AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF COMPUTER SCIENCE,
ELECTRONICS AND TELECOMMUNICATIONS

TOOL FOR MONITORING OF DISTRIBUTED APPLICATIONS IN AKKA TOOLKIT

DEVELOPMENT PROCESS DOCUMENTATION

Michał Ciołczyk
Mariusz Wojakowski

TUTOR
Maciej Malawski, Ph.D.

KRAKÓW 2016

Table of Contents

1 Project goals and vision
- 1.1 Main project goal
- 1.2 The actor model
- 1.3 Message-passing semantics
- 1.4 Debugging and tracing
- 1.5 The need for tracing
- 1.6 Illustrative example
- 1.7 Product goals
- 1.8 Discussion of problems and their solutions

2 Feasibility study
- 2.1 Risk analysis
  - 2.1.1 Need to learn new technologies
    - Possible solution
  - 2.1.2 Integration with existing collectors
    - Possible solution
  - 2.1.3 Too ambitious client’s requirements
    - Possible solution
  - 2.1.4 Research oriented nature of the project
    - Possible solution

3 Adopted methodology of work
- 3.1 Project management tools
- 3.2 Meetings with client and manager
- 3.3 Roles and tasks division

4 Verification of project’s results

5 Work progress - milestones
- 5.1 Description of milestones
  - 5.1.1 Research and prototyping
    - “Pure Akka” implementation
    - Bytecode manipulation
    - Aspect-oriented programming (AOP)
    - Milestone summary
  - 5.1.2 Preliminary implementation of tracing using aspect-oriented programming
    - Milestone summary
  - 5.1.3 Working project prototype
    - Milestone summary
  - 5.1.4 Visualization tool
    - Milestone summary
  - 5.1.5 SBT plugin
    - Milestone summary
  - 5.1.6 Documentation and tutorial with working library and example
    - Milestone summary

6 End product evaluation

7 Future works

References

1 Project goals and vision

1.1 Main Project goal

The goal of the Project is to provide a tool for monitoring of distributed applications in actor model in Akka framework. We begin with the short introduction to the basics, e.g. the actor model, message-passing semantics and practical problems with debugging distributed systems. Based on this introduction we define our problem precisely and describe in more details the tool that we will be creating.

1.2 The actor model

The actor model in computer science is a mathematical model of concurrent computation that treats ‘actors’ as the universal primitives of these computations. In response to a message that it receives an actor can:

- make a local decision,
- create more actors,
- send messages to other actors,
- determine how to respond to the next message received.

In Figure 1 you can see a simple diagram of actor system and how actors communicate with each other.

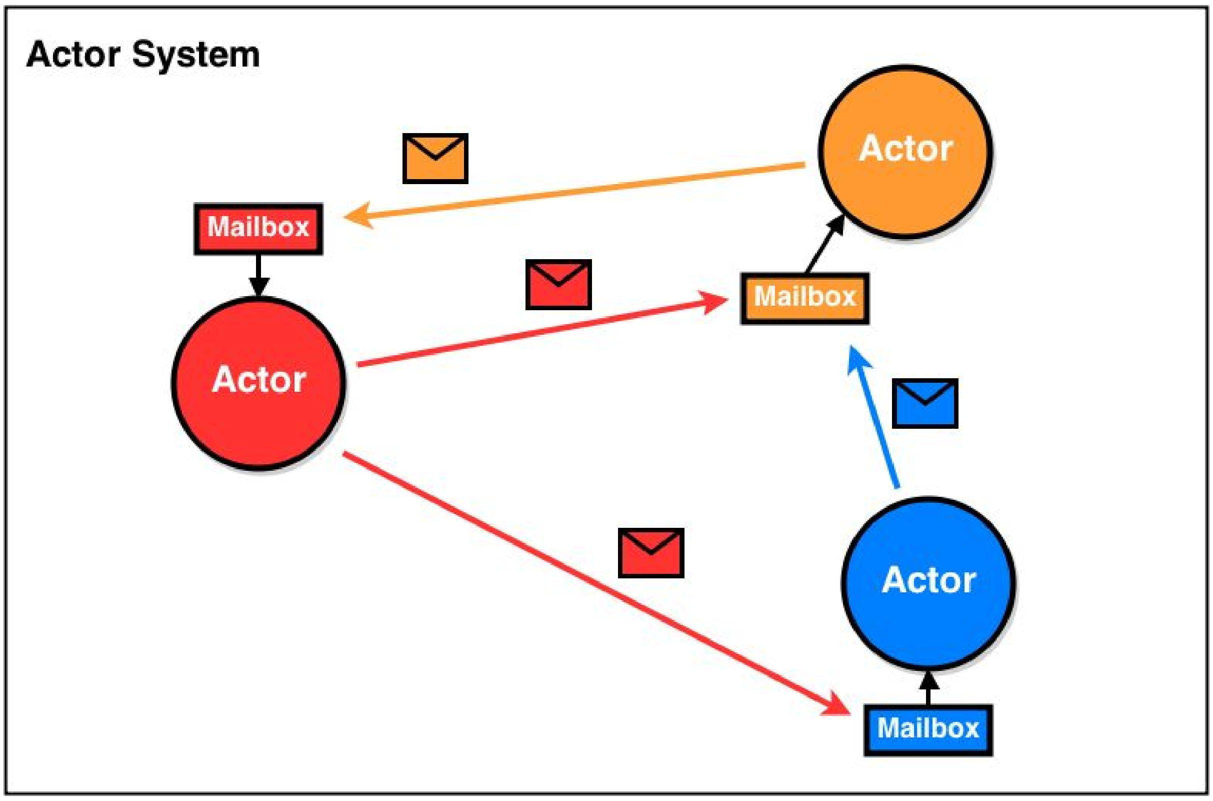


Figure 1. Simple diagram of actor system model (source: [1])

The actor model originated in 1973 and was proposed i.a. by Carl Hewitt [2]. Its development was motivated by creating highly parallel computing machines consisting of huge amount of independent processors that use them all.

The actor model has been used both as a framework for a theoretical understanding of computations and also for practical implementation of concurrent systems.

1.3 Message-passing semantics

The actor model is is very strictly connected with message-passing semantics. Message-passing programming paradigm assumes many instances of process work in sequential mode and they can communicate by sending messages to each other. In pure message-passing these computing units don’t share any data between them. However, programs written using this semantics can be run on architecture that supports a different computational model. For more information about message-passing semantics, please see S.S. Kadam’s presentation [3].

One of the most important distinctions among message passing systems is whether they use synchronous or asynchronous message passing: synchronous message passing is what typical object-oriented programming languages such as Java and Smalltalk use. Asynchronous message passing requires additional capabilities for storing and retransmitting data for systems that may not run concurrently.

In Figure 2 you can see messages passing between actors A and B, every of them is working in sequential mode and don’t share anything with external actors:

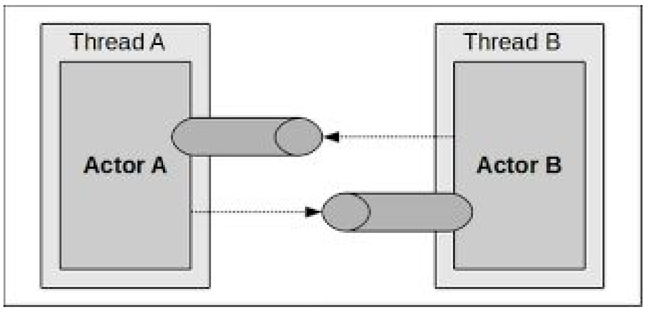


Figure 2. Messages passing between actors A and B (source: [4])

Message-passing semantics is easy to reason about and simpler than another existing computational models used to create complex, distributed systems.

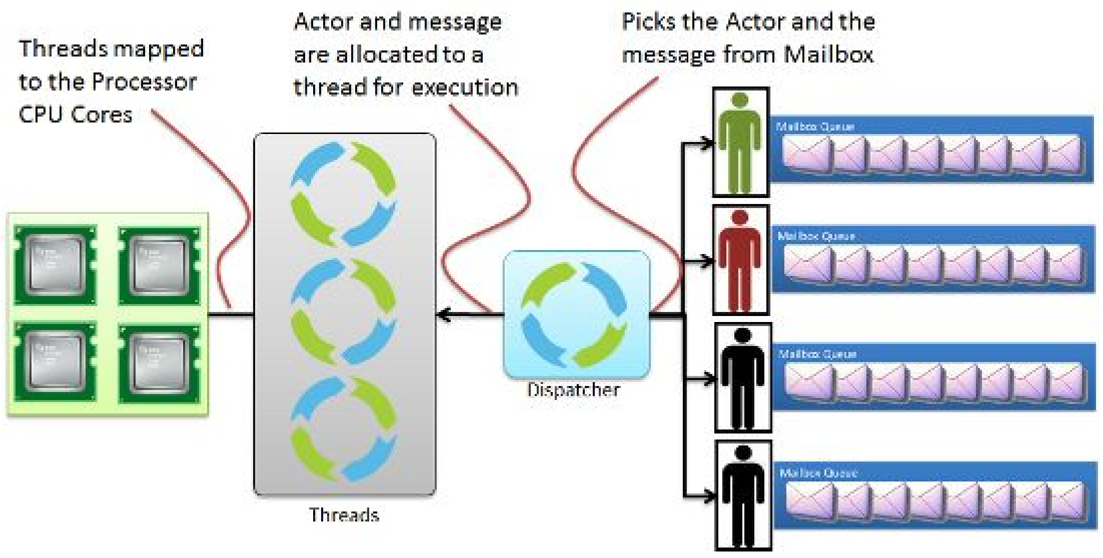
1.4 Debugging and tracing

Developing applications is a very complex process and consists among other things of testing. It is very important, because of that you can increase the quality of codebase and ease future maintenance. Various types of metrics were established to measure tests coverage. But even 100% of coverage can’t make you sure that your application doesn’t contain any bugs and flaws. That’s why we need debugging - sometimes you don’t cover every case in the system and you have to check what causes a bug and find it.

Developing a complex system requires usage of different debugging techniques: from manually instrumenting code by logging information on *standard output* up to usage of sophisticated tools that allow you to stop, check variable states and even evaluate expressions.

In a sequential environment using and reasoning about results of such a tool are obvious and intuitive. Problems mount up when we meet multiple threads: it's not possible to predict interlacing of different processes. There are debuggers for multi-threaded applications but complexity of using them is much greater. Ok, so what will happen when we move to the actor model? Actors are independent: they process messages sequentially in separation of any others. Everything is asynchronous and you can't easily connect sent message with it actual processing in receiver. This is also connected with a dispatcher - every message can be dispatched in different threads so it's hard to follow that. It's getting even more complicated when you run your application on multiple JVMs.

In Figure 3 you can see how actors' code are dispatched and prepared for execution on CPU processors:

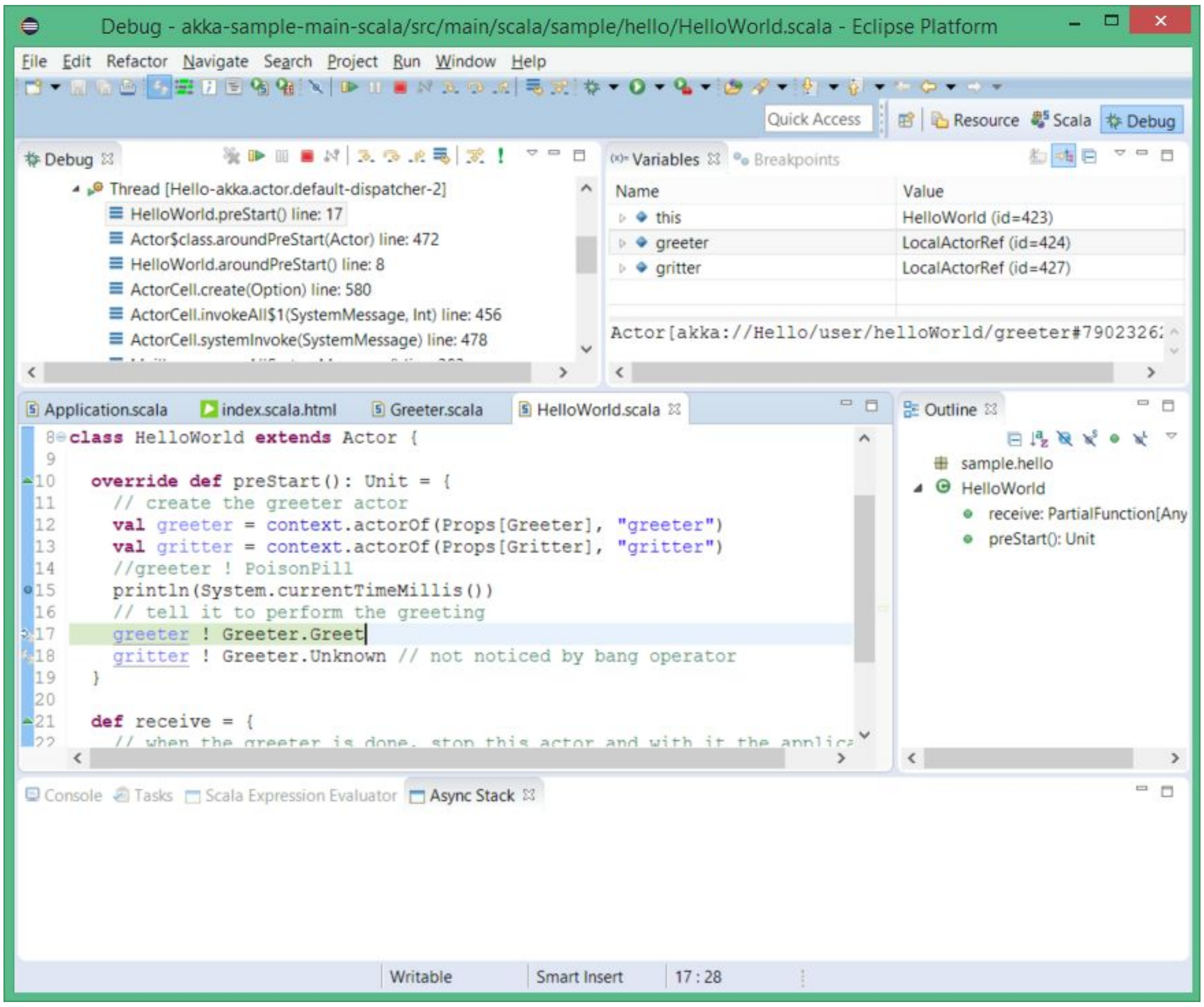


Threads mapped to the Processor CPU Cores
Actor and message are allocated to a thread for execution
Picks the Actor and the message from Mailbox
Mailbox Queue
Mailbox Queue
Mailbox Queue
Mailbox Queue
Dispatcher
Threads

Figure 3. Diagram of dispatching and executing actors' code on threads (source: [5])

Debugging in one actor is relatively easy (because it's sequential), but what happens when we want to move to another actors that will be receiving that message? Well-known in standard debugger button "Next line/Step into" is not really useful in such systems. One proposal for a convenient tool was suggested by Iulian Dragos: an async debugger [6]. He pointed out that debugging that kind of applications is like being a detective: you need to go from effects back to causes, attempt a fix, and check whether everything works fine. Threads and locks are replaced by higher-level abstractions like futures, actors and parallel collections which use a thread-pool for executing their computations. He also suggested that instead of 'Step into' we should add ability to 'step-with-message' - this can happen in a different thread or even these two actors can exchanged different messages in the meantime! It's not important for us: we want to move to the next logical step which is receiving the message.

As you can see below, in Figure 4, there is an implementation of that behaviour. Currently, it is available as a feature in Scala IDE.


Debug - akka-sample-main-scala/src/main/scala/sample/hello/HelloWorld.scala - Eclipse Platform
File Edit Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Scala
Debug
Debug
Variables
Breakpoints
Application.scala
index.scala.html
Greeter.scala
HelloWorld.scala
Outline
Console
Tasks
Scala Expression Evaluator
Async Stack

```
Thread [Hello-akka.actor.default-dispatcher-2]
    HelloWorld.preStart() line: 17
    Actor$class.aroundPreStart(Actor) line: 472
    HelloWorld.aroundPreStart() line: 8
    ActorCell.create(Option) line: 580
    ActorCell.invokeAll$1(SystemMessage, Int) line: 456
    ActorCell.systemInvoke(SystemMessage) line: 478
```

Name	Value
this	HelloWorld (id=423)
greeter	LocalActorRef (id=424)
gritter	LocalActorRef (id=427)

```
Actor[akka://Hello/user/helloWorld/greeter#7902326
```

```scala
class HelloWorld extends Actor {

  override def preStart(): Unit = {
    // create the greeter actor
    val greeter = context.actorOf(Props[Greeter], "greeter")
    val gritter = context.actorOf(Props[Gritter], "gritter")
    //greeter ! PoisonPill
    println(System.currentTimeMillis())
    // tell it to perform the greeting
    greeter ! Greeter.Greet
    gritter ! Greeter.Unknown // not noticed by bang operator
  }

  def receive = {
    // when the greeter is done, stop this actor and with it the applic
```

```
sample.hello
HelloWorld
    receive: PartialFunction[Any
    preStart(): Unit
```

```
Writable    Smart Insert    17 : 28
```

Figure 4. A screenshot from Scala IDE - in debug frame you can see green 'bang' icon that allows using *step-with-message* ability (source: [7])

1.5 The need for tracing

Very often during debugging it is helpful to have information about the whole message path. Tracing allows developers to inspect what causes unwanted behaviour. Sometimes with tracing developers can gather statistics and test performance of application. Full tracing is very expensive process so it's reasonable only for development time. However, tracing is possible in production environment combined with sampling.

Information gathered through tracing developers can use in many ways:

- espy unwanted message and check its sender,
- check message correctness,
- filter by actors, message types or contents,
- gather different aggregate statistics.

Main idea for the product that is the subject of this Project was a tool that enables developers to ease debugging of application with the usage of traces and message flow. Below you can see an example of situation where such tool could give us sensible information about what happened.

1.6 Illustrative example

In Figure 5, you can see that problem depicted in a simple diagram: An actor indicates the existence of the problem by e.g. throwing an exception. The cause of this error was in Actor 4 processing a faulty message originated by Actor 2 which is 2 interactions before. The stack trace of Actor 4 shows only information about current actor's context. However, we would like to know what really caused this problem.

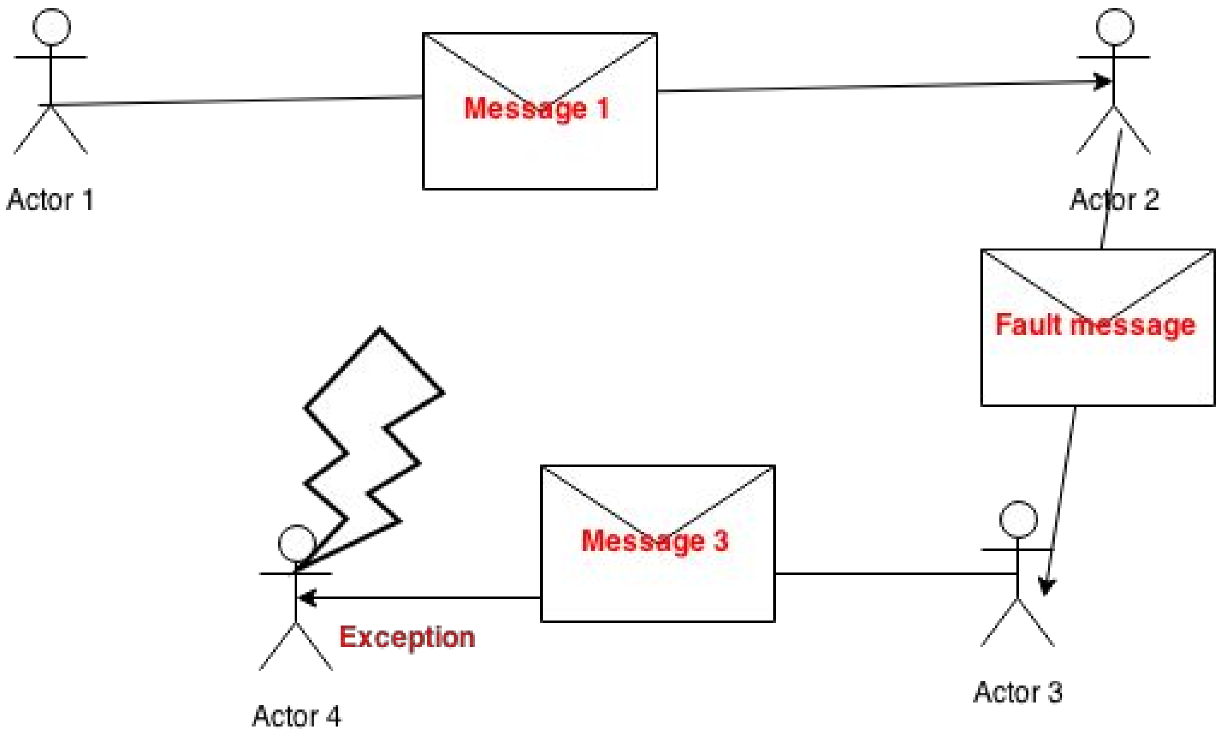


Figure 5. Messages flow that causes an unwanted behaviour

1.7 Product goals

We would like to have a tool that:

- enables the user to collect and see the messages passed between actors in a user's actor system,
- provides a way easily enable/disable tracing,
- requires minimum workload for programmers to use it,
- does not require drastic changes in the actor system that user wants to trace,
- should not have large impact on the performance of the traced system,
- provides a way to visualize collected traces either through some external tool or built during the development process.

1.8 Discussion of problems and their solutions

In the scope of the work we encountered several problems that we needed to solve:

- **integration with existing collectors and visualization tools** - we knew about very good, production tested systems for distributed tracing, but it could turned out that integration would be very hard or even impossible due to differences between data formats or gathered informations. We also didn't want to run against these systems - we wanted to create something a little bit different than existing implementation. We appreciated that we had to create our own version of collector.
- **client requirements** - they could be too strong, which means that gathered tracing data may be not helpful at all. We had to figure it out and check frequently whether this product introduces real value into developing.
- **more research topic than some practical project**, so it demanded from us more prototyping than coding in the initial phase. We had to very carefully create schedule and stick to it, because it was so easy to not meet client requirements.
- **simplicity of usage** - it is very important for developers: they don't want to spend a lot of time configuring and doing manual hand-work to getting things done. We've struggled to gain best results in this field. Compared to Akka-tracing we try to collect message flows in generic way without specific hints and instrumentation in codebase, so-called *zero application specific hints instrumentation.* Although we know that in specific case it's very useful to manually define additional data to log but we haven't considered it much - it can be done in future works.

2 Feasibility study

The feasibility study was the first phase of our Project. The process of creating a tool that is the subject of the Project was associated very strongly with potential problems, which could cause delays and even lead to situations where it was impossible to end the work with existing requirements. We knew that this phase of our Project is very important and we've tried to predict most of the problems and minimize the possibility of failure and prepare best foundation for future work.

In this phase we gained confidence on technologies we wanted to use and made sure that our task and plan was somehow feasible. We prepared some prototypes and after that we chose *AspectJ* for instrumentation from other propositions: bytecode manipulation and 'pure Akka' library. We also investigated that other tracing framework also use this type of instrumentation so we were sure that this was doable. Also in this phase we developed a prototype version of the architecture and format of configuration file.

We were thinking about testing strategy too. Testing such library which operation depends heavily on the system using it is hard. Also we knew that testing it on big, complex systems was not going to work. We decided that we would focus on the acceptance tests of some small systems, some deterministic scenarios.

2.1 Risk analysis

Below is a list of points, that we knew at the beginning could cause problems. With every potential problem we tried to take steps that allow us to minimize failure of failing deadline. As a result every identified risk is associated with a short description of what we planned at the beginning to prevent it from happening.

2.1.1 Need to learn new technologies

In this case the choice was imposed by product requirements. We were aware that we would spend a certain amount of time intended for work on learning Scala language and Akka toolkit.

Possible solution

Learning new technology, new language always takes some time and we planned to spend some at the beginning on that. Each of us had contact with these technologies so we thought it wouldn't be a big problem - in the initial phase we could work on designing and prototyping part concurrently with learning and gaining knowledge in these areas.

Also our client was very familiar with mentioned technologies so we could always ask him about any advice and presenting some problems with hope that he at least pointed a proper way to solve them.

Each of us also had very big motivation to learn these technologies. We didn't see that as an obstacle - it was a big opportunity combined with great incentive in the form of the final evaluation of engineering work.

2.1.2 Integration with existing collectors

At the beginning we planned integration with existing collector, e.g. Zipkin [8]. It was difficult to predict how much time it would take because of unspecified product requirements and unclear goals yet. Earlier mentioned Zipkin provides production-ready system for tracing, querying and visualizing flows so we wanted to use it very willingly. We also knew that this integration might be very hard or even impossible at some point.

Possible solution

It is very hard to predict something when you even don't know what exactly your tool will be doing. Only sensible thing we could do was to create a plan that took into account potential integration with existing collector. We safely assigned 1 iteration time for that. That time covered: deep recognition of chosen collector, introducing inevitable changes into code and troubleshooting.

Integration with existing collector wasn’t our high priority - we thought only that we could easily benefit from production-tested tool that proved its usefulness and provided nice out of the box system for visualization.

2.1.3 Too ambitious client’s requirements

Simplicity of usage was an important goal in our Project and we tried to meet client’s requirements fully. But we also looked from the perspective of developers and we were thinking about potential limitations. Client requirements could be too strong and too generic, which means that gathered data may be not helpful at all. We had to figure it out at the phase of designing as well as initial coding.

Possible solution

This risk was intrinsically connected with research - we only knew that we had to figure out between simplicity of usage and usefulness of our tool. To avoid working fruitlessly we planned to meet often with our client and discuss every part of that Project with him. We knew that experienced person could give us good clues and provided invaluable help.

We also planned to ship first version of product very quickly - it would be very big help to gain feedback from first versions and focus on more important parts from the point of client view.

2.1.4 Research oriented nature of the Project

The subject of this Project is more connected with research than implementing product based on well-defined requirements collected from client. We were aware that it could impact risk of not meeting deadlines. Creating prototypes forced us to spend some time on learning new technologies or acquire knowledge in different way. We knew that wrong estimates and not sticking to the plan could cause very serious problems during work.

Possible solution

It demanded from us more prototyping than coding in the initial phase. The only proper way to avoid problems was to very carefully create a schedule and stick to it. Not meeting client’s requirements was so easy in our case and we didn’t want to fail because of that. Frequent meetings also would help us receiving regular feedback from client and react to it by changing plan or leaving irrelevant part of work for subsequent iteration.

3 Adopted methodology of work

The first part of the development was actually not writing the code - it consisted of comprehensive research about different ways of instrumentation that our library could use. Due to the research part, it couldn't be done as an iterative or incremental process. There were some possible solutions that were just doomed to fail. Therefore, we decided to adopt the throw-away prototyping in this part of the Project. It did splendidly - we could determine the pros and cons of each option and decide which way of instrumentation showed promise to make it work. This enabled us to efficiently research the best approach to solve our problem. The first part - research and prototyping last for about 2 months and resulted in a few of prototypes and proofs of concepts that allowed us to decide which approach would be used in our Project.

The rest of our work (about 7 months) was actually done in an incremental process in agile spirit. First we tried to divide our work in equal or nearly equal parts. Unfortunately, what really happened was fact that even though tasks were divided in a good way, different tasks required different amount of work (even though we tried to estimate it). Due to this fact, the iterations were very irregular. Therefore we think that our work should not be seen from iterations perspective but from milestones perspective. We established some main goals to do in each development step and every development step corresponded to different milestone. Therefore we think of the incremental process not as divided by iterations but by milestones and we'll describe it in this way.

A milestone is a specific moment of time that, when reached, new major functionality is delivered to the project or major project's development step is completed. The time between two milestones can vary - this is the opposite approach to the iterative development where iterations must last the exact or nearly the same amount of time (e.g. every iteration lasts 2 weeks). The milestone is focused on delivering strictly specified functionality whereas iteration, while having its goals, mainly focuses on the tasks, but often not on the client's requirements.

We've chosen the milestone approach due to the fact that it was extremely hard to predict the amount of time to complete the tasks (due to the research topic and their complexity) and because of the simpler development process management due to not having to split the tasks into somehow artificial iterations. It also allowed us to focus on delivering the functionality and meeting the requirements that was crucial to our client.

3.1 Project management tools

We used a few tools to help us in managing our Project. The main tool for storing all information regarding the decisions, meetings and other important aspects of our library is Atlassian's Confluence - a wiki for IT projects. It was used in every step of the Project's development. It was the place where we documented the progress in our work as well as client's requirements and our architecture decisions. The tool's functionality helped a lot with planning and documenting the development progress.

The other tool, which helped us with time management as well as the division of tasks, was another Atlassian's product - JIRA. This tool enabled us to plan the future development steps and divide the tasks between us. It was the place where we went daily to check what we have to do next, what is actually done in the current milestone and what else is to do. The functionality to assign team member to certain task helped a lot in the tasks' division and enabled us to just look and see what each member is responsible for.

Other tools that were useful (especially for presentations and documentation) were Google Drive and Google Docs. These two products are closely related - Google Drive is actually a disk space where you can create Google Docs - collaborative documents, spreadsheets and presentations. We used these tools to create various presentations and documents (one of which you are reading right now). Its functionality to work together allowed the team to prepare needed documents much faster than it would be possible with e-mail-driven communication with attached files.

The other tool that we used was GitHub. It is a service that allows you to store and version the code. We used it to collaborate during the coding phase - every part of code that we created was stored on some repository on GitHub. Every piece of code was also reviewed by other team member during the "Code review" with the usage of GitHub's pull requests. During this process the other team member (not the author of the code) read the code to try and spot mistakes and maintain the quality of code

The last tool that we used is a service called Travis. It is a Continuous Integration platform. It's closely related to GitHub and extremely easy to set up - that's why we decided to introduce it. After each commit the code was sent to some machine to compile and run unit tests. The results of this process was clearly visible to the team members. If something failed, it was a sign that the commit's author has made some mistakes and he should correct them - it is great to use because it's fully automatic - there is no need to manually run tests or compile the project. We introduced this tool after we had some core functionality written so it wasn't used from beginning. It was due to the fact that at the beginning we had no functional library and it was pointless to use due to the inability to actually compile and test anything.

3.2 Meetings with client and manager

During the development and research we met with client on regular basis. During the first part when we learned new technologies and researched different approaches to solve our problems, we met with client every two weeks. Every meeting was very helpful as we could establish the current priorities in our work. Later we met with client every week to present the finished tasks and the current status of work.

Meetings with manager were more irregular. The frequency was adopted to the current needs. These meetings were rather oriented on the software development process itself. Each was useful as we could tweak our methodology and task management to help us deliver the effects quicker.

Each meeting were documented on Confluence. In this way, we could remember the conclusions and important decisions that were established during the meetings. It also helped us to have the development process itself more organized. Because of the frequency of the meetings (especially with client) we could be flexible to the current needs.

3.3 Roles and tasks division

The Project was developed by two team members. The tasks were divided in a fair, equal way that each person from the team accepted. We could distinguish following roles in our Project with their responsibilities:

- Business analyst - meetings with client, establishing the requirements.
- Architecture designer - designing the library's architecture.
- Programmer - implementing the library's source code as well as the unit tests.
- Tester - testing the library on existing use cases.

As the roles are quite flexible (being a tester didn't mean that you can't be a programmer), each person actually was responsible (at least at some part) in the tasks of every role. We actually divided the work based on the tasks that we had to do, not based on the different roles as it was quite impossible due to the small number of team members.

In Table 1 we can see the detailed tasks division between the two developers during the whole development process.

Michał Ciołczyk's tasks	**Mariusz Wojakowski's tasks**
● Research about "pure Akka" and bytecode manipulation methods of instrumenting library. ● Database connectivity tasks using user provided connection definition. ● Implementation of the Visualization tool. ● Implementation of the SBT plugin with new aspect generation method. ● Development Process Documentation (Sections 3-7). ● Technical Documentation (Sections 1.3, 2, 4). ● Various bug fixes.	● Research about aspect-oriented programming. ● Implementation of the aspects. ● Implementation of the preliminary aspects generation. ● Implementation of the message contents' persistence. ● Examples and tutorial. ● Tests. ● Development Process Documentation (Sections 1-2). ● Technical Documentation, Sections 1.1, 1.2, 3). ● User Documentation.

Table 1. Tasks division between developers

4 Verification of Project's results

Developing a library which operation is very dependent on the application that it will be connected to is very hard - not only because we don't control what user will actually do, but also due to the fact that it's very difficult to verify if the library is actually doing the things that it actually should be doing.

Due to this fact, we created a few examples of simple actor systems that could verify that our library is actually working. We also created a very simple visualization tool that enabled us to visually see if the traces that we were collecting were correct. There was simply no other way to test whether the library is working properly.

Of course, we also used unit tests that enabled us to check if the different parts of our library were working in expected way. In this way we confirmed that different parts of the project were actually working. For example, we verified if the configuration parser read the configuration file properly; if the database initialization mechanism created correct tables in database and many more other things.

A few of the tests actually could not be automated and therefore we had to verify the results ourselves. These tests were mainly connected to the non-functional requirements - for example the machines cannot check if something is easy to use. Therefore, we relied on the client's opinion as well as our own thoughts about the matter. The integration tests couldn't be easily automated as well as they would require an environment that could interact between actor system and a tested library. Therefore, in more sophisticated examples, we needed to check ourselves if the produced traces are correct.

Both the automated and manual tests confirm that the requirements have been met by our library. The library and created tools are working in the expected way and do allow users the functionality that was assumed that they would be providing.

The automated tests are thoroughly described in the Technical Documentation of our library in Section 3.

5 Work progress - milestones

As we said earlier, we don't actually see the work progress from iterations or time perspective as the iterations could be very irregular due to the fact that the tasks' complexity and difficulty were very varying. Therefore, we decided to present our work progress from milestones perspective - the main goals that were completed with each development step. We have distinguished the following milestones:

1. **Research and prototyping** (completed in 2 months) - this milestone was about researching different approaches to the instrumentation of the library in the user's code.

2. **Preliminary implementation of tracing using aspect-oriented programming** (completed in 1 month) - during this iteration we implemented (working only on specific project) first version of the library.
3. **Working project prototype** (completed in 1 month) - during this iteration we implemented working version of the library with manually inserted instrumentation code.
4. **Visualization tool** (completed in 2 weeks) - this short iteration was solely dedicated to create very simple tool to help us see if the traces we were collecting were correct. It's not a main or necessary part of the product but rather some side product that were required due to the need of verification of the results.
5. **SBT plugin** (completed in 2 months) - in this iteration we implemented the automatic generation of aspects which enabled us to greatly reduce the user's actions to instrument the code.
6. **Documentation and tutorial with working library and example** (completed in 2 months) - this part was dedicated solely to the documentation and manuals for the library's user - we created a simple small example for user that demonstrates how to connect our library to some existing project.

5.1 Description of milestones

5.1.1 Research and prototyping

This milestone had two main goals:

- Make a decision on technologies and other important technical aspects:
    - Language and IDE,
    - Code versioning tool,
    - Management tools,
    - Process methodology.
- Research of different approaches of how to implement our library.

The following decisions were made:

- Language: Scala.
- IDE: IntelliJ IDEA.
- Building tool: SBT.
- Code versioning tool: Git.
- Management tools: JIRA, Confluence, GitHub.
- Test framework: Scalatest.
- Process methodology: throw-away prototyping (research phase), incremental (coding phase).

The research part was mainly dedicated on finding the way that we can instrument the library's code in user's actor system. The main three that we considered were:

- “pure Akka” implementation,
- bytecode manipulation,
- aspect-oriented programming.

We considered each approach and created some very basic prototypes of how it could be implemented using different technology.

“Pure Akka” implementation

This method was checked first. It was caused by obvious reasons:

- no external dependencies,
- prospective minimize amount of code necessary to instrumentation.

At the initial phase of work we couldn’t determine whether it was achievable in intended by us way so we decided to spend some time of the research phase on initial version of implementation. Moreover Akka toolkit was known by us only from the developer point of view and API, so we had to spend a certain amount of time on deep investigating and getting oriented in internal implementation of Akka.

To use this method we needed to deliver own version of actors’ provider and add new behaviour to existing implementation, which is sending information about relation between messages. At that moment we didn’t know how to do it in proper way. Also providing another implementation seemed for us as not best idea. We decided that we wouldn’t follow that way.

Bytecode manipulation

This idea appeared for obvious reasons: Scala is a language that is compiled to Java bytecode that enables it to run on JVM. We tried to use *javassist* library to modify the bytecode after compilation stage. However complexity of this method sowed doubts whether we could do this while meeting the deadlines. We can imagine existence of such solution, but for us it was too complicated and we couldn’t undertake this method - the risk of failure was too large.

Also we didn’t know how to instrument code without interference into Akka toolkit - we preferred not to change its compiled version of classes.

Bytecode manipulation is undoubtedly the most difficult solution, characterized by high complexity and demanding thorough knowledge of JVM, although the resulting product could be very efficient with low overhead of executed instructions.

Aspect-oriented programming (AOP)

This idea appeared for several reasons. One is that it required the least workload of all described methods. Aspects are very quick to write and very powerful. It is also easy to instrument them into the code. The other reason is that there were some successful tries at tracing actors using this instrumentation method (for example Kamon uses aspects to instrument the library’s code).

These reasons were undoubtedly the deciding factors in the choosing of the instrumentation method in our library. We decided that aspect-oriented programming seemed to be the most promising and the easiest of the methods that we were considering to use.

Milestone summary

In this milestone the decisions about the Project’s technical and organizational aspects were made. We established management tools to work with as well as language and frameworks that we’ll be using while implementing the library.

We also made important decisions regarding the most difficult aspect of our library - code instrumentation. We decided to use aspects to instrument our code as they seemed to be the most promising of the three methods that were taken into consideration.

5.1.2 Preliminary implementation of tracing using aspect-oriented programming

This milestone focused mainly on creating the first meaningful and working version of aspect template needed for tracing. We had to learn thorough specific parts of aspect-oriented programming to use it in our library and it took major amount of time of the iteration.

The previous milestone showed us that it was possible to do using aspect-oriented programming. However, we didn’t know how to properly handle correlation of messages. In this preliminary implementation emerged the idea of keeping information in trait mixed into actor’s object. It turned out that this implementation is good enough and it stayed unchanged up to the last version.

Product of this iteration allowed us to manually include this aspect into project and see on output intercepted messages which could be easily connected into traces.

Milestone summary

What was done:

- Meaningful version of aspect template
- Correlation of messages into traces using trait implementation

What remained unanswered was:

- How to make whole process easier for developers and how to include it into build definition?
- In what form should we persist information about traces?

5.1.3 Working project prototype

This milestone was focused on delivering working project prototype to our client. In the previous milestone we have successfully created aspects that we could use to trace the messages (without storing the information). There were some issues on how to generate the aspects so they work in any project. Requiring that user would write his own aspects was definitely not a way to increase the usability and easiness of integration the library with user's projects. Therefore, we needed some way to actually automatically create aspects that would collect information about messages as well as persisting that information to the database.

We also needed to make decision about the persistence library. Due to the Slick's popularity and simplicity we decided to use it to implement persistence of traces into relational database. We chose PostgreSQL as the database that will be used for the time being due to the familiarity with this database engine.

This milestone resulted in a very preliminary way of aspect generation. We included some code into the build definition file of the testing example. Therefore, the aspect was actually generated during the build phase and user didn't have to provide the aspects on his own.

We also worked a little more on the correlation of messages so they could be connected into traces. Previously there were some problems in some use cases. In this milestone we fixed these problems.

Milestone summary

What was done:

- Aspects generation based on the code inserted into user's project build definition file. Unfortunately, it required a lot of code to be inserted there which wasn't user friendly.
- Persistence of the traces in database.
- Correlation of messages fixes for some fail cases.

What remained unanswered was:

- How to check if the library is really producing good traces?
- How to reduce the amount of code inserted to the build definition file?

5.1.4 Visualization tool

This milestone was dedicated solely on implementing a very simple tool that could allow us to actually view the traces collected by aspects and persisted in the database. We didn’t expect this tool to actually be a part of our Project but, unfortunately, attempts of integrating our library with existing visualization engines failed and we needed something very quickly.

Therefore we decided to build a very simple visualization tool that would enable to see the traces as directed graph. The nodes of the graph were actors and the edges were passed messages. The tool was written using Scala API of the Play! framework, Slick and JS graph visualization library called Sigma JS. We present screenshot from the visualization below.

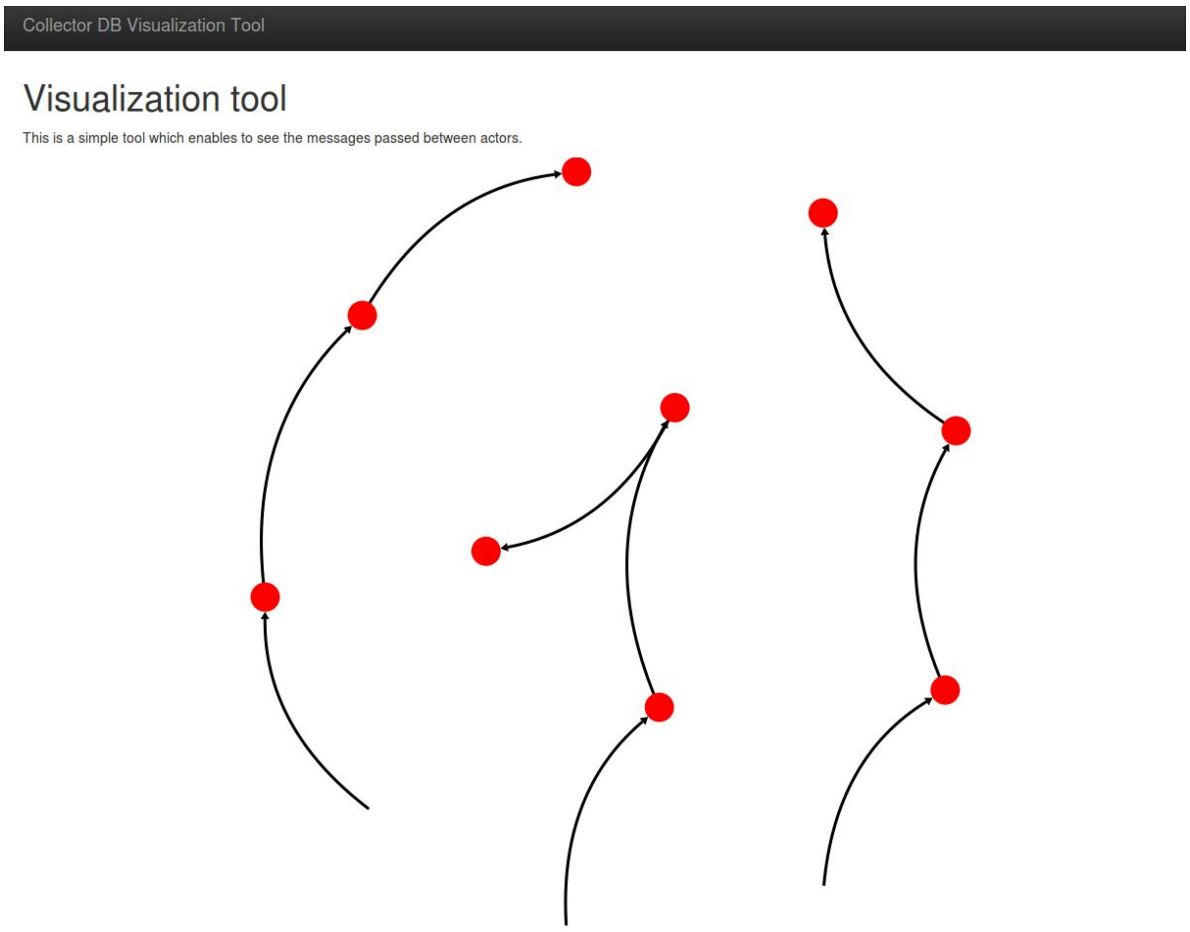

Collector DB Visualization Tool
Visualization tool
This is a simple tool which enables to see the messages passed between actors.

Figure 6. Screenshot from the visualization tool

As we can see in Figure 6, the tool only enables us to see the traces - there are no interactions with user. But it fulfilled its purpose - we could visually see if the collected traces were correct. Therefore we could verify if the library was working well.

Milestone summary

This milestone resulted in a simple tool which allows users to see collected traces as a directed graph.

5.1.5 SBT plugin

This milestone was solely focused in reducing the amount of code that was needed to be inserted to the build definition file. We needed some configuration format that would allow user to specify which actors should be traced. We also wanted to do it in a recognized file format in Scala community. Therefore, we decided to use Typesafe’s Config library and HOCON file format - Human-Optimized Config Object Notation which is based on the JSON format. JSON is de facto now standard in storing data - also for configuration purposes. That’s why we decided to use this format.

We developed a SBT plugin which loads and parses configuration file, generates proper aspect, the aspect’s configuration file and integrates generated sources into the user’s application.

For increasing the usability, we allowed user to just specify packages which contains the actors that should be traced (but explicit configuration - specifying only some classes from the packages is also possible). This was possible due to the AspectJ pointcut definition that can take * as the class name which translates to “any class”. Therefore, we crossed a big bridge in bringing the usability to our library.

Another part of this milestone was to tweak and optimize aspect’s template so that the message’s contents were also persisted. Another thing that our client wanted was to change the default database engine to SQLite as it does not require any database engine installed on user’s system - SQLite itself can be run directly on the JVM as part of our library.

Milestone summary

This milestone resulted in delivering a SBT plugin which automatizes the aspect generation process enabling much greater usability and making our library much easier to use for the user.

There were some tweaks and optimization in the aspect’s template that allowed us to have the message’s contents persisted in our trace.

We also changed the database engine to SQLite based on the client’s suggestion and requirements.

5.1.6 Documentation and tutorial with working library and example

At the beginning of the developing process to deliver this milestone it occurred that there are some small bugs in the code. So the first step was to resolve them. We fixed them and presented the library to our client.

At this point the library was fully functional. Our client confirmed that it is doing what it should be doing. What he wanted was an example that describes the integration of our Project into some existing actor system.

Although he knew how the library worked because of the regular meetings, our client wanted to have this tutorial written to a user which does not understand how the library worked.

Therefore, we created a tutorial showing a potential new user how to integrate our library into an existing actor system. We adopted one of our examples that we created before as the example actor system used in this short instruction.

We showed user how to add our library step by step to the project. We also informed him more or less what each step does, so he could understand (at least at some general, abstract way) how the library work.

Our client suggested also creating a repository or branch that would allow client to see the fully integrated library. We decided to create a branch on the example's repository that showed exactly that - what user can expect as a result after working his way through the tutorial.

The last step of this milestone was to write most of the missing documentation of our library. We created Development Process Documentation (this document), Technical Documentation and User Documentation. These documents were formally required as a part of our Project. But, regardless of formal requirements, it was necessary to document our library so potential users could use it in their projects and, if they desire, change its code to adapt to their needs which weren't included or thought of in our Project's vision.

Milestone summary

A fully functional example of the library's usage was created during this iteration. A tutorial has been written that shows how to integrate the library into an existing actor system. What is more, some bugs were corrected. This was also the iteration that important documentation was written so the users can learn about the library itself.

6 End product evaluation

The development resulted in a fully functional Scala library that allows users to trace their Akka applications. The product allows to collect and visualize information about the messages passed between actors with full contents of the messages, its sender and its receiver.

The following requirements were made about the library:

- The library should be user-friendly - the amount of required actions that user needs to take to include the library to his project has to be small.
- The library should collect information about messages passed between actors.
- The library should be easily extendable.
  The above requirements have been fully met by our product.

The following requirements were not met:

- The library should allow users to see the full stack trace if an exception is thrown in actor.
- The library's configuration can be changed when the actor system is running and the new contents will be taken into account in the tracing.
- The tool should provide an integration feature with existing visualization tools.

The requirements above were not met due to the following reasons:

- Not enough time to make them work properly in any actor system.
- Not enough understanding of the inner structure and the data APIs of the external visualization tools.
- Complexity of the requirements.

During the development process several proofs of concepts were created as well as working prototype of the product. The prototypes and proofs of concepts were then evaluated. This allowed us to control the product's development process and the next steps that were needed to made.

7 Future works

The library provides basic information about messages passed by the actors in an actor system. However, tracing is a more general concept than collecting information about how the system is working. Below are some of the ideas about the future works that can be done to produce an even better library:

- **adding additional information in the message's wrapper** - right now there is only message's id which is needed to persist the message; you can insert some other information - e.g. execution time or some statistics data,
- **sampling of the messages being stored in database** - usually there are lots of messages passed between actors - there should be some ways to reduce saved information because there are a lot of traces that are practically identical and do not provide any new information to the user,
- **better visualization** - there should be some interactive features that allow user to filter the trace to show the information that is interesting to him,
- **database queries enabling to find specific traces or information about them** - the library allows only the persistence of the traces. There should be some tools to analyse the traces to extract some information or to search by some criteria to find the specified traces.

References

[1] Richard Doyle, "Using Akka and Scala to Render a Mandelbrot Set" blog post. Available at: http://blog.scottlogic.com/2014/08/15/using-akka-and-scala-to-render-a-mandelbrot-set.html [Online; accessed 02.01.2016]

[2] Carl Hewitt, Peter Bishop, Richard Steiger, "A Universal Modular Actor Formalism for Artificial Intelligence", 1973. Available at: http://worrydream.com/refs/Hewitt-ActorModel.pdf [Online; accessed 03.01.2016]

[3] S.S. Kadam, Presentation about "Message-Passing Programming Paradigm". Available at: http://www.iacs.res.in/MPI_Collective_Communications.pdf [Online; accessed 02.01.2016]

[4] Florian Hopf, "Getting rid of synchronized: Using Akka from Java" blog post. Available at: http://blog.florian-hopf.de/2012/08/getting-rid-of-synchronized-using-akka.html [Online; accessed 02.01.2016]

[5] Munish K. Gupta, "Dispatcher and Routers", Packt Publishing online article. Available at: https://www.packtpub.com/books/content/dispatchers-and-routers [Online; accessed 02.01.2016]

[6] Iulian Dragos, "Rethinking the debugger" presentation about asynchronous debuggers from ScalaCamp 2014. Available at: http://scalacamp.pl/data/async-debugger-slides [Online; accessed 31.12.2015]

[7] Scala IDE documentation, "Asynchronous Debugger" section. Available at: http://scala-ide.org/docs/current-user-doc/features/async-debugger/ [Online; accessed 02.01.2016]

[8] Zipkin project website. Available at: http://zipkin.io [Online; accessed 31.12.2015]