Loosely Coupled Actor Systems for the Internet of Things

Master Seminar I

Raphael Hiesgen

Hamburg University of Applied Sciences

February 28, 2015

Contents

1 Introduction

The Internet of Things (IoT) describes a network of nodes connected by Internet standards and often requires minimal human interaction to work. Individual nodes often have limited hardware capabilities and are dedicated to a single, simple task. Complex services are composed of many cooperating nodes. This leads to a highly distributed work flow that relies on machine-to-machine (M2M) communication. Further, communication is built upon open network standards and commonly includes connectivity to the Internet.

Traditional application scenarios include sensor networks, which can collect data such as environmental or cyber-physical conditions. Besides sensors, IoT networks include actuators that can influence their environment, often in a very limited way. Built from these nodes are complex applications that enable home automation, tracking physical and environmental conditions. These systems enable machines to upload data to Internet servers, a task that originally required human interaction. Thus, they facilitate the availability of information everywhere and anytime.

Developers of distributed applications need synchronization primitives as well as mechanisms for error detection and propagation to ensure an appropriate service quality while working on a network of machines. When faced with these challenges, many developers fall back to low-level coding that focuses on specialized knowledge. As a result, code is barely portable, and often hand-crafted, which introduces high complexity, many sources of errors and little generality.

The actor model is designed to model and develop concurrent systems and provides a high level of abstraction for distributed software. It describes concurrent software entities known as actors that communicate via network-transparent message passing. Developers can benefit form a high abstraction level by using the actor model to develop applications for the IoT. It allows them to focus on the application logic instead of spending time on implementing low-level primitives.

We contribute the `C++ Actor Framework` (`CAF`) [1] that allows for native development in C++ at high efficiency and a very low memory footprint. Its API is designed in a style familiar to C++ developers. Furthermore, `CAF` features type-safe actor interfaces and a scalable work-stealing scheduler. We are working to provide runtime inspections tools that enhance distributed debugging. The communication in `CAF` is targets locally distributed multicore machines and is built with the strong coupling known from traditional actor systems.

Our adaptations for the IoT weaken the coupling between actors and add features to enable deployment in low-powered and lossy networks. These features include the handling of unreliable links and infrastructure failures, provide a suitable error propagation model as well as a lightweight secure and authenticated connectivity. We rely on protocols optimized for the use in IoT environments. Specifically, we provide a network stack for `CAF` based on IPv6 over Low-power Wireless Area Networks (6LoWPAN) [2], the Constrained Application Protocol (CoAP) [3] and the Datagram Transport Layer Security protocol (DTLS) [4]. Currently, `CAF` is ported to RIOT [5], the friendly operating system (OS) for the IoT.

The remaining work is structured as follows. Section 2 introduces the the actor model as a concept for distributed and concurrent applications. Further, the open source `C++ Actor Framework` is presented. The following Section 3 discusses the use of the actor model for the development of IoT applications and takes a look at the resulting challenges. In Section 4, we outline our approach to adopt `CAF` to the IoT environment. Section 5 presents the questions we want to answer in this project as well as the associated risks. In addition, the Section summarizes of our next steps. Thereafter, Section 6 stresses the importance of testing and introduces some available testbeds. Finally, a conclusion is drawn in Section 7.

2 The Actor Model

The actor model is designed for concurrent and distributed environments. It defines isolated entities called actors, that run in parallel and solely interact via network transparent message passing based on unique identifiers. As such, the model avoids race conditions and prevents actors from corrupting the state of other actors.

New actors are create using the operation `spawn`. This operation is not limited to the runtime, but can be called by actors as well. Thereby, workloads can be easily distributed, e.g., in a divide and conquer approach.

Error handling in the actor model is implemented in monitors and links, which allow error detection and propagation in local as well as distributed systems. When a monitored actor terminates, the runtime environment sends a message containing the exit reason to all monitoring actors. Links are bidirectional and express a stronger coupling. In a linked set of actors, each actor will terminate with the same error code as its links. This allows the modeling of (sub-) systems in which actors fail collective and thus avoid inconsistent or intermediate states. Moreover, failed actors can be redeployed at runtime.

Hewitt et al. [6] proposed the actor model in 1973 to address the problems of concurrency and distribution. Later, Agha focused on theoretical aspects in his dissertation [7] and introduced mailboxing for processing actor messages. Further, he created the foundation for an open, external communication [8].

A well-known implementation on the actor model is the Erlang programming language. It was designed by Armstrong [9] and targeted telephony applications. As such is was meant for distributed systems that run with minimal downtime. Although Erlang does not mention the actor model directly, it provided the first de-facto implementation. A more recent implementation is provided by the Akka framework [10] as part of the Scala standard distribution. It offers object oriented as well as functional programming and runs in Java virtual machines (JVM).

2.1 `CAF`—A Scalable Actor System

The open source `C++ Actor Framework` (`CAF`) [11] aims to improve the development of concurrent and distributed software with a focus on scalability. It features an exchangeable runtime environment to enable scalability up to many cores and machines [12] as well as down to embedded hardware. This versatility gives developers the opportunity to test and verify their code on desktop machines before re-compiling and deploying the software on low-end devices. Hence, `CAF` provides a seamless development cycle, enabling developers to ship only well-tested components to the IoT. To ease testing and debugging, `CAF` offers type-safe messaging interfaces that eradicate a whole category of runtime errors. Furthermore, we are working on tools to monitor distributed actor systems. Under the title "Runtime Inspection and Configuration" (riac) these tools allow developers to track messages between actors and inject new ones. Thereby, specific scenarios can be recreated and tested as part of a distributed deployment.

In `CAF`, actors are created using the function `spawn`. This function takes a `C++` functor or class and returns a handle to the created actor. Hence, functions are first-class citizens in our design and developers can choose whether they prefer an object-oriented or a functional software design. Per default, actors are scheduled cooperatively using a work-stealing algorithm [13]. This results in a lightweight and scalable actor implementation that does not rely on system-level calls, e.g., required when mapping actors to threads.

Each actor has a behavior that consists of a set of message handlers which specify how it processes incoming messages. Messages are buffered in the mailbox of the receiver in order of arrival before they are processed. Behaviors may include a message handler that is triggered

if no other message arrives within a declared time frame. Actors are allowed to dynamically change their behavior at runtime using `become`.

`C++` is a strongly typed language that performs static type checking. Building upon this, it is only natural to provide similar characteristics for actors. With typed actors, `CAF` provides a convenient way to specify the messaging interface in the type system itself. This enables the compiler to detect type violations at compile time and to reject invalid programs. In contrast, untyped actors allow for a rapid prototyping and extended flexibility. Since `CAF` supports both kinds of actors, developers can choose which to use for which occasion.

A major concern when using high-level abstractions in the context of embedded devices is memory consumption. Facing hardware that is constrained to a few kilobytes of RAM, virtualized runtime environments and memory inefficient garbage collectors are too costly. Careful resource management as well as a small memory footprint are needed. To demonstrate the applicability of our implementation to the IoT, we compare our system to the actor implementations in Erlang and Scala (using the Akka library). Both systems are widely deployed and often referred to as the most mature actor implementations available.

Figure 1 shows a box plot depicting the memory consumption for a simple benchmark program creating 2^{20}, i.e., more than a million, actors. It compares the resident set size in MB for `CAF`, Scala, and Erlang. `CAF` shows a mean memory usage around 300 MB, while Scala consumes a mean of 500 MB and Erlang uses around 1300 MB of memory. In all cases, the median memory consumption is a bit lower than the mean. Creating such numbers of actors on an embedded device is not feasible, but the benchmark illustrates that each actor in `CAF` requires only a few hundred bytes. In addition, the graph shows how far the measured values stray. For `CAF`, 1% of the measurements consumed up to 700 MB of memory. Scala consumes up to the double amount, while Erlang peaks at more than 3 GB. Although this margin is placed around the twice the mean in all cases, the absolute difference is far greater in the cases of Scala and Erlang. Since embedded device are often constrained in memory, having lower peaks in memory consumption allows for more precise runtime predictions and is desirable.

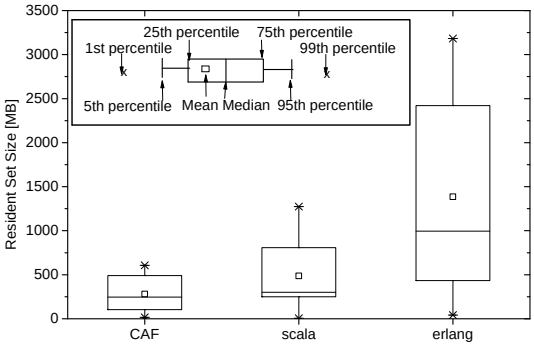

Figure 1: Memory consumption for the creation of 2^{20} actors

When comparing our system to the virtualized approaches, `CAF` reveals an extraordinary small memory footprint in realistic application scenarios, while outperforming existing mature actor implementations on commodity hardware [1].

3 A Programming Model for the IoT

The Internet of Things (IoT) is a major trend in the recent years. A lot of companies release new devices in the hope to be first ones with the new idea that catches on. New smartwatches, fitness trackers and home automation device are released frequently. Google already released their operating system Android for watches and bough Nest for more than a Billion Dollars, a company that develops “smart” thermostats and smoke detectors. Apple offers a hub for home automation application and fitness tracking in their operating systems. Besides these big players, a lot of smaller companies emerge that offer new products or hardware development kits for hobbyists. Along with proprietary developments, new standards and open development tools arise that help programmers to implement applications for these devices. In any case, the

number of connected devices is rising fast. As a result, more and more developers will create applications for this environment.

Software for the IoT addresses a highly distributed environment. Nodes may be part of larger networks of dynamic size and are possibly mobile. Hardware is often constrained and limited in processing power, link performance and battery life. In addition, small packet sizes, packet loss due to interference and temporary connection failures need to be considered.

Development for embedded systems is often done in low-level languages such as `C`. `C` does not only give developers a lot of freedom and possibilities, but introduces many opportunities to make mistakes. Moreover, message exchange and synchronization as well as error propagation and mitigation have to be implemented. Communication is not only required to work properly and with little overhead, but has to scale up to many devices, maybe more than expected during the development process. For example, synchronization errors in concurrent software may depend on the number of cores or nodes. When done incorrectly, communication overhead my amplify and race conditions or life-/deadlocks occur. In addition, programs that perform well on a few cores or nodes need not scale with increasing the available hardware resources, but may even show worse performance. Implementing these primitives requires specialized knowledge from multiple domains to develop and is usable by many different applications.

Debugging applications is an essential part of software development. A global view on a distributed system is hard to achieve. Nodes do not share the same time and cannot applications can not executed step by step as common on local machines. Further, messages are not always received in a deterministic order over multiple tests. In addition, a network of constrained devices is easily influenced by interference from the environment and own transmissions. This leads to hardly reproducible test cases, which are, nevertheless, required to ensure scalability and validate the application. Lastly, creating portable software is not an easy task in the presence of a heterogeneous hardware environment. New hardware requires new drivers and possibly adjustments to a different architecture.

A suitable runtime environment deployed on operating system (OS) can ease the development process. The OS wraps hardware specific functionality and provides an standardized API that improves software portability. Deployed on top, the middleware abstracts over communication and synchronization primitives and addresses scalability. There are several embedded operating systems available, namely ARM mbed OS [14], Contiki OS [15], TinyOS [16] and RIOT [5]. We target RIOT with our project as it is open source and provides `C++` support as well as multi-threading. It should be noted, that support for `C++` is not a given as it introduces additional overhead. For the middleware, we suggest the actor model, which is designed for concurrent and distributed applications.

The actor model is characterized by a message-driven work flow for distributed systems. Implemented as an efficient middleware layer it provides a scalable development platform. However, it can not simply be deployed, as this new domain introduces new challenges. Traditional actor environments do not take lossy networks and low-power nodes into account. Furthermore, the strong coupling that was originally part of the actor model is not present here. In addition, the distributed error handling capabilities were not designed with these constraints in mind and require adjustment.

Security considerations are not included in the actor model and left to the runtime environment. IoT devices such as fitness trackers or home automation systems have access to a range of private data. In particular, security systems should work reliably and remain resilient against tampering. Most IoT devices depend on communication, e.g., for joint operations or data collection. Wireless networks are widely deployed as they ease setup and support mobility of participants. As a result, the network should be secured as it is easily accessible in the vicinity Security considerations should include encryption, authentication and authorization.

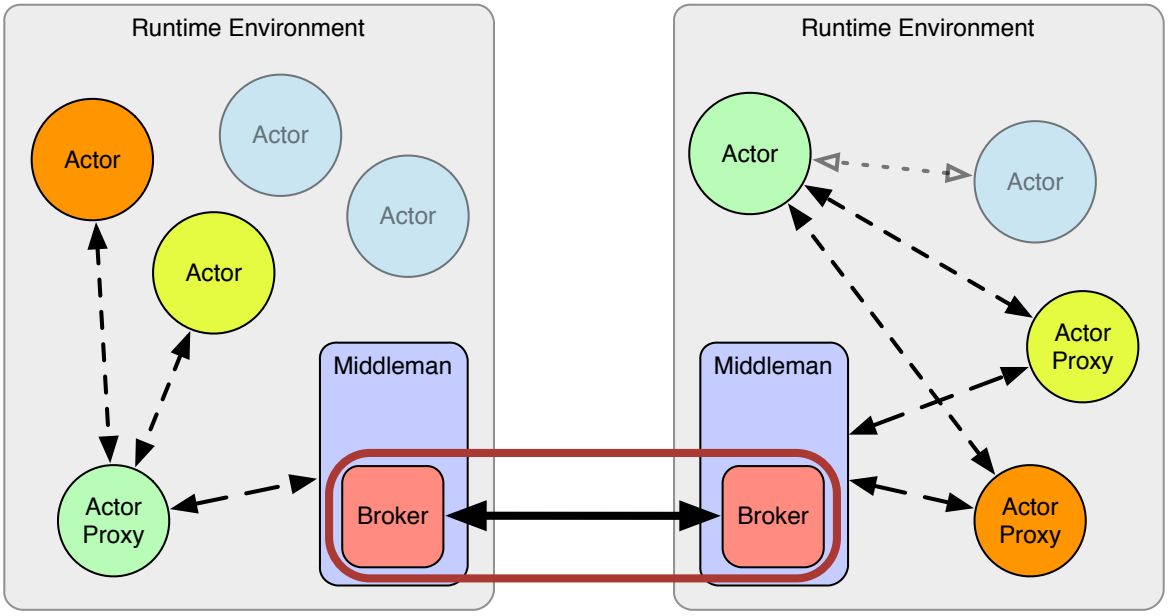

Figure 2: A simplified presentation of the communication between actors. The red-circled exchanged has to be secured.

It is not necessary to secure the connection between all actors. Figure 2 depicts two simplified runtime environments of `CAF`, running four and two actors respectively. Actors within the same runtime environment run within the same process and can exchange messages directly. Actors that communicate with remote actors are represented by an *Actor Proxy* in the remote runtime environment, marked with the color in the Figure. When addressed, an Actor Proxy communicates with the remote actor via the middleman, which handles the message exchange. The middleman uses brokers to abstract over low-level IO. Instead of implementing encryption and authentication in each actor, it is sufficient have a broker handle these tasks. The message exchange that has to be secured and authenticated is circled in red.

4 Embedded Actors with `CAF`

Our overall goal is to ease the development of applications for the IoT. We provide a middleware that raises the abstraction level when developing distributed applications. This reduces the development overhead as the middleware handles often used functionality, such as network communication, synchronization and provides error handling capabilities. Moreover, we aim to improve the debugging process for distributed applications by providing tools that help monitor the system and allow developers to step in and recreate scenarios. Using a maintained and well tested middleware supports the development of reusable and robust code. Combined with an embedded operating system portability is eased immensely. All in all, our efforts contribute to a professionalization of embedded and distributed applications.

Enabling our framework to meet the challenges of IoT environments requires us to adjust the messaging layer to handle link and infrastructure and implement a valid security scheme that offers encryption and provides authentication and authorization for nodes.

Our adjusted message-passing layer models each message exchange as a transaction. It is based on the request-response model specified by CoAP. The Confirmable (CON) message type offers reliable message exchange as well as duplicate message detection. As a result, each message exchange is independent and less vulnerable to connection failure than a data stream. To handle cases where messages cannot be delivered after multiple retries, our runtime environment requires

error propagation and mitigation capabilities.

The overhead of reliable message transfer is not always desirable in IoT applications. For example, regular updates from sensors may track a slow change over time, where a single message may be lost without impact on the application. As a result we want to offer this choice to the user. Instead of introducing new functionality, we map the semantics of synchronous and asynchronous messages to the corresponding message types offered by CoAP. The unreliable message type of CoAP, Non-confirmable (NON) message, is used for asynchronous communication whereas the reliable message type, Confirmable (CON) message, is used for synchronous communication. Hence, CoAP can be seen as a natural container for carrying actor messages over the network layer.

Our transactional network stack for the IoT differs from the default TCP-based implementation in `CAF`. The stream-based stack uses TCP and IP over LAN/WAN type networks. In contrast, we target at IEEE 802.15.4 or Bluetooth on the network access layer. The IP layer deploys 6LoWPAN to keep IPv6 compatibility while the transport layer uses UDP. The message exchange will be based on the request-response model from CoAP. This allows us to design the message exchanges as transactions, which increases the robustness of our network stack.

Concerning security, we want to rely on DTLS for encryption. Naturally, the encryption deployed on constrained hardware must be strong enough to resist crypto attacks on desktop grade systems or clusters.

Wireless communication cannot only be easily captured, but messages can also be injected into the network. Furthermore, it is important to prevent malicious nodes from joining our network. Hence, we are working on an authentication and authorization scheme for IoT environments [17].

Deciding which node to trust is a nontrivial challenge. While distributing pre-shared keys allows signing messages appropriately, the keys can be obtained if with the hardware is compromised. Provided the incident is detected, trust of the compromised key can be revoked. Public-key infrastructure (PKI) commonly deploys a "central" instance where trust is tracked. In a loosely couple IoT environment this can not be guaranteed. The IETF working group on Authentication and Authorization for Constrained Environments (ACE) is working on a draft that specifies a straight forward approach to two-way authentication for IoT scenarios [18]. They suggest the deployment of such a trusted entity as a resource-rich server using X.509 certificates [19] for authentication.

5 Overview—Implementing Embedded Actors

The previous chapters provided an entry point to the research topic. The actor model can help to achieve this goal by introducing it to the highly distributed IoT environment. This section will present the research questions we hope to answer and shortly discuss the risks we face. Lastly, it maps out our next steps.

5.1 Research Questions

Introducing an established concept to a new environment raises obvious questions regarding suitability. Besides, technical details are of interest, i.e., if the protocols we deploy suit our use case. If the implementation itself proves to be solid, user adaption is critical—but mostly out of our hands.

What is the right way to the map the abstract communication between actors to low-level protocols? Hiding complexity and implementation details results in a trade off between generalization vs. configurability and complexity. As a result, we need to find the

right way to map the primitives of the actor model to the lower-level protocols. In our case we abstract over a number of protocols, from IEEE 802.15.4 on the network access layer to CoAP on the application layer. The resulting API offered to the application developer should meet the expectations for the actor model, such as synchronous and asynchronous communication as well as links and monitors for error handling. A desirable mapping provides an API that suggests the associated costs.

Can we meet efficiency expectations regarding hardware resources? IoT devices are often low-powered and have limited resources, such as battery life, CPU power or memory consumption. Further, packet loss and link failures are common. Using an additional layer of abstraction may introduce an ideally negligible overhead. In our case, the use of `C++` leads to an increase in memory usage. Furthermore, we provide a middleware that manages actors and their network communication. As such, we have manage these resource efficiently to provide a feasible solution. Once the basic implementation is working with a suitable mapping between actor communication and low-level protocols, we need to optimized the implementation for this constraint environment.

Is the actor model well suited to express typical application scenarios? While it is often possible to model problems in many different ways, a good model provides abstraction, is straight forward to express and has minimal overhead. Furthermore, extensibility and maintainability are desired. How many of these criteria can be met for implementations of typical use cases? Use cases in this category include the collection of data at sink, sending updates to all node in the network and periodic monitoring tasks.

5.2 Risks

The success of this project depends on may different areas. Most obvious is the technical feasibility, which includes efficient management of resources such as processing power, battery life and memory.

If applications developed with our framework increases the energy consumption significantly compared to a hand-crafted low-level solution, the abstraction is offers may not be worth it. The same is true for memory requirements and processing overhead. Our first test show increase in the size of our executable. Although memory is constantly getting cheaper, this does not imply larger built-in memory. For IoT devices the focus is often on costs and device size. Costs are calculated for the total number of devices produced which leads to a large amount, even if the difference is only a few cents for a single node. In addition, we have to take the realization of our high level concept into account.

Even if the implementation works, the overall concept might not succeed. he goal is to provide a painless gain for the development of IoT applications. If the implementation of typical tasks is cumbersome or does not allow enough opportunity for optimization the abstraction might not be worth it.

Another one of our goals is a security scheme for IoT applications. Traditionally, security is hard to do right and often entails unforeseen holes or flaws. As such, a scheme that meets our demands may turn out to be vulnerable to new attacks or ask too much of our target environment.

Lastly, community adoption is required to improve the project on a long term. Without application to real-world problems it is hard to notice missing features, inconvenient application scenarios and optimization opportunities. It should be noted, that the projects we are building

upon are doing well and receive public interest. While this is definitely not a guarantee it provides a good foundation to reach interested people.

5.3 Next Steps

Applications in the IoT are often not deployed as a stand alone application, but compiled with a light weight operating system (OS). Commonly, these systems provide significantly less functionality than a desktop OS. Hence, libraries may need specific adjustments depending on the target platform. Ongoing efforts to professionalize OS concepts and software for the IoT—including a clear hardware abstraction—are undertaken by RIOT [5], the friendly OS for the IoT.

The actor framework is split into two important parts that we need to enable on RIOT, the `core` library that includes actors and local interactions and the `io` part implements allows communication with remote nodes. Our development progress enable the `core` on RIOT. We can execute simple programs on embedded devices using RIOT and `CAF`.

Before porting the `io` part to RIOT, we need to implement the network stack in `CAF`. It requires UDP and CoAP for the transactional message passing layer. The UDP implementation is straight forward based on the Berkeley socket API. CoAP on the other hand is standardized, but lacks a widely adopted implementation that features a `C++` API. For the prototype, we used the open source library libcoap [1]. However, it required adjustments to be usable in our context and does not feature a `C++` wrapper. Before implementing CoAP ourselves, research is necessary to see if newer libraries with a suitable interface are available by now. In addition to libcoap, RIOT has support for microcoap [2]. Once the transactional layer is implemented in `CAF`, it needs to be ported to RIOT as well. Hopefully, the experience with the core library will ease the process.

We motivated the need for authentication in Section 3. Since this is not offered by DTLS, a separate concept is required. As part of our ongoing research, we are working on a concept that utilizes public-key cryptography, is suitable for low-powered nodes and can handle small message sizes as well as compromised hardware.

After the communication with remote system works as desired, evaluation is required. This includes an analyzation of the network traffic, to see if out fully implemented stack can back up our previous analysis. Furthermore, it is interesting to see if our implementations scales similar to actors systems on desktop hardware.

Finally, we need to consider error handling. Links and monitors require a new implementation as they can not simply track a TCP connections anymore. This goes hand-in-hand with error propagation.

To enhance out network stack, there are several additional drafts for CoAP. Communication over 802.15.4 imposes constraints on the packet size, reducing it to 127 bytes. While IP layer fragmentation is possible, it is not desirable as the loss of a single packet requires the retransmission of all fragments. While this is not a problem with TCP based communication–TCP performs a segmentation of its data streams–the IETF CoRE working group is preparing a draft for allowing fragmentation on the application layer by CoAP block messages [20]. This will allow splitting data into multiple chunks of 2^x bytes, with an exponent x between 4 and 10. The draft specifies two different Block Options, one for requests (Block 1) and one for responses (Block 2) which require up to 3 bytes and manage the exchange of block messages.

Another CoRE draft of interest for future work is the CoAP Simple Congestion Control/Advanced (CoCoA) [21], which proposes an alternative congestion control mechanism for CoAP.

[1] `http://libcoap.sourceforge.net`

[2] `https://github.com/1248/microcoap`

Basic CoAP uses a binary exponential backoff similarly to TCP for retransmitting CON messages. The initial timeout is chosen from the interval of 2 s to 3 s. CoCoA suggest to maintain two estimators, one calculated from messages that required retransmission and one from messages that did not. The timeout is then calculated as a weighted average. Both estimators are initialized to 2 s, and still use a binary exponential backoff when retransmitting.

This strategy leads to a stronger adaption of the timeout to the network characteristics, as the initial timeout rises after messages have been often retransmitted, or drops if they have been acknowledged quickly. Since CoAP allows piggy backing data in ACK messages, this behavior would allow adjusting the retransmit timeout if calculation is necessary to acquire the data.

6 How to Test and Evaluate in the IoT

The functionality of `CAF` as an actor library is well tested and ensured through regular testing and work on the framework itself. Hence, we do not have to concern ourselves with validating `CAF` as an implementation of the actor model. To ensure we did not break core functionality, we can port the tests shipped with the framework to an embedded OS.

A major milestone of this project is the implementation of a new network stack as described in Section 4. Therefore, its functionality has to be validated. A small setup is sufficient to ensure that message exchanges satisfy the protocols and validate packet headers.

Once we are confident that our network stack is implemented correctly, we can move on to test how well it suits our domain. When using IEEE 802.15.4, message sizes are constrained to 127 Bytes with up to 108 Bytes for the payload. Larger messages lead to fragmentation on the IP layer, which can heavily effect our network performance as all packets have to be retransmitted. Our evaluation can show how heavy this impact is depending on the number of message fragments. In case this drives performance into the ground the CoAP extension for block messages could be prove useful [22]. On the other hand, the impact of fragmentation can be magnified by packet loss, which is common in this environment.

A major source for packet loss is the interference form nearby nodes. As such, the number of nodes and the topology have impact on the packet loss. The work of Betzler et al. [23] already examined the packet loss based on different topologies and introduced an alternative algorithm for congestion control.

There are different resources to consider in our evaluation. The first one is processing power, which can be measured in CPU cycles. Basic actions, such as sending messages or context switches between actors, each consume CPU. The addition resources we consume can then be measured by the additional CPU cycles we require to perform an action, for example a context switch between actors compared to a context switch between threads. Furthermore, energy consumption should be evaluated. IoT devices often contain sleep cycles, during which energy consumption should drop accordingly. On the other hand, there are many implementation details that can influence energy consumption. For example, while in many cases busy waiting is considered inefficient, there are situations where context switch requires more resources.

Memory is only a secondary concern as the available memory is expected to keep growing in the next years. As long as our library is small enough to be deployable on our testbeds reducing memory requirements is more of an optimization than a requirement.

One of the questions in Section 5.1 concerns the suitability of the actor model for expressing typical application scenarios. Abstraction and suitability are not straight forward to measure. Sivieri et al. [24] measure the expressiveness of their Erlang framework ELIoT by implementing different protocols for distributed embedded applications and comparing the uncommented lines of code. They use Contiki and TinyOS as established IoT platforms for comparison.

A challenge for tests in IoT environments in reproducibility. The test environment heavily impacts the outcome of results, most often not in a deterministic way. As noted above, a major concern is interference which may stem from other nearby nodes as well as sources outside of the network. Therefore, predictions are often unreliable. To ensure applications work as intended experimentally driven testing is important, i.e., test software often and under changing conditions.

6.1 Testbeds

We have several test environments at hand. Each has different availability and effort requires to deploy. While the environment easiest to use does not resemble a realistic environment, the deployment that requires the most effort offers more realistic conditions. Hence, our testbeds range from comfortable and fast to realistic and slow.

There are two testbeds available in our lab. RIOT, the embedded OS we use, provides a *native port* which emulates the OS on a desktop machine. It only offers quick test to see if the program compiles or to test specific functionality. Furthermore, we have small number of Raspberry Pi nodes in our lab. They are powerful enough to run Linux and provide a good test environment when implementing the new networks stack. We acquired USB dongles to enable 802.15.4 connectivity for the Pis as well as desktop PCs. In addition, we have a few embedded devices in our lab that run RIOT. This heterogeneous setup is useful for a proof-of-concept that validates our network stack. However, it does not provide enough nodes to setup different topologies and observe highly distributed applications.

The FU Berlin owns a larger testbed. It offers up to 60 nodes distributed over several rooms, floors and buildings. Even more nodes are deployed in the INRIA FIT IoT-LAB [25]. It deploys more than 2500 nodes throughout France, which are available to the public and can be booked online. Further, the IoT-LAB offers monitoring of energy consumption and network metrics.

7 Conclusion

Developing applications for the Internet of Things requires a lot of specialized knowledge. The IoT is a highly distributed environment that relies on network communication and synchronization. This work presented an approach to develop applications for the IoT domain from a high abstraction level. Raising the level of abstraction will lead to an improvement in robustness and code quality. Furthermore, it supports generalization and professionalization of the development process.

Our approach introduces the `C++ Actor Framework` to the IoT domain. We proposed a new, adaptive communication layer for `CAF` to enable its use in constrained networks. It is based on the request-response layer from CoAP to provide a transactional message exchange. To ease deployment and portability, we are working to enable `CAF` on the embedded OS RIOT. While actors already run on single nodes, they cannot communicate yet, as the network stack is not implemented.

An outline of upcoming work was given in Section 5.3. The next major task is the implementation of the transactional message passing layer. A local testbed allows validating the protocol implementations before completing the port to RIOT. In general testing is important when developing applications for the IoT because reproducibility is not a given in this domain. Instead, experimentally driven testing is important.

Finally, the design concept of actor-based IoT applications is another important topic. For example, how many actors should be deployed on a single node and is there a structure to build applications this way?

References

[1] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments," in *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2013.

[2] N. Kushalnagar, G. Montenegro, and C. Schumacher, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals," IETF, RFC 4919, August 2007.

[3] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," IETF, RFC 7252, June 2014.

[4] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," IETF, RFC 6347, January 2012.

[5] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. of the 32nd IEEE INFOCOM. Poster.* Piscataway, NJ, USA: IEEE Press, 2013.

[6] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proceedings of the 3rd IJCAI.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[7] G. Agha, "Actors: A Model Of Concurrent Computation In Distributed Systems," MIT, Cambridge, MA, USA, Tech. Rep. 844, 1986.

[8] G. Agha, I. A. Mason, S. Smith, and C. Talcott, "Towards a Theory of Actor Computation," in *Proceedings of CONCUR*, ser. LNCS, vol. 630. Heidelberg: Springer-Verlag, 1992, pp. 565–579.

[9] J. Armstrong, "A History of Erlang," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III).* New York, NY, USA: ACM, 2007, pp. 6–1–6–26.

[10] Typesafe Inc., "Akka," akka.io, March 2012.

[11] D. Charousset, R. Hiesgen, and T. C. Schmidt, "CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications," in *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2014.

[12] M. Vallentin, D. Charousset, T. C. Schmidt, V. Paxson, and M. Wählisch, "Native Actors: How to Scale Network Forensics," in *Proc. of ACM SIGCOMM, Demo Session.* New York: ACM, August 2014, pp. 141–142. [Online]. Available: http://dx.doi.org/10.1145/2619239.26314714

[13] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[14] ARM Ltd., "ARM mbed IoT Device Platform," https://mbed.org, November 2014.

[15] Dunkels, Adam and Gronvall, Bjorn and Voigt, Thiemo, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.

[16] Philip Levis and Sam Madden and Joseph Polastre and Robert Szewczyk and Alec Woo and David Gay and Jason Hill and Matt Welsh and Eric Brewer and David Culler, "TinyOS: An Operating System for Sensor Networks," in *Ambient Intelligence.* Springer Verlag, 2004.

[17] T. Markmann, "Performance Analysis of Identity-based Signatures," HAW Hamburg, Dept. Informatik, Technical Report, August 2014.

[18] C. Schmitt and B. Stiller, "Two-way Authentication for IoT," IETF, Internet-Draft – work in progress 01, December 2014.

[19] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," IETF, RFC 5280, May 2008.

[20] C. Bormann and Z. Shelby, "Blockwise transfers in CoAP," IETF, Internet-Draft – work in progress 16, October 2014.

[21] C. Bormann, A. Betzler, C. Gomez, and I. Demirkol, "CoAP Simple Congestion Control/Advanced," IETF, Internet-Draft – work in progress 02, July 2014.

[22] C. Bormann and Z. Shelby, "Blockwise transfers in CoAP," IETF, Internet-Draft – work in progress 15, July 2014.

[23] A. Betzler, C. Gomez, I. Demirkol, and J. Paradells, "Congestion Control in Reliable CoAP Communication," in *Proceedings of the 16th ACM International Conference on Modeling, Analysis & Simulation of Wireless and Mobile Systems*, ser. MSWiM '13. New York, NY, USA: ACM, 2013, pp. 365–372.

[24] Alessandro Sivieri and Luca Mottola and Gianpaolo Cugola, "Drop the phone and talk to the physical world: Programming the internet of things with Erlang," in *SESENA'12*, 2012, pp. 8–14.

[25] INRIA, "FIT/IoT-LAB," https://www.iot-lab.info, November 2014.

[26] R. Hiesgen, D. Charousset, and T. C. Schmidt, "Embedded Actors – Towards Distributed Programming in the IoT," in *Proc. of the 4th IEEE Int. Conf. on Consumer Electronics - Berlin*, ser. ICCE-Berlin'14. Piscataway, NJ, USA: IEEE Press, Sep. 2014, pp. 371–375.