A Case Study on Using Functional Programming for Internet of Things Applications

By Till Haenisch[*]

In this paper a case study, which shows that the code size and complexity of a system which collects and interprets sensor data in an Internet of Things scenario can be reduced using functional programming techniques, is presented. On one hand this is especially important for security reasons: Such a system must run for a long time without an effective way to distribute software patches. On the other hand in this kind of system the consequences of a malfunction (intended or not) are much more critical than in standard computing situations, because real world buildings or industrial sites are affected. From a high level perspective the data processing at the base station of such a sensor network can be considered as a set of mathematical functions operating on a stream of values. Each function creates a new stream of values, which might be processed by another function. This means that the complete functionality can easily be described and programmed in a functional language, such as elixir, Erlang or Scala.

Keywords: *Functional programming, Internet of Things, IT-Security, Software Architecture*

Introduction

Internet of Things appliances, such like light switches, thermostats or other kinds of sensors or actors, are especially sensitive to software errors. While minor malfunctions may be acceptable, software bugs might lead to security problems, which are not acceptable, since they will have consequences in the real world.

Today's method of keeping systems, e.g. operating systems, secure is to patch them permanently to solve security problems. That is not practical for the Internet of Things (IoT). The neccessity to patch on a regular base combined with the long lifespan of components like building automation systems would result in a severe configuration management problem: It is almost impossible to properly test systems composed of that many components, with different hardware and software versions. Constant updates will sooner or later result in interoperability problems. Even automatic patching will not solve this issue.

Since regular updates are not feasible, a different way of keeping the system secure is required. There are basically two ways to achieve this. One possible solution is a self healing system. While there are different research

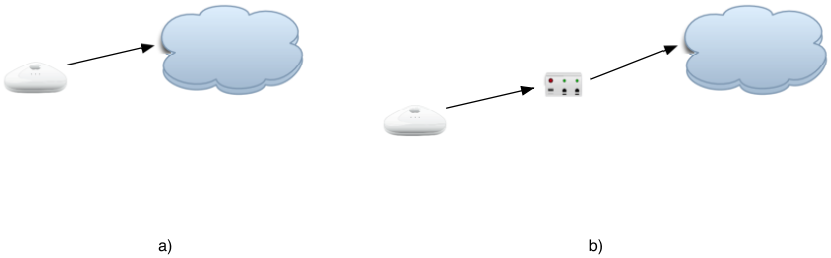
∗ Professor, Baden Württemberg State University Heidenheim, Germany.

efforts to develop methods for creating such systems, there are no practical solutions yet. The Defense Advanced Research Projects Agency (DARPA) has identified this option of addressing security problems and started the Cyber Grand Challenge in 2013 to stimulate research about self healing networks (DARPA, 2013). In 2015 DARPA launched an initiative called Building Resource Adaptive Software Systems (BRASS) to build "software systems and data to remain robust and functional in excess of 100 years" (DARPA, 2015). The only way to achieve this is enabling self adaptive systems which adapt themselves to changing environments. While this might lead to interesting results in the future, this solution is not available for current systems.

Without usable techniques to automatically solve security problems, it is desirable to keep the number of bugs close to zero. One way to lower the number of bugs is small code size and low complexity. Fewer lines of code and lower coupling, especially as few side effects as possible, means fewer bugs. The question is, how to achieve that.

In the simplest case, and only this case will be considered here, IoT means, that things talk to the internet. There are two common architectures for this kind of system: The first and simplest is a sensor node that is directly connected to the internet, typically by WLAN (Figure 1a). This requires a WLAN interface and, in most cases, an operating system that provides the necessary functionality. Typical hardware platforms for these kinds of applications are Raspberry Pi, Intel Galileo or Carambola, running some kind of Unix or Windows OS. These systems are flexible and powerful, however they require a continuous power supply since their energy consumption of up to 15 Watt (Reese, 2015) cannot be delivered by batteries.

Figure 1. *a) Sensor Node Transmitting Directly to the Internet, b) Sensor Node Transmitting to a Base, which Transmits Data to the Internet*



a) b)

In the second kind of applications, small battery powered sensors like fitness trackers or sensor nodes send their data to a base station (Figure 1b).

The base station can either be a smartphone, a PC or a special appliance depending on the technical requirements of the system. Data is collected here and can be made accessible via the Internet or at least locally via standard internet protocols.

The disadvantage of the second architecture is the base station required in addition to the sensor nodes. The advantage is, that the sensor nodes can be very simple, and might not even require an operating system. This means, they can be cheap and battery powered. This is the scenario discussed in this paper.

The paper is made up of two parts. The first part in "Functional Programming for the IoT" section discusses, why functional programming techniques are worth considering for the Internet of Things applications. The second part in the "Case Study" section presents a case study using functional programming in such a scenario.

Contributions of this Paper

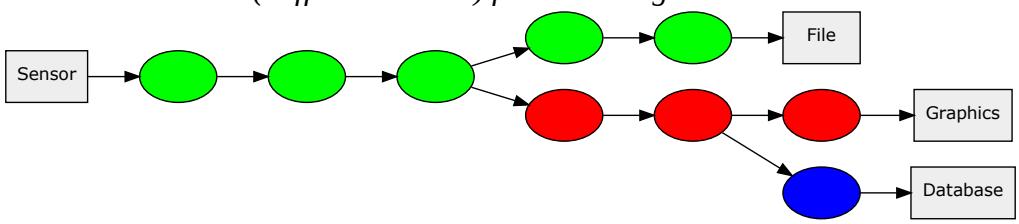
- Based on the architecture described above, we give a motivation, why functional programming languages might solve the described problems.
- In a case study it is shown, that functional programming can reduce code size and complexity in a real world Internet of Things application.
- It is also shown, that elixir, a relatively new functional programming language, is ready for use in real applications at least in this application domain.

Functional Programming for the IoT

In most cases the structure of sensor data is rather simple: Typically a measurement consists of a value, a timestamp, an identification of a location and a sensor id (what, when, where, who). Data is usually written only once and only sequentially. This gives way to using a message oriented architecture for the Internet of Things applications, for an overview see example (Zhou, and Zhang, 2014). MQTT (www.mqtt.org) is seen as the upcoming platform at least for cloud based IoT applications, all big names like IBM (Bin Tang, 2015), amazon ("AWS IoT", 2015), Eclipse ("IoT Standards", 2015) are going this way. But what happens with the messages when they arrive at their destination?

Data is read by different applications at different times for different purposes. However, most data is processed only once, at the time of its creation. Data is usually analyzed in some way at this time, for example to detect anomalies, to generate statistical distribution parameters, to aggregate the data for later time series analysis and to display a graphical representation of the current data with a surrounding context.

Figure 2. *Sensor Values Considered as Streams. Functions Generate Additional Streams (Different Colors) from the Original Sensor Stream*



All of these processing activities can be seen as the application of a mathematical function on a set of sensor values, with the special case that the

set contains only one value. Thus, the functionality of a processing node for an IoT application can be considered as a set of mathematical functions operating on a stream of values (Newton and Welsh, 2004). Each function creates a new stream of values, which might be processed by another function (see Figure 2). This means, that the complete functionality can easily be described and programmed in a functional language like elixir, Erlang, Scala or Haskell.

There is a considerable debate about the advantages of using functional programming languages or at least functional programming techniques. Many languages adopt functional features to allow using functional techniques in the preferred environment, for example (Subramaniam, 2014). This debate is not new (Gat, 2000). In Gat's classic experiment it was shown, that many properties of programs like programmer productivity, performance etc. were better when the programs were written in Lisp, a very old functional language, compared to Java, a then modern imperative language.

According to (Wortmann and Flüchter, 2015) Internet of Things platforms have to be open, simple and prospective, functional programming beeing one of the key features. Platforms like Erlang/OTP have these properties and should therefore be considered candidates for these applications.

Functional programming languages (and their environments like Erlang/OTP) are very good for writing reliable, highly concurrent applications with many concurrent processes and especially process failures (Armstrong, 2010). Writing applications like that was the reason for the development of the Erlang ecosystem in telecommunication systems like phone exchanges.

The same reasons for using functional languages in these environments are given in IoT scenarios. Concurrent event sources, e.g. sensor modules, unreliable communication with spurious errors because of wireless data transmission and a system that has to work highly reliable under any of these problems, for a discussion of the relevance of these properties in IoT applications see (Sivieri et al., 2012). Even if some sensors in a building or a factory setting are not working correctly, the data and data transformation must continue at least with the undisturbed data, the main control flow must not be affected by errors in other parts of the system. Nobody would tolerate a building where you can not turn on the lights, because a thermostat node crashes.

But this is not the most important point for choosing functional languages. An even stronger advantage of functional languages, is, that the code for transformations like the ones described above, is much more concise than with traditional imperative languages. Although there is no formal proof for this assumption, there is a large number of anecdotal cases, for example from (Ford, 2013) or the case study described in a later section of this paper. An impressive case is John Carmack from ID software, who reimplemented Wolfenstein 3D in Haskell and found, besides other promising benefits that the code size was reduced significantly (Carmack, 2013).

Short code without side effects (pure functional languages do not have side effects) is easier to verify for correctness than the imperative code. That means, it contains fewer errors. While there is a significant, but only small correlation

between the programming language and the error rate, there is a clear dependency between code size and error rate (Ray et al., 2014). Since programs written in functional languages tend to be shorter than programs written in imperative languages, they should contain fewer errors.

Fewer errors means less security problems, which is the main point. Internet of Things applications have a direct relation to the real world. Security problems in this context mean not only damaged files on a disk, which might be restored from a backup, but cause damage and or monetary loss in the real world.

Another advantage of functional programming techniques is that they reduce side effects: This is the main idea of functional programming, composing a program from "pure" functions. Security wise this is a good idea, especially in embedded scenarios. The Industrial Internet Reference Architecture (IIC, 2015) recommends to "avoid introducing unknown or undesired side effects"

To limit the possible damage by security problems in the IoT applications, it is either neccessary to develop and deploy a widely accepted platform, that has few bugs and is constantly updated throughout the world like for example Apples iOS or we need as much diversity in these systems as we can get to reduce the risk of a complete failure (Schneier, 2010). Software diversity is a promising way to achieve anti-fragile systems (Hole, 2015). Lacking other accepted technical solutions that means individually developed software with as few bugs as possible. And that means short, simple programs, which are easy to test and verify.

In the following chapters a case study is presented which shows that the code size and complexity for systems which collects and interprets sensor data in an IoT scenario can be reduced using functional programming techniques.

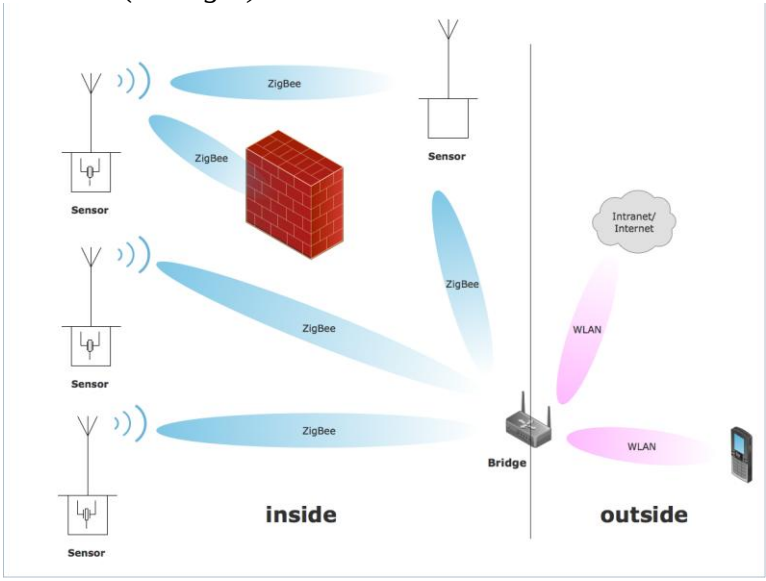
Case Study

The case discussed in this paper is a low cost low power sensor network to save energy in paper machines (Hänisch, 2014). By using wireless sensors for measuring temperature and humidity in the dryer section of a paper machine it is possible to optimize energy consumption by adjusting heating and air flow. Because the sensors need to be battery powered and send the data almost in real time for monitoring purposes, a low power network technology is needed, in this case ZigBee. The data is sent to a base station in packets with no guaranteed delivery, resulting in an at most once semantic. This results in some complexity of the base station code, which consists mainly of error handling and monitoring or logging functions.

In this article, only the code running on the base station (see Figure 3) will be considered, the code on the sensor nodes mainly handles communication with the sensor hardware and has a very simple structure since no data is stored locally. In more complex cases this part could also be implemented using

functional programming techniques like functional reactive programming (Khare et al., 2015).

Figure 3. *Architecture of the Sensor Network Consisting of the Sensor Nodes and a Base Station ("Bridge")*

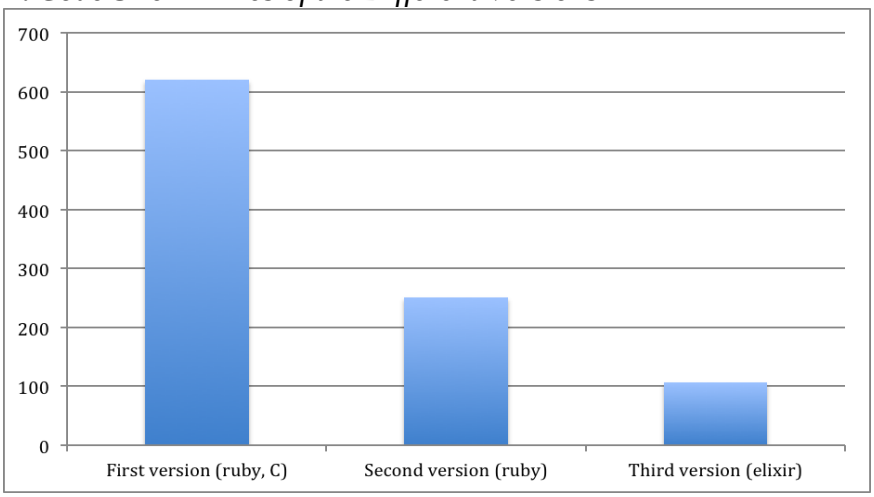


As Figure 4 shows, the original version consists of 620 lines of source code (262 lines of C, 355 lines of ruby, 3 lines of python) plus a few shell scripts for startup tasks etc.

By reevaluating the user requirements the code size could be reduced to 251 lines of ruby (including comments and empty lines, that is 225 nonempty lines resp. 194 nonblank lines without comments). That is some 40% of the original size. The reimplementation in elixir resulted in 106 lines of code (including comments and empty lines, that is 86 non-blank lines resp. 68 lines of code without comments). That is some 42% of the second version, 17% of the original version.

Remark: The Erlang version was about the same size as the elixir version (which is no surprise, since it has the same structure, the same functions etc.) but felt somewhat alien at least to the author and was dropped in favour of the elixir version.

Figure 4. *Code Size in Lines of the Different Versions*



That means, that the code size was reduced by a factor of 5.

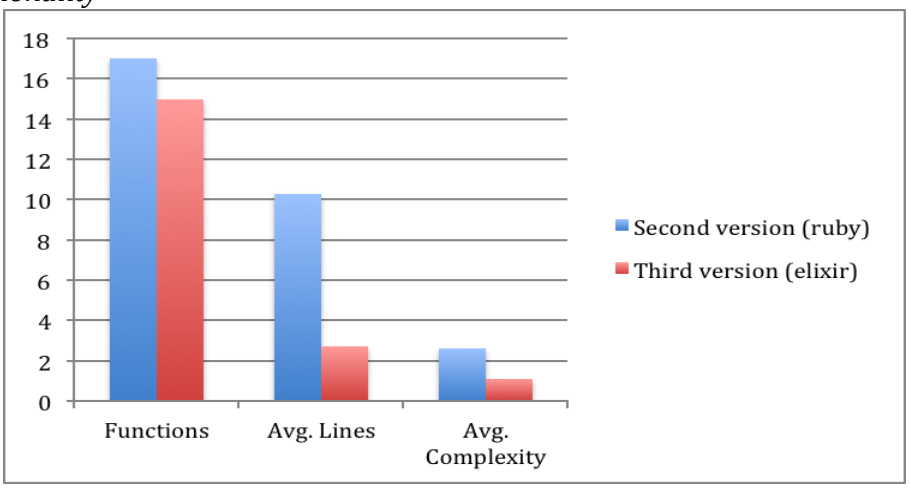
Figure 5. *Listing of the Most Complex Function of the Elixir Version*

```
def process_file(input_file,current_values) do
  task = Task.async(fn -> IO.read(input_file, :line) end)
  try do
    row = Task.await(task,5000)
    if (row != :eof) do
      new_values = process_line(
        String.split(String.rstrip(row), " "),current_values)
      process_file(input_file,new_values)
    end
  catch
    :exit, _ -> IO.puts "timeout"
      process_file(input_file,current_values)
  end
end
```

The final elixir version consists of 15 functions with an average length of 2.7 lines (only 6 functions have a length of more than 1 line). Only one function has a cyclomatic complexity greater than 1, this is the function shown in Figure 5. This function handles timeouts when receiving data, so a complexity greater than one is mandatory.

The ruby versions consist of 3 classes with 17 methods having an average length of 10.3 lines. The average cyclomatic complexity is 2.6, the maximum cyclomatic complexity is 10. These numbers (and Figure 6) show that the elixir version is not only much smaller (42 % of the size of the ruby version) but also the complexity is lower, by a similar factor. According to (Watson and McCabe, 1996) cyclomatic complexity correlates with the number of errors in software modules. So the elixir version should have fewer errors than the much longer and more complex ruby version.

Figure 6. *Complexity Measures of the Ruby and Elixir Versions with the Same Functionality*



The original version (ruby and C) was replaced by the second, simplified ruby version because it showed a number of critical errors which were hard to find because they occured only rarely. This lead to a simplified ruby version which worked over a period of nearly two years with only two non critical bugs. The elixir version is running for only a few months now and showed no bug til today.

Discussion

The problems with a study with n=1 are well known, see for example (Harrison, 2000). But experiments in software engineering are hard to do: Controlled experiments with n>1 give better results, if and only if both samples are from the same basic population. This basic population must be representative for the real word. This is the problem with the controlled experiment approach. Usually experiments are done with voluntary students, but it would be difficult to find students which have the same amount of experience level, in ruby and elixir in this case. Typically someone knowing those two languages has way more experience with ruby than with elixir, since elixir is newer. Programmers knowing elixir or Lisp, Scala, Erlang will tend to have a more theoretical background than the typical developer of embedded systems, but much less practical experience. So even if an experiment with a large number of participants would be of limited use for real projects.

The size of the example described above is much smaller than typical industrial projects. So the only firm conclusion that may be drawn from this case is that further, larger experiments are needed. On the other hand the processing of data in Internet of Things scenarios might (and probably should) (Namiot and Sneps-Sneppe, 2014) be implemented as microservices, with a size comparable to the case described here.

Both of these points are valid, but controlled experiments with realistic project sizes are very hard to do: The group of people who would volunteer to work for a few years on a software project that is developed by a large number of other teams concurrently just to get some statistically valid data about program complexity is limited and certainly not representative for real world software engineers. So this problem is unsolvable and we will have to stay with small n=1 case studies.

Using the cyclomatic complexity as a measure for the expected number of errors in code is debatable, see for example (Abran et al., 2004). On the other hand it is widely accepted and used in tools to measure complexity for exactly this purpose. In conclusion the correlation might not be absolutely proven, but in real world experience it works and it is plausible: The more paths in the code, the harder to understand and test, the harder to understand and test, the more errors.

Conclusions

Using functional programming techniques and/or languages can reduce the code size and the complexity of the Internet of Things applications. Reduced code size and complexity means less bugs, that means less security problems.

Functional programming techniques fit well to the architecture of the Internet of Things applications. It is therefore plausible that the described reduction in code size and complexity could be realized in other projects as well. Elixir seems to be a good choice as an implementation language for Internet of Things applications.

References

Abran, A., Lopez, M., and Habra, N. 2004. An Analysis of the McCabe Cyclomatic Complexit Number, Proceedings of the 14th International Workshop on Software Measurement (IWSM) IWSM-Metrikon, 2004, Magdeburg, Germany: Springer-Verlag, 391-405.

AWS IoT 2015. aws.amazon.com/iot.

Armstrong, J. 2010. Erlang, *Communications of the ACM* 53(9).

Bin Tang, C., 2015. Explore MQTT and the Internet of Things service on IBM Bluemix. http://ibm.co/1LDiJFD.

Carmack, J. 2013. *Keynote at QUAKECON 2013*, http://bit.ly/1Ij5u2a.

DARPA 2013. DARPA Cyber Grand Challenge Competitor Portal. https://cgc.darpa.mil.

DARPA 2015. DARPA Seeks to Create Software Systems That Could Last 100 Years. http://bit.ly/1aLNazw.

Ford, N. 2013. Functional thinking: Why functional programming is on the rise. http://ibm.co/1jsUymLGat 2000.

Gat, E. 2000. Point of view: Lisp as an alternative to Java, *Intelligence* 11(4): 21-24.

Hänisch, T. 2014. Using a Sensor Network for Energy Optimization of Paper Machine Dryer Sections, *Athens Journal of Technology Engineering* 1(3).

Harrison, W. 2000. N=1, an Alternative for Software Engineering Research? Proc. Workshop Beg, Borrow, or Steal: Using Multidisciplinary Approaches in Empirical Software Eng. Research, Int'l Conf. Software Eng., Aug. 2000.

Hole, K. J. 2015. Toward Anti-fragility: A Malware-Halting Technique, IEEE Security/Privacy, July/August 2015.

IIC 2015, Industrial Internet Reference Architecture. http://bit.ly/1MnpgG0.

IoT Standards 2015. http://iot.eclipse.org/standards.

Khare, S., Tambe, S., An, K., Gokhale, A. and Pazandak, P. 2015. Functional Reactive Stream Processing for Data-centric Publish/Subscribe Systems. http://bit.ly/1Y CDz15.

Namiot, D. and Sneps-Sneppe, M. 2014. On IoT Programming, *International Journal of Open Information Technologies* 2(10).

Newton, R. and Welsh, M. 2004. Region streams: functional macroprogramming for sensor networks, Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB, 78-87.

Ray, B., Posnett, D., Filkov, V. and Devanbu, P. T. 2014. "A Large Scale Study of Programming Languages and Code Quality in Github" Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.

Reese, L. 2015. A Comparison of Open Source Hardware: Intel Galileo vs. Raspberry Pi. *Technical Report. Mouser Electronics*. http://bit.ly/1Opw3np.

Schneier, B. 2010. The Dangers of a Software Monoculture. *Information Security Magazine*, November 2010.

Sivieri, A., Mottola, L. and Cugola, G. 2012. Drop the Phone and Talk to the Physical World: Programming the Internet of Things with Erlang, SESENA '12 Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications

Subramaniam, V. 2014. Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions, *O'Reilly*.

Watson, A. H. and McCabe, T. J. 1996. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publication*.

Wortmann, F. and Flüchter, K. 2015. Internet of Things - Technology and Value Added, *Business & Information Systems Engineering* 57(3): 221-224.

Zhou, C. and Zhang, X., 2014. Toward the Internet of Things Application and Management: A Practical Approach, WOWMOM, 2014, 2014 IEEE 15th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM).