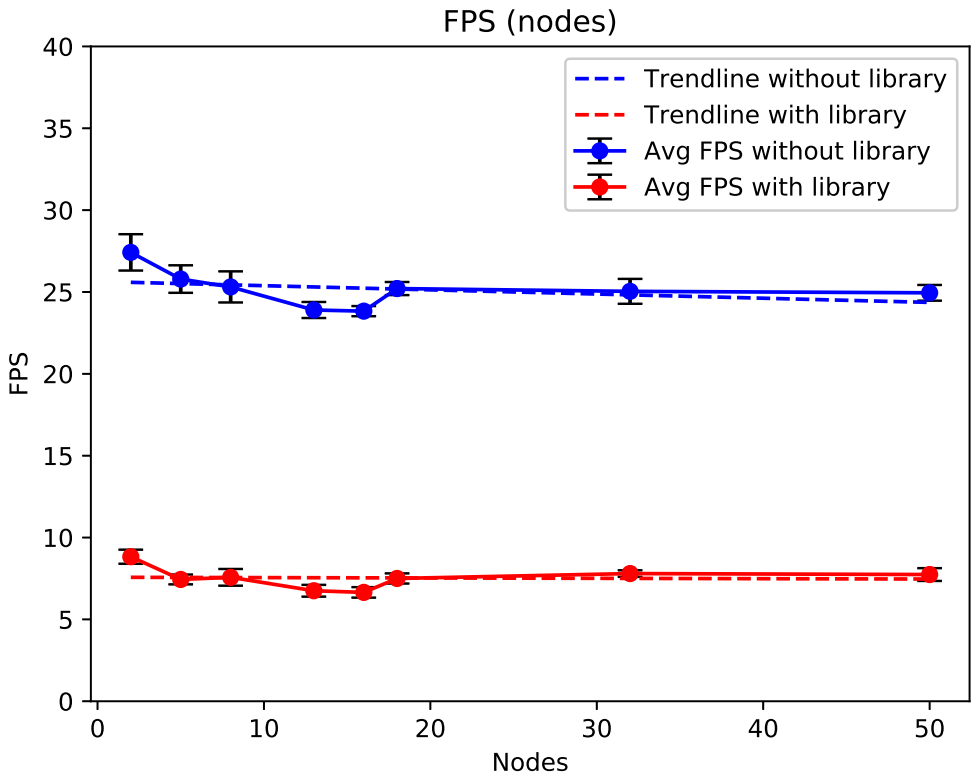

Figure 13: Chart showing the performance impact of the library when the number of actors running on node is less than half available CPU cores (APN: 8)

Table 5: The performance impact of the library when the average number of actors running on node is less than half available CPU cores (APN: 8)

City size	4×4	6×6	8×8	10×10	11×11	12×12	16×16	20×20
Areas (actors)	16	36	64	100	121	144	256	400
Nodes	**2**	**5**	**8**	**13**	**16**	**18**	**32**	**50**
Average FPS (without library)	**27.42**	**25.79**	**25.31**	**23.90**	**23.83**	**25.21**	**25.04**	**24.95**
Standard deviation	**1.11**	**0.84**	**0.95**	**0.49**	**0.31**	**0.40**	**0.76**	**0.48**
Average FPS (with library)	**8.83**	**7.44**	**7.57**	**6.75**	**6.65**	**7.50**	**7.80**	**7.74**
Standard deviation	**0.43**	**0.30**	**0.51**	**0.36**	**0.43**	**0.31**	**0.20**	**0.39**

The results can lead to the conclusion that in this setup the library does scale linearly while we increase the number of nodes that the actor system is running on. This was a fundamental requirement for the library being the subject of this thesis.

4.4.2 Test 2 – APN: 16

These tests were conducted with 16 actors per single worker node. That means that the average amount of actors was at about 75% of the number of available cores. This setup can model a situation when the actor system is a bit loaded, but the load is far from the CPU limits. This is a common situation in a normal day operation of production actor systems – the load is high, but there are no rush hours.

The simulation was run for each city size 10 times, then the FPS average and standard deviation were calculated and chart showing the relationship of FPS from nodes was drawn.

The results are shown in Fig. 14 and in Table 6. The performance of the application does decrease significantly when the library is turned on, but the performance loss (approximately 68% FPS loss on average) remains more or less constant while we are increasing number of nodes. The small decrease in application's performance seen at about 13 nodes is mirrored by the library.

The results confirm the conclusion that in this setup the library does scale linearly while we increase the number of nodes that the actor system is running on.

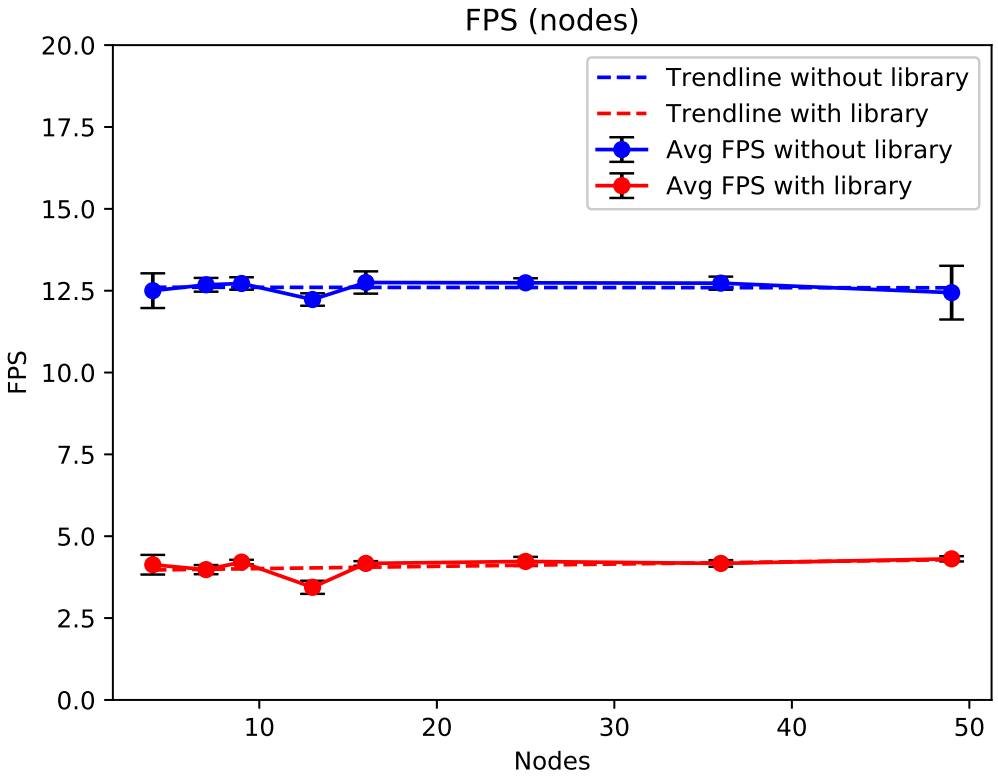

Figure 14: Chart showing the performance impact of the library when the average number of actors running on node is at about 75% of available CPU cores (APN: 16)

Table 6: The performance impact of the library when the average number of actors running on node is at about 75% of available CPU cores (APN: 16)

City size	8×8	10×10	12×12	14×14	16×16	20×20	24×24	28×28
Areas (actors)	64	100	144	196	256	400	576	784
Nodes	**4**	**7**	**9**	**13**	**16**	**25**	**36**	**49**
Average FPS (without library)	**12.50**	**12.68**	**12.72**	**12.23**	**12.75**	**12.74**	**12.73**	**12.44**
Standard deviation	**0.53**	**0.21**	**0.19**	**0.19**	**0.34**	**0.14**	**0.20**	**0.82**
Average FPS (with library)	**4.13**	**3.98**	**4.21**	**3.44**	**4.17**	**4.23**	**4.17**	**4.31**
Standard deviation	**0.30**	**0.14**	**0.07**	**0.20**	**0.07**	**0.14**	**0.10**	**0.08**

4.5 The performance overhead with different tracing configurations

The tests conducted before had the assumption that we want to trace every single message in every actor in the actor system being traced. While this configuration is common, very often user may want to trace only part of the system or, if the system is very large, the messages can be sampled to preserve the performance of such systems. *Akka Tracing Tool* has the concept of message and actor *filtering* that enables users to collect data only from specific actors, trace only specific messages and the messages can be sampled. All these features are available using the provided filters from the library core (see Section 3.3 and Fig. 8).

While the library provides the possibility to do this, it is useful to know how the impact of the performance on the traced system is related to the amount of traced messages. To measure this, the following test has been conducted: given the size of the city and the number of nodes, check how increasing message sampling will affect the performance impact on the traced system. The increasing sampling probability simulates the different filtering configurations that user can have that decrease/increase the number of traced actors and messages. The simulation parameters that were used in this test are presented in Table 7.

Table 7: Simulation parameters used during the overhead tests with different tracing configurations

Parameter name	Rows/cols	APN	Traffic density	Road cells	Area size
Value	16×16	16	8%	100	4

This test has been performed on 16 nodes with 16 areas (actors) per node. This configuration is sufficient to check this, as the results from Section 4.4.2 show that the performance stabilizes at this point.

The results of these tests are shown in Fig. 15 and in Table 8. The overhead of the library (the FPS loss) quickly raises between sampling probabilities 0% and 10%. This means that the overhead of the persistence mechanism is very high. As we move to higher sampling probabilities, the overhead increases (nearly linearly) to about 65% and then remains nearly constant (after 60% sampling probability). This shows that the overhead of the library depends on the number of messages, but the gains from introducing filtering is not that great. However, filtering traces can be valuable – if the only interesting data can be collected only from one part of the system, it would be better to filter the messages or actors – it would cause less data being persisted in the database – so the data extraction and other operations on the collected traces would be much quicker.

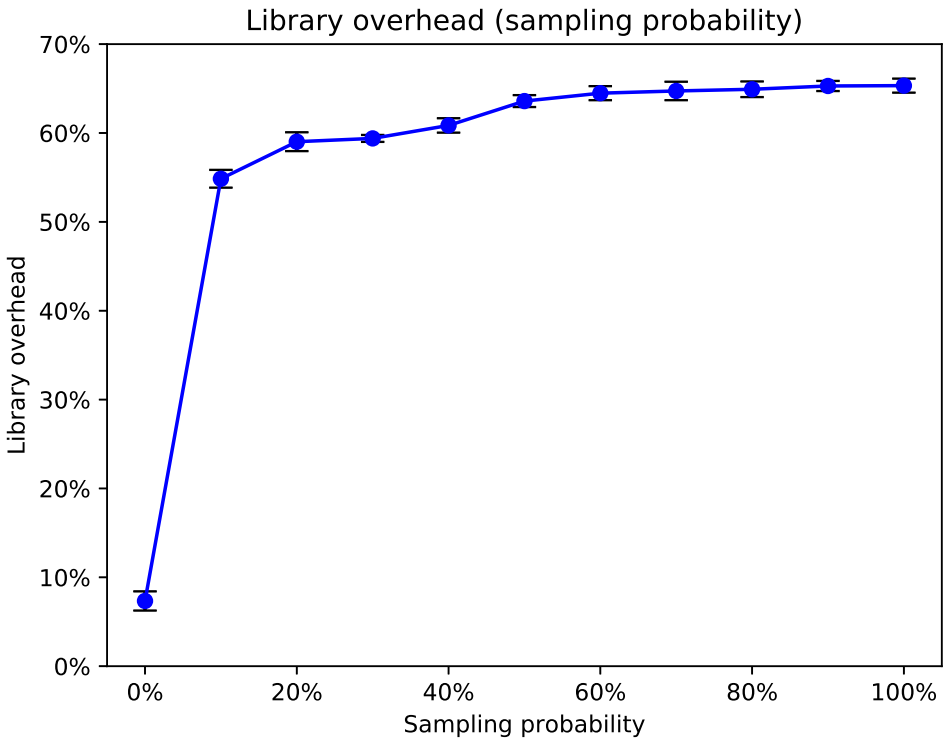

Figure 15: Chart showing the performance impact of the library with different tracing configurations (APN: 16)

Table 8: The performance impact of the library with different tracing configurations (APN: 16)

Sampling probability	Average FPS	Standard deviation	Library overhead	Standard deviation
No library	12.75	0.34	–	–
0%	11.82	0.14	7.3%	1.1%
10%	5.76	0.13	54.9%	1.0%
20%	5.23	0.14	59.0%	1.1%
30%	5.18	0.05	59.4%	0.4%
40%	4.99	0.10	60.9%	0.8%
50%	4.64	0.09	63.6%	0.7%
60%	4.53	0.10	64.5%	0.8%
70%	4.50	0.13	64.7%	1.0%
80%	4.47	0.11	64.9%	0.9%
90%	4.43	0.07	65.3%	0.6%
100%	4.42	0.10	65.3%	0.8%

4.6 The *user-friendliness* of the tool

Evaluation of the *user-friendliness* of any tool or library is a very difficult task. This concept is very subjective – some programmers can say that for them annotating 100 classes with 2 annotations is not a big deal, whereas for other users this is a very burdensome task and they would avoid it like a plague.

That being said, one of the requirements of the library that is the subject of this thesis was to create a tool that is user-friendly, that users will find it to be very easy to use. Therefore, such evaluation needed to be performed. As it would be quite difficult and inconvenient to ask developers to use the library (e.g. go through a tutorial) as well as several others, another metric needed to be proposed.

The author decided to compare the amount of actions needed to be performed in the proposed library as well as two similar libraries – *Kamon* [32] and *Akka Tracing* [33] – and check which one needs the least actions. The results are shown in Table 9.

The table was created with following assumptions about the usage of the libraries and the actor system being traced or monitored:

- an actor system is using Akka Remoting,
- the full potential of the libraries should be utilized,
- all possible data should be collected,
- the actor system is run from SBT,
- the data needs to be visualized as it is better to reason about anything if it is brought as a figure instead of raw numeric data,
- only necessary configuration files and options should be added.

Each action that users need to perform to include the library is worth some points. In the case of configuration, no one likes to do it – that is why every line in configuration or build tools files is worth 1 point. Every very burdensome task is worth 20 points, especially one that needs action in many source files. In case of using external tools, it is worth 5 points, whereas using provided tools is worth 2 points.

The results presented in the table were obtained by trying to add the libraries to a very simple project, containing only 2 actors passing messages between them. In case of *Kamon*, the Java Message Extensions (JMX) recorder has been chosen to collect data. In every library, standard configuration settings were used and only the necessary ones were configured.

The proposed library – *Akka Tracing Tool* – requires actions worth 42 points, while other libraries require actions worth much more (73 points in *Kamon* and 72 points in *Akka Tracing Library*). It shows that the proposed library has much less actions that users need

to perform to include it to their project; and due to this fact, the mentioned requirement that the proposed solution is user-friendly has been fulfilled.

Table 9: Comparison of actions needed to be performed by users that want to use *Akka Tracing Tool* and other similar libraries (*Kamon* [32] and *Akka Tracing* [33])

Criterion	***Akka Tracing Tool***	***Kamon* [32]**	***Akka Tracing* [33]**
Adding library to project	5 lines in SBT files (+5 points)	4 lines in SBT files (+4 points)	1 line in SBT files (+1 point)
Modifying actors	mix trait to every actor (+ 20 points)	none (0 points)	mix trait to every actor (+ 20 points)
Modifying messages	none (0 points)	none (0 points)	mix trait to every message (+ 20 points)
Configuration	15 lines in akka_tracing.conf (+15 points)	29 lines in application.conf (+29 points)	6 lines in application.conf (+6 points)
Instrumentation	automatic (running AspectJ automatically) (0 points)	manual (running AspectJ manually or by SBT plugin), needs also manual start and stop, manual context propagation (+30 points)	semi-manual (running as Akka extension), messages need to be explicitely traced (+20 points)
Collecting data	automatic (0 points)	external tool (+5 points)	automatic (0 points)
Visualization	provided tool (+2 points)	external tool (+5 points)	external tool (+5 points)
Total	**42 points**	**73 points**	**72 points**

4.7 Results summary

In this chapter, the proposed library was thoroughly evaluated. Three sets of tests were conducted and resulted in the following:

- The first set of tests checked whether the library is scalable. It measured the impact of the library on the traced system performance. It occurred that the average library overhead (the loss of the FPS of the simulation – the actor system being traced) was 70% when there were 8 actors running on a single node and 68% when there were 16 actors running on a single node. The overhead is quite high, but the tests show that despite the FPS loss, the library has still proved to be scalable with respect to the number of nodes in actor system.

- The second test checked if the data collection mechanism is scalable with respect to the number of messages. The library overhead (FPS loss) was 7.3% when the message sampling probability (the probability whether the message was included into the trace) was set to 0% and then changed between 54.9% to 65.3% when the probability was changing from 10% to 100%. This means that the data collection mechanism, while still not impacting the library scalability, introduces very large overhead to the library.

- The last assessment checked whether the library is user-friendly. The author compared the amount of actions needed to be performed to include the proposed solution as well as similar libraries to a very small actor system and check which one needs the least actions. The results show that the proposed solution needs the least amount of actions, thus proving to be the easiest to use compared to other libraries.

Chapter 5. Summary

In this Chapter the author provides the summary of the thesis. In Section 5.1 the objectives set up in Section 1.3 are revisited and the assessment is given if they were solved in this thesis. The Section 5.2 contains the conclusions from the solution's evaluation and in Section 5.3, the author provides future research that can be conducted about tracing actor systems.

5.1 Goals achieved

In Section 1.3 several objectives of this thesis were defined. In this Section, the author discusses these goals and provides descriptions how they were achieved in this thesis.

✓ **Create a *proof-of-concept* of a library written in Scala language which enables creating the traces graph in a scalable way for applications written in Akka framework.**
In Chapter 3, author describes a *proof-of-concept* of such library. The tests presented in Chapter 4 proved that the solution is linearly scalable while increasing the number of nodes in actor systems. The tests were conducted in an environment that is very common for production actor systems (a computing cloud) on a real application, so the results are as accurate as possible in terms of reproducing the environment that the actor systems are deployed to.

✓ **Prepare an efficient mechanism able to collect data about the traces from a distributed actor system.**
The mechanism has been proposed in Section 3.8. It consists of providing a local database – CouchDB – to every node of the application and setting up one-way replication to the central database. The solution was later evaluated in Section 4.5. The mechanism proved to be scalable, although its efficiency is something that more work needs to be done in the future. This will be discussed in Section 5.3.

✓ **Conduct large-scale tests of the library on the impact on the performance of the traced systems real application.**
The tests in large-scale environment (computing cloud) have been conducted and are thoroughly described in Chapter 4. The author evaluated proposed solution both on the impact of the library on the performance of the system being traced as well as the *user-friendliness* of the library. Both kind of tests proved the proposed solution to be feasible in production environment.

5.2 Results of the study

The author proposed *tracing* as a method of monitoring a distributed application written as an actor system. As there were no existing tools for Akka actor library, a *proof-of-concept* of a library that enables to create the traces graph of actor systems written in Akka has been created and then evaluated.

The tests show that the average library overhead (the loss of the frames per second (FPS) of the simulation – the actor system being traced) was 70% when there were 8 actors running on a single node and 68% when there were 16 actors running on a single node. The overhead is quite high, but the tests show that despite the FPS loss, the library has still been proven to be scalable with respect to the number of nodes in actor system – the simulation was still run and the FPS was more or less constant when the city size was increasing as well as number of nodes was increased to match the city size.

The other test that was conducted was to check if the data collection mechanism is scalable with respect to the number of messages. This was conducted on a simulation with constant city size, but a sampler was introduced that changed the probability whether the message should be added into a trace or not. The probability was increasing from 0% to 100%. The library overhead (FPS loss) was 7% when the probability was set to 0% and then changed between 55% to 65% when the probability was changing from 10% to 100%. This means that the data collection mechanism, while still not impacting the library scalability, introduces very large overhead to the library and needs more work.

The last test conducted was about the *user-friendliness* of the library. As there are no other tools for tracing actor systems, the author compared his solution to other similar libraries – *Kamon* [32] and *Akka Tracing* [33]. The author decided to compare the amount of actions needed to be performed to include the mentioned libraries to a very small actor system and check which one needs the least actions. The results show that the proposed solution needs the least amount of actions. That means that the library is easy to use by potential users.

All in all, a new way to monitor distributed actor systems has been proposed – *tracing*. The author provided a *proof-of-concept* of a library that realizes the proposed concept. The tests have shown that the library is scalable with respect to the number of nodes in actor system, but the data collection mechanism needs still more work, as it introduces a large overhead – the FPS of the test application (car traffic simulation written as an actor system) suffered 68% to 70% loss on average. The overhead is caused probably by poor management of threading and concurrency inside the collector as well as the extensive usage of network during replication process.

5.3 Future work

There are several areas that the future research can focus on as well as there are several things that the library needs more work on.

Firstly, the data collection mechanism (described in Section 3.8) needs more work to reduce the overhead it introduces to the library. It is a very difficult task as the data comes from many actors (there are many sources of data) and it must be put in a single local database. There is also a large number of messages being saved (even millions or hundreds of thousands messages per minute) so the database must be able to take them quite efficiently. The author focused on CouchDB [5, 15] database, due to both his familiarity with it and efficient replication mechanism. One can check the performance of other databases that can provide similar replication mechanisms.

Secondly, the visualization tool provides only very basic insight on the collected traces. As was stated in Section 3.6, this part of the solution was not a main focus of the development process. The visualization tool cannot be used to visualize large traces and lacks the ability to e.g. filter collected data, process collected data, etc. The topic of the traces visualization was certainly not in the scope of this thesis and needs more work. One of the tools discussed in Section 2.3, *Erlang Performance Lab toolkit* [19], provides a great interface for showing the messages being passed between actors. It would benefit the user experience of using the proposed library if an integration can be done between this kind of visualization and collecting traces from Akka systems.

What is more, the library currently does not collect any statistics about the actor system. It would be very beneficial, if the library provided data that, for example, Kamon [32] provides: the average time that message is in mailbox, average message processing time, mailbox size, the number of actor failures, etc. If one can integrate this metric collection mechanism into the library, it would be very useful for users.

Another beneficial thing would be to introduce a trace search tool. This would be beneficial for users that want to search a large traces data set e.g. to answer if there is a message with wrong data, if there was an order from a specific user, etc. It could include some *data mining* algorithms and methods as well as *machine learning*. This would enable, for example, to detect anomalies in messages and alert the system administrators that something is wrong with the system used in production. Other use cases may include pattern detection in messages (can be useful for example to detect which products are commonly bought together) or prognosis of system load.

The last thing that author considers for future research is the analysis of different methods of code instrumentation. While using aspects allows to practically do any action in Akka library, especially in the private API and internal code, it is not really user-friendly – it introduces the *Java agent* that must be included in the traced application's command line. What is more, the aspects need to be instantiated during the start of the traced system,

which slows down the start up of the application. Other methods of code instrumentation that can be researched include direct Java bytecode manipulation (e.g. provided by *Apache Commons BCEL* [6, 16] and *ASM* [8, 11, 36] libraries), creating an Akka extension or modifying Akka code to enable data collection mechanisms.

Appendix A. *Akka Tracing Tool*

This Appendix provides link to the repositories and describes their structure (section A.1) as well as provides a tutorial of how to include *Akka Tracing Tool* to an actor system (section A.2) and how to use visualization tool (Section A.3).

A.1 Repositories

The surce code of *Akka Tracing Tool* is available online on GitHub service [22] at the following address: `https://github.com/akka-tracing-tool`. At the time of creating the thesis, *Akka Tracing Tool* consists of 8 repositories:

- **akka-tracing-core** – the core part of the tool,
- **akka-tracing-docs** – the documentation of the library,
- **akka-tracing-sbt** – SBT plugin for Akka Tracing Tool,
- **akka-tracing-relational-database-collector** – relational databases collector for Akka Tracing Tool,
- **akka-tracing-couchdb-collector** – CouchDB collector for Akka Tracing Tool,
- **akka-tracing-examples** – examples of usage of the Akka Tracing Tool,
- **akka-tracing-tutorial** – simple project that explains how to use Akka Tracing Tool,
- **akka-tracing-visualization** – visualization tool for visualizing the traces.

A.2 Tutorial – how to use *Akka Tracing Tool*

This part of the appendix will be presented in a tutorial form – it shows how to include the library to a simple actor system.

First, you need to clone the repository containg an example actor system:

```
$ git clone \
 https://github.com/akka-tracing-tool/akka-tracing-tutorial.git
```

Downloaded project contains a very simple actor system shown in Fig. 16.

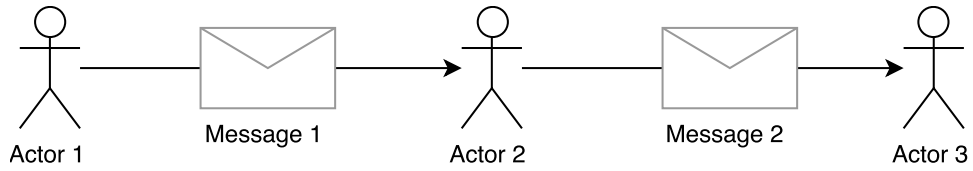

Figure 16: A simple actor system from downloaded project

The project structure is shown in Listing 1.

Listing 1: Project structure of the example

```
+ project
| - build.properties (1)
| - plugins.sbt (2)
+ src/main
| + resources
| | - akka_tracing.conf (3)
| + scala/pl/agh/edu/iet/akka_tracing/tutorial (4)
| | + actors
| | | - FirstActor.scala
| | | - SecondActor.scala
| | | - ThirdActor.scala
| | - Runner.scala
- build.sbt (5)
```

Below the most important files and directories are briefly described:

1. `build.properties` - contains information about SBT version.
2. `plugins.sbt` - convention says that here are put information connected with plugins. The library provides its SBT plugin for code instrumentation as well as automatic addition of the other library's dependencies to the project.
3. `akka_tracing.conf` - contains library's configuration for collecting traces, database connection with settings for it and package names that should be instrumented.
4. source code - all actor classes and runner are in proper directories according to packages. Additionally, every actor being traced has to mix trait `TracedActor`.
5. `build.sbt` - contains another important settings: database library dependencies and enabling Akka Tracing plugin.

A.2.1 Adding library dependencies to the project

Before instrumenting with our plugin the usual content of the `plugins.sbt` file is small and contains one line, as shown in Listing 2.

Listing 2: Usual content of `plugins.sbt` file

```
1 logLevel := Level.Warn
```

We need to use method `addSbtPlugin` to explicitly include plugin in proper version into the project. Below we need to provide URL path where our plugin is available. It's done by adding new resolver to Bintray's repository. The content of `plugins.sbt` after these additions is shown in Listing 3.

Listing 3: New content of `plugins.sbt` file

```
logLevel := Level.Warn

addSbtPlugin("pl.edu.agh.iet" % "akka-tracing-sbt" % "0.1")

resolvers += Resolver.url("Akka Tracing",
 url("https://dl.bintray.com/salceson/maven/"))(
 Resolver.ivyStylePatterns)
```

The next file to change is `build.sbt` file. The traces needs to be persisted somehow. For the scope of this tutorial, let's assume that SQLite database will be used for persistence. You need to add the following line to the project's dependencies:

```
"org.xerial" % "sqlite-jdbc" % "3.8.11.1"
```

This concludes the step of adding the library as project's dependency.

A.2.2 Library configuration

The file `akka_tracing.conf` contains main configuration settings for plugin and it's expected to be in resources directory. Of course, the name of the file is customizable: the only thing you need to change is the value of the key `aspectsConfigurationFile` in the plugin. The content of `akka_tracing.conf` is shown in Listing 4.

Listing 4: Contents of `akka_tracing.conf` file

```
akka_tracing {
 collector {
  className = "relational"
  database {
   profile = "slick.jdbc.SQLiteProfile$"
   db {
    driver = "org.sqlite.JDBC"
    url = "jdbc:sqlite:akka-tracing-tutorial.sqlite"
   }
  }
 }

```

```
13   packages = [
14     "pl.edu.agh.iet.akka_tracing.tutorial.actors"
15   ]
16 }
```

A.2.3 Mixing `TracedActor` trait

Every actor being traced has to mix `TracedActor` trait – it is necessary to provide messages correlation functionality. Initial implementation for one actor is shown in Listing 5.

Listing 5: Initial implementation for one of the actors

```
1 class SecondActor(val actorRef: ActorRef) extends Actor {
2   override def receive: Receive = {
3     case a =>
4       actorRef ! a
5   }
6 }
```

Firstly, you need to enable plugin in your project. It will include Akka Tracing Core dependency without manually adding it through `libraryDependencies` setting. It's done by this line in `build.sbt` file:

```
lazy val root = (project in file(".")).enablePlugins(AkkaTracingPlugin)
```

That's all changes you have to provide into your build configuration files! This plugin automatically include any remaining dependencies. Then, if you're using some kind of IDE, you should easily be able to import the `TracedActor` trait from the `pl.edu.agh.iet.akka_tracing` package. Next step is to mix that into actor class as shown in Listing 6.

Listing 6: Updated implementation for one of the actors

```
1 import pl.edu.agh.iet.akka_tracing.TracedActor
2 // (...)
3 class SecondActor(val actorRef: ActorRef) extends Actor with
      TracedActor {
4   override def receive: Receive = {
5     case a =>
6       actorRef ! a
7   }
8 }
```

A.2.4 Running the project

First, let's create the database that will be used for traces. You can do it by running the following command:

```
$ sbt initDatabase
```

Now you can run your application in the same way as previous: all necessary initializations depends on compilation phase and will be done automatically. Let's run application using SBT command:

```
$ sbt run
```

You can see that after database initialization a new file is created in the project root directory: `akka-tracing-tutorial.sqlite`. It contains data for SQLite database.

A.3 Using visualization tool

When you have all information in database, you can process the data and e.g. visualize them. On *GitHub* repository there is a simple tool for visualizing traces collected using Akka Tracing Tool. In order to use it, please clone the repository by running the command:

```
$ git clone \
 https://github.com/akka-tracing-tool/akka-tracing-visualization.git
```

Downloaded project contains Play application which can visualize traces in web browser. You need only to provide configuration file. Example is available in the repository under the name `database.conf.example`, as shown in Listing 7.

Listing 7: The contents of `database.conf.example` file

```
1  //Postgres configuration
2  profile = "slick.jdbc.PostgresProfile$"
3  db {
4    connectionPool = disabled
5    driver = "org.postgresql.Driver"
6    url = "jdbc:postgresql://localhost:5432/<DB_NAME>"
7    user = ""
8    password = ""
9    numThreads = 10
10 }
11
12 //SQLite configuration
13 profile = "slick.jdbc.SQLiteProfile$"
```

```
14 db {
15   driver = "org.sqlite.JDBC"
16   url = "jdbc:sqlite:<<FILE>>"
17 }
```

As we are using the SQLite database, we need to use the content as shown in Listing 8.

Listing 8: The contents of `database.conf` file

```
1 profile = "slick.jdbc.SQLiteProfile$"
2 db {
3   driver = "org.sqlite.JDBC"
4   url = "jdbc:sqlite:/path/to/tutorial/akka-tracing-tutorial.sqlite"
5 }
```

When all configuration changes are done you can run Play application:

```
$ sbt run
```

You can now open the URL `http://localhost:9000` in your web browser and see the visualization of the collected traces.

References

[1] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Artificial Intelligence, 1986.

[2] Akka Framework. `http://akka.io/`. Accessed: 2017-05-25.

[3] Akka.NET library. `http://getakka.net/`. Accessed: 2017-06-13.

[4] Akka Tracing Tool. `https://github.com/akka-tracing-tool/`. Accessed: 2017-05-25.

[5] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide Time to Relax*. O'Reilly Media, Inc., 1st edition, 2010.

[6] Apache Commons BCEL library. `https://commons.apache.org/proper/commons-bcel/`. Accessed: 2017-06-17.

[7] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

[8] ASM library. `http://asm.ow2.org/`. Accessed: 2017-06-17.

[9] Amazon Web Services. `https://aws.amazon.com/`. Accessed: 2017-06-01.

[10] H. Baker and C. Hewitt. Laws for communicating parallel processes. *MIT Artificial Intelligence Laboratory Working Papers*, 1977.

[11] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

[12] M. Ciołczyk and M. Wojakowski. *Tool for monitoring of distributed applications in Akka toolkit*. BSc thesis, AGH University of Science and Technology, 2016.

[13] W. D. Clinger. Foundations of actor semantics. *AITR-633*, 1981.

[14] Lightbend's config. `https://github.com/typesafehub/config`. Accessed: 2017-06-01.

[15] CouchDB Database. `http://couchdb.apache.org/`. Accessed: 2017-05-28.

[16] M. Dahm. Byte code engineering. In *JIT'99*, pages 267–277. Springer, 1999.

[17] Eclipse AspectJ. `https://eclipse.org/aspectj/`. Accessed: 2017-05-31.

[18] Erlang language. https://www.erlang.org/. Accessed: 2017-06-13.

[19] Erlang Performance Lab Toolkit. http://www.erlang.pl/. Accessed: 2017-06-13.

[20] ExplorViz Tracing Tool. https://www.explorviz.net/. Accessed: 2017-06-13.

[21] F. Fittkau. *Live Trace Visualization for System and Program Comprehension in Large Software Landscapes*. Number 2015/7 in Kiel Computer Science Series. Department of Computer Science, Kiel University, Dec. 2015. Dissertation, Faculty of Engineering, Kiel University.

[22] GitHub. https://github.com/. Accessed: 2017-05-28.

[23] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[24] I. Greif. Semantics of Communicating Parallell Processes. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[25] M. Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[26] C. Hewitt. Message Passing Semantics. Technical report, DTIC Document, 1981.

[27] C. Hewitt, P. Bishop, I. Greif, B. Smith, T. Matson, and R. Steiger. Actor Induction and Meta-evaluation. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 153–168, New York, NY, USA, 1973. ACM.

[28] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[29] M. Janczykowski. *Scalable method for urban traffic simulation*. MSc thesis, AGH University of Science and Technology, 2017.

[30] Java language. https://www.java.com. Accessed: 2017-06-13.

[31] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring Distributed Systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, Mar. 1987.

[32] Kamon Metrics Collection Library. http://kamon.io. Accessed: 2017-06-13.

[33] L. Khomich. Akka Tracing Library. `https://github.com/levkhomich/akka-tracing`. Accessed: 2017-06-13.

[34] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, 2001. Springer-Verlag.

[35] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir Performance Analysis Tool-Set. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, pages 139–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[36] E. Kuleshov. Using the ASM framework to implement common Java bytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.

[37] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[38] S. Moore, D. Cronk, K. London, and J. Dongarra. Review of performance analysis tools for MPI parallel programs. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 241–248. Springer, 2001.

[39] A. Nataraj, A. D. Malony, A. Morris, D. C. Arnold, and B. P. Miller. A framework for scalable, parallel performance monitoring. *Concurrency and Computation: Practice and Experience*, 22(6):720–735, 2010.

[40] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696 – 710, 2009. Best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007).

[41] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.

[42] Pykka library. `https://github.com/jodal/pykka`. Accessed: 2017-06-13.

[43] Scala language. `http://www.scala-lang.org/`. Accessed: 2017-05-25.

[44] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[45] Slick library. `http://slick.lightbend.com/`. Accessed: 2017-06-01.

[46] Apache Spark – Lightning-fast cluster computing. `http://spark.apache.org/`. Accessed: 2017-06-14.

[47] M. Thurau. Akka framework. *University of Lübeck*, 2012.

[48] Travis CI. `https://travis-ci.org/`. Accessed: 2017-05-28.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95, 2010.

[50] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward Scalable Performance Visualization with Jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, 1999.

[51] Zipkin Tracing System. `http://zipkin.io/`. Accessed: 2017-06-13.