Embedded Actors – A Better Abstraction for Distributed Messaging in the IoT

Project 1

Raphael Hiesgen

Hamburg University of Applied Sciences

February 10, 2015

The number of devices connected to the Internet of Things (IoT) is rising fast. 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.

Developers 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 by using the actor model to develop applications for the IoT, because they can focus on the application logic instead of spending time on implementing low-level primitives.

We contribute the `C++ Actor Framework` *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. The communication in* `CAF` *is built for locally distributed multicore machines and 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) [16], the Constrained Application Protocol (CoAP) [25] and the Datagram Transport Layer Security protocol (DTLS) [22]. Currently,* `CAF` *is ported to RIOT [5], the friendly operating system (OS) for the IoT.*

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. Participating nodes perform machine-to-machine communication for the sake of a common task. These devices are often constrained while connected to the global Internet. Although individual nodes can only perform simple jobs, a networked distributed system enables the processing of complex duties. The composition of many cooperating devices leads to a highly distributed work flow.

Traditional application scenarios include sensor networks, which can collect data such as environmental or cyber-physical conditions. In recent years, the sizes of nodes decreased dramatically. Besides sensors, IoT networks include actuators that can influence their environment, often in a very limited way. Build from these nodes are complex applications that enable home automation as well as tracking of health data among other things. These systems enable machines to upload data to Internet servers, a task that originally required human interaction. Thus, they allow the tracking of data everywhere and anytime.

Traditional programming and runtime environments are too heavy-weight for this constrained environment. While stronger devices bring sufficient resources to handle runtimes like virtual machines, constrained devices often lack the memory to do so. In those cases, the available memory has to be managed carefully and the developer may not want to hand this control to a garbage collector. As a result, many developers fall back to low-level, hardware-specific programming. This requires specialized knowledge and introduces complexity and multiple sources for errors.

In this work, we propose the actor model to develop software for IoT environments. It provides a high level of abstraction to program concurrent and distributed systems and includes network transparent message passing, as well as an error model designed for distributed systems. Specifically, we contribute the `C++ Actor Framework` (`CAF`) [10], an open-source implementation of the actor model in C++11. The framework focuses on the development of parallel or distributed applications. It features type-safe actor interfaces and a scalable work-stealing scheduler. In addition, we are working to provide runtime inspections tools that enhance distributed debugging.

The remaining article is structured as follows. In Section 2, we introduce actors as a concept for programming concurrent and distributed systems, as well as the `C++ Actor Framework`. Section 3 examines the challenges we face when adapting actors to the IoT environment. The following Section 4 takes a look at available protocol standards for constrained environments that we can build on. Section 5 outlines a first design approach to a communication architecture tailored to the needs of IoT applications. In

Section 6, we present a prototypic implementation and report on problems we encountered. Subsequently, Section 7 compares the message exchange of our new implementation to the TCP-based one. Finally, Section 8 draws a conclusion and presents open questions regarding future work.

2 The Actor Model

The actor model defines entities called actors. Actors are concurrent, isolated and solely interact via network transparent message passing based on unique identifiers. As a result, actors cannot corrupt the state of other actors. The actor model is designed to avoid race conditions.

Each actor can create new actors by an operation called `spawn`. This is often used to distribute workload, e.g., in a divide and conquer approach. An actor divides a problem and spawns new actors to handle the subproblems concurrently. Thereby, the created actors can divide and distribute their problems further.

To detect and propagate errors in local as well as distributed systems, actors can monitor each other. When a monitored actor terminates, the runtime environment sends a message containing the exit reason to all monitoring actors. A stronger coupling of lifetime relations can be expressed using bidirectional links. In a linked set of actors, each actor will terminate with the same error code as its links. As a consequence, links form a (sub-) system in which either all actors are alive or have failed collectively. This allows developers to re-deploy failed parts of the system at runtime and prohibits invalid or intermediate states.

Hewitt et al. [12] proposed the actor model in 1973 to address the problems of concurrency and distribution. Later, Agha focused on theoretical aspects in his dissertation [1] and introduced mailboxing for processing actor messages. Further, he created the foundation for an open, external communication [2].

Multiple actor-based languages have been developed in the last decades. A typical example is the Erlang programming language introduced by Armstrong [4]. It was designed to build distributed system that run without downtime and originally targeted telephony applications. Actors are included in Erlang in the form of processes of the same characteristics as actors. Erlang provided the first de-facto implementation of the actor model, despite using a different vocabulary. Scala is a programming language that includes actors as part of its standard distribution through the Akka framework [27]. It offers object oriented as well as functional programming and runs in Java virtual machines (JVM).

With the advent of multicore machines and cloud computing, the actor approach

has gained momentum over the last decade. In this environment, scalability and fault-toleranency are important traits.

2.1 CAF: A Native Actor System for C++

Our C++ framework [9] (CAF) allows development of native software at the highest abstraction layer possible without sacrificing performance. To make optimized use of system resources, we provide different implementations of the runtime environment exchangeable at compile time. In this way, our framework can scale up to large installations in performance-critical applications [28], while at the same time it is able to scale down to embedded devices. In the context of the IoT, this flexibility allows developers 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. The effort of testing and debugging applications can be significantly reduced by using the type-safe messaging interfaces provided by CAF, as they eradicate a whole category of runtime errors.

The best known implementations of the actor model, Erlang and Akka, are both implemented in languages that rely on virtual machines. In contrast, our actor library is implemented in C++ and thus compiles to in native code. C++ is used across the industry from high performance computing installations running on thousands of computing nodes all the way down to systems on a chip. With CAF, we propose a C++ framework to fill the gap between the high level of abstraction offered by the actor model and an efficient, native runtime environment.

In our system, actors are created using `spawn`. The 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 [6]. 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.

Unlike other implementations of the actor model, we differentiate between *addresses* and *handles*. The former is used for operations supported by all actors such as monitoring. The latter is used for message passing and restricts messages to the specified messaging interface in case of strongly typed handles, or allows any kind of message in case of untyped handles. Access to local and remote actors is transparent. There is no differentiation between them on the API level, thus hiding the physical deployment at runtime.

Actors can communicate asynchronously by using `send` or synchronously by using `sync_send`. While the runtime does not block in both cases, `sync_send` lets the user synchronize two actors. When sending such a message, a message handler can be set

to handle a response. Then, the sending actor waits until a response is received before processing other incoming messages. Alternatively a timeout can be specified to return to the previous behavior if no response is received. In case of errors, e.g. when the receiver is no longer available or does not respond, an error message will be sent to the sender and cause it to exit unless a custom error handler is defined.

Messages are buffered in the mailbox of the receiver in order of arrival before they are processed using the designated behavior. A behavior in our system consists of a set of message handlers. These 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.

Originally, `CAF` maintained a lookup table for type information per connection. When a connection between nodes was initiated, the nodes announced their message types individually assigning unique IDs. These information were used to annotate messages and thus slim down the message headers instead of including the complete information each time. This stateful type compression approach reduced the meta information to a bare minimum.

In the `CAF` release 0.10.0, the network and communication has been redesigned. The new design allows communication via a third node, which eases propagation of actors. We also removed the exchange of type information as mentioned above, because it significantly increased the management of lookup tables in `CAF`. As a result, all meta information are sent along with the actual data, resulting in slightly increased packet size.

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.

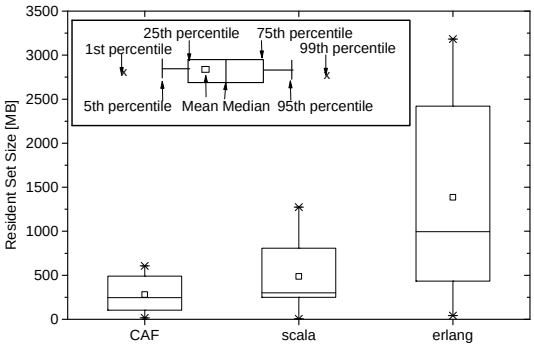


Figure 1: Memory consumption for the creation of 2^{20} actors

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 [10].

3 Challenges for Actors in the IoT and Related Work

Networks in the IoT are built from a large number of nodes. These, potentially low-powered, embedded devices are connected through Internet protocols and rely on a distributed workflow that is built on machine-to-machine communication. Hardware constraints include low memory and processing power as well as constrained battery sizes. In addition, small packet sizes, packet loss due to interference and temporary connection failures need to be considered.

Developers have to deal with a highly distributed application design while taking the potentially constrained hardware into account. Message exchange and synchronization as well as error propagation and mitigation have to be considered. Managing these problems is a complex, error-prone task because specialized knowledge from multiple domains is required. 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. Using a runtime environment that is built to meet these challenges eases the development process and allows to keep focus on the application domain.

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 for distributed applications. However, the IoT domain introduces new sources of errors that are not present in the traditional applications of the actors systems. Traditional actor environment 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

First of all, the communication between nodes should be encrypted to protect the privacy of the exchanged data. Naturally, the encryption deployed on constrained hardware must be strong enough to resist crypto attacks on desktop grade systems or clusters. Next, wireless communication cannot only be easily captured, but messages can also be injected into the network. Authentication of nodes is necessary to prevent malicious nodes from joining our network. Deciding which node to trust is a nontrivial challenge. A pre-shared key allows signing messages appropriately, but can be obtained if with the hardware is compromised. Hence, it is necessary to detect such incidents and revoke trust. A common way to track revoked trust is a central node which needs to be accessible at all times, a characteristic that cannot be guaranteed. Lastly, authorization ensures that services are only offered and used by the designated nodes, either within our trusted network or offered openly.

There are several existing approaches to ease the development IoT application by providing a higher level of abstraction. Instead of implementing a runtime environment, a domain-specific language (DSL) can be used to allow code-generation for embedded operating systems. An advantage of this approach is a low memory demand of the target platform as native code is generated and neither a runtime environment nor a virtual

machine are needed. The work of Mainland et al. [18] presents Flask, a Haskell-based DSL targeted at sensor networks. Flask includes a restricted subset of the Haskell language, called Red, and a extensible runtime with support for commonly-used services. Red allows the combination of Haskell with `nesC`, a `C` dialect used by TinyOS, which gives access to existing code and TinyOS. Further, Flask includes high-level library to develop applications as well as a staging mechanism that separates node-level code from the meta-language used to generate it. With their work the authors target low-powered hardware, including 16-bit CPUs with tens of kilobytes of RAM. The paper explains the development of Flask and goes into detail on the capabilities of the language Red.

Going a step further than a DSL, there are several projects that make use of the actor model to provide a higher abstraction for the development of embedded applications. The actor model itself is already designed to model and develop distributed applications and provides a solid foundation to built upon. Making use of a virtual machine (VM) provides benefits with regard to hot-swapping of code and platform independency, if an implementation of the VM is available. A drawback is the memory requirements for the VM as well as the usual garbage collecting, which leads to unpredictable resource usage. Erlang was originally targeted at the infrastructure of telephony and provides an implementation of the actor model. Sivieri et al. [3] present the Erlang based ELIoT framework with a focus to develop decentralized systems for the Internet of Things. They provide an adjusted communication layer, an interpreter striped of unnecessary features and a simulation environment that allows a transparent migration to real hardware. Another approach that makes use of the actor model in sensor networks is actorNet developed by Kwon et. al [17]. In addition to an embedded Scheme interpreter as an alternative to low-level languages such as NesC/C, the paper presents a virtual memory manager that extends the available SRAM and includes optimized garbage collection. Furthermore, actorNet includes basic multitasking capabilities. See the paper for an implementation on Mica2 nodes and an evaluation.

Many protocols used in IoT environments are have been specified by the IETF, mainly DTLS, UDP and 6LoWPAN. While encryption and integrity can be provided through DTLS, there is no goto concept for authentication and authorization. An IETF working group called Authentication and Authorization for Constrained Environments (ACE) is working to change this. Their draft [23] outlines a straight forward approach, which originates in the work of Kothmayr et al. [15]. It proposes authentication by using public-key cryptography with X.509 certificates [11] signed by a Certificate Authority (CA). Furthermore, a resource-rich Access Control Server (ACS) is deployed as a trusted entity in the network. It stores rights for nodes and distributed access tickets to prove authentication and authenticity. This work is currently continued as an IETF draft [23]

in the working group Authentication and Authorization for Constrained Environments (ACE). Depending on device strength the draft suggest to use either 2048 *bit* RSA for stronger devices or elliptic curve cryptography (ECC) for low-powered ones.

4 Standards for the Internet of Things

When designing a communication layer for the IoT, unreliable links and infrastructure failures need consideration in the design. Several protocols have been developed and standardized to approach these challenges. While this section presents the protocols we want to use in our design, Section 5 presents the complete design of our network stack for the IoT.

4.1 IPv6 over Low-power Wireless Personal Area Networks

IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN) [16] enables Internet connectivity on constrained links. It is often used in conjunction with the low PAN radio standards IEEE 802.15.4 [14] or Bluetooth LE [20]. Traditional Internet protocols require an effort that cannot be met by most embedded devices and a minimum frame size of 1280 bytes [24]. 6LoWPAN addresses these problems while maintaining compatibility to IPv6. A stateless translation between 6LoWPAN and IPv6 can be done by edge routers.

Instead of the minimum 1280 bytes size required by IPv6, IEEE 802.15.4 frames have a length of 127 bytes. With 25 bytes frame overhead, 102 bytes remain for IP, a transport protocol and the payload. The IP header requires 40 bytes and UDP header 8 bytes. This would leave a maximum of 53 bytes for the payload. 6LoWPAN includes stateless as well as statful header compression [14] to allow a payload of up to 108 bytes. For example, the 128 bits for the source and destination address can be compressed to either 64 bits, 16 bits or even 0 bits, depending on the information available from the context.

4.2 The Constrained Application Protocol

The Constrained Application Protocol (CoAP) [25] is an application layer protocol designed for IoT environments. It defines a request-response model adapted from HTTP, but tries to avoid its complexity. As such, it implements the GET, PUT, POST and DELETE methods known from HTTP. However, CoAP works asynchronously and is designed to be used via UDP.

Requests can contain a method code to request an action or a URI to identify a resource. Responses contain a status code and optionally a resource. Four messages types are defined: Confirmable (CON), Non-confirmable (NON), Acknowledgement (ACK),

Reset (RST). CON and NON messages are used for requests and responses, in addition a responses can be piggy-backed in an ACK message. CoAP integrates the following two layers in a single protocol.

The *messaging layer* deals with datagrams and their asynchronous nature. In both, requests and responses, a 4 bytes binary header is followed by compact binary options and a payload. Messages contain a 2 bytes message ID for detecting duplicates. The ID is also used to provide reliability in CON messages, which are retransmitted after a timeout until an ACK with the same message ID is received. RST messages are used to respond to messages that could not be processed.

The *request-response layer* is embedded in the messaging layer. Each message contains either a method or response code, optionally request or response information and a token that is used to match responses to requests independently of the messaging layer. A single request may lead to multiple responses, e.g., when using multicast.

4.3 Datagram Transport Layer Security

Transport Layer Security (TLS) [26] is the most common security protocol on the Internet and among other things used to secure mail and web traffic. The protocol ensures confidentiality, integrity for exchanged data as well as server authentication via certificates. Because TLS requires a reliable transport protocol, it does not work over datagram protocols such as UDP. The Datagram Transport Layer Security protocol (DTLS) [19] is adapted from TLS and implements encryption and integrity for datagram protocols [22]. As such, it handles reliable message transfer and packet reordering for the initial handshake.

It should be noted that DTLS does not provide authentication. Authentication can implemented via public-key cryptography using signed messages, i.e., RSA or alternatively elliptic curve cryptography (ECC) to reduce signature sizes.

5 Messaging Architecture Between Actors

We want to deploy a transactional message-passing layer based on the request-response model offered by CoAP (Section 4.2). 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.

Figure 2 shows our transactional network stack for the IoT compared to the default TCP-based implementation in `CAF`. The TCP-based stack uses TCP and IP over Ethernet or WLAN. Although it does not support TLS or HTTP at the moment, the layers are included in the figure to match the IoT side. In contrast, the IoT stack is based on the protocols discussed in Section 4. On the network access layer we target IEEE 802.15.4 or Bluetooth. The IP layer deploys 6LoWPAN to keep IPv6 compatibility while the transport layer uses UDP. With regard to security we want to rely on DTLS for encryption. Further, we are working on an authentication concept for IoT environments. 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.

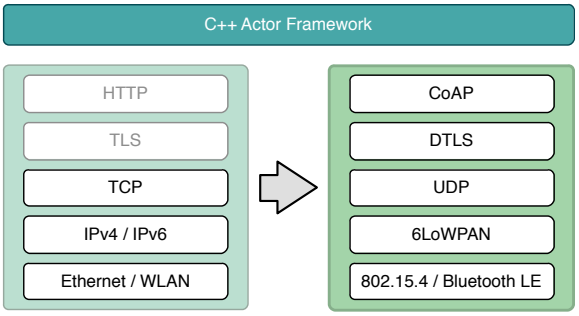


Figure 2: The network stack used in `CAF`, the present one on the left, and the IoT stack on the right.

The overhead of meta information in message could be reduced by using a stateless compression algorithm optimized for short strings. A stateless approach is also applicable to scenarios in which the receivers are usually unknown. In this way, self-consistent messages allow us to make use of features such as the group communication extension for CoAP [21] in the future.

Another concern with constrained message sizes is the delivery of large packets that require fragmentation. If this fragmentation is done on the IP layer, all fragments have to be resend if a single one is lost. A CoAP draft [8] aims to solve this problem by fragmenting messages on the application layer and only retransmitting the lost fragment.

6 Implementation

In `CAF`, network-layer communication is encapsulated by the `Middleman` that multiplexes socket IO and manages proxy instances for remotely running actors. Whenever a socket

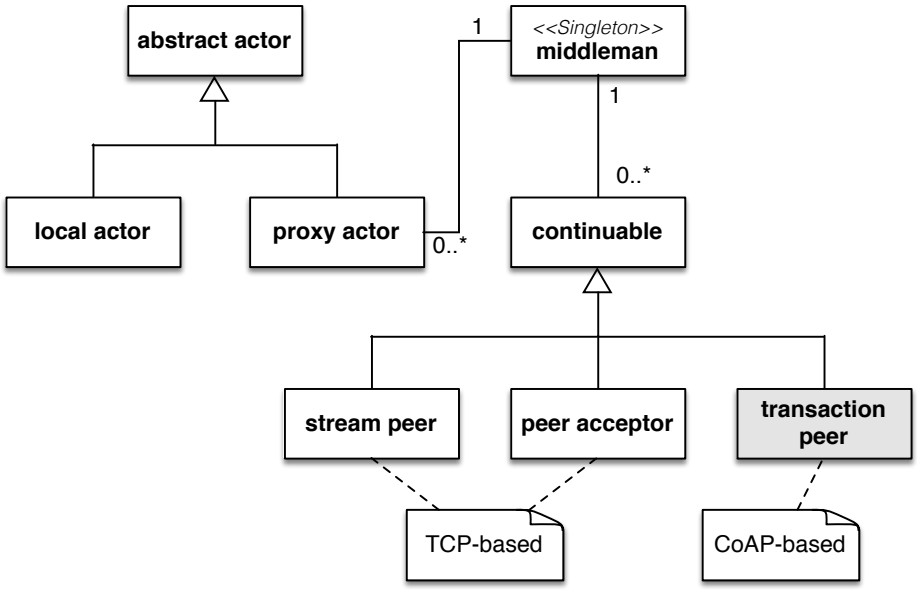


Figure 3: UML depicting the relationship of actors, the middleman and peers in `CAF`.

becomes ready for read operations, the `Middleman` runs the callback of the object that implements the `Continuable` interface associated with the socket. `CAF` does not allow access to the `Middleman` from its user-level API. Instead, the two functions `publish` and `remote_actor` are provided to initiate message passing to other nodes in the network.

The TCP-based default backend of `CAF` is implemented in the two classes `stream peer` and `peer acceptor` which both implement the `Continuable` interface. The former handles an established TCP connection, whereas the latter is listening on a local port to accept incoming connects. Once a connection has been accepted and the initial handshake succeeded, the `peer acceptor` delegates the socket to a new `stream peer` instance. Since `CAF` uses logical identifiers for each node rather than protocol-specific locators, redundant connections can only be detected after the initial handshake. Whenever a redundant connection is detected, the `stream acceptor` discards the accepted socket and re-uses the existing `stream peer` for the requested operation.

Our transaction-based communication stack for the IoT is implemented in the class `transaction peer` that implements the `Continuable` interface as well. A UML representation of the software design including all relevant classes is shown in Figure 3. For a consistent API design, we named the front-end functions `coap_publish` and `coap_remote_actor`—similar to the default functions `publish` and `remote_actor`. In the final implementation a name that hints at the transactional functionality will be a better fit.

Unlike TCP, UDP does not require to accept new connections. It allows receiving from all communication partners on a single socket. To differentiate between different endpoints, our `transaction peer` sorts actor identifiers according to their host address and port. In combination with an actor ID, these information can be used to address

actors later on.

The handshake implemented in the CoAP-based peer differs from the TCP-based peer in several ways. The CoAP-based implementation does neither require the TCP handshake nor the type exchange of the TCP-based implementation. This reduces the amount of packets transported between peers. The omitted type exchange results in an increased packet size, because all messages are self contained. Since the handshake is vital, the exchange uses CON messages to achieve reliability.

The current implementation of `CAF` creates a monitor whenever a connection to a remote peer is established. However, the information is only used by the runtime environment after the developer created a monitor or link that then triggers the runtime environment to forward these information. Regarding optimizations, this behavior could be changed to be an explicit operation and thus avoid the message overhead and resources to maintain the monitor if it is not required.

7 Evaluation

In a first analysis, we compared the packet flow of the CoAP-based communication backend to the TCP-based default backend [13]. For our evaluation, we consider a sensor data collection scenario, where a node periodically sends data to a sink. In this IoT typical use case, we focus on measuring the number of packets sent as well as the packet sizes.

We use a robust setup based on Ethernet, since we solely focus our analysis on conversational characteristics. Our setup consists of a Raspberry Pi that forwards data to a desktop PC, both running Linux. The tests are based on the `CAF` version 0.9.4 and still include the type exchange in the TCP-based backend.

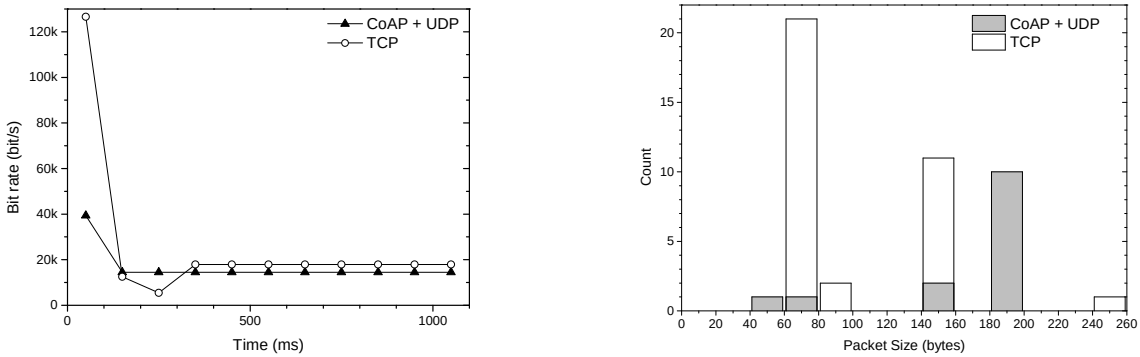
The client periodically reads data from a sensor and forwards it to the server. Using `CAF`, This behavior can be implemented in few lines of code with a high level syntax, as shown below. As a first argument, the client expects a pointer to the actor itself and a handle to the server collecting the data as a second argument. Similar to the `this` pointer for objects, `self` is a handle to the current actor and provides access to actor specific functionality, such as sending messages. The functions returns a set of message handlers for incoming messages, which are later used by the runtime to handle messages that address this actor. In this case, we only specify one message handler that the actor itself triggers every 100 ms by sending delayed messages to itself. Time-based, message-driven loops like this are a common idiom in actor programming and provide a simple and convenient way to poll a resource periodically. Instead of reading data from a sensor, we sent predefined messages with a payload of 10 $bytes$.

```
1 behavior client(event_based_actor* self,
```

```
2                   const actor& server) {
3     return {
4         on(atom("next")) >> [=] {
5             auto sensor_data = read_data();
6             self->send(server, sensor_data);
7             self->delayed_send(self,
8                 chrono::milliseconds(100),
9                 atom("next")
10            );
11        }
12    };
13 }
```

For our evaluation, we recorded the TCP and UDP network traffic between our two test systems for a period of one second. Network traffic measurements include the initial `CAF` handshake as well as all protocol messages, such as ACKs from TCP. Figure 4(a) shows the effective bit rates of each channel as functions of the elapsed time. Figure 4(b) depicts the distribution of packet sizes for the same experiment. During the initial 100 ms interval of our experiment, a very high bit rate becomes visible. This is due to the different handshakes. The TCP-based implementation performs a TCP handshake, followed by a handshake between the peers and the exchange of type information. In comparison, our CoAP-based backend only requires the handshake between peers and omits the type exchange. This reduces handshake costs to less than 30 %. After the initial phase, both implementation show a constant stream of messages triggered by the 100 ms interval of our message-driven loop. The TCP-based implementation requires two messages per sensor data as TCP acknowledges each packet. In contrast, our CoAP-based implementation sends only a single message per sensor data. This difference in message count is stressed by Figure 4(b). The use of CoAP halves the number of messages and slightly decreases the bandwidth usage from 18 000 $bits/s$, which includes packets as well as the TCP ACK, to 14 500 $bits/s$ per message exchange.

Examining the distribution of message sizes in Figure 4(b) reveals a limit of 200 $bytes$ for the CoAP case. This becomes all the more important when using 6LoWPAN, where an effective MTU size on the transmission media of about 100 $bytes$ is likely. Larger frames require fragmentation and thereby increase load and sensitivity towards packet loss. As visible from the standard case, the impact of an upfront type exchange may have a heavy impact on the available payload size and increase the need for fragmentation.



(a) Raw bit rate over time

(b) Distribution of packet sizes

Figure 4: Recorded network traffic for the first second of runtime

8 Conclusion and Future Work

This work presented an approach to develop applications for the IoT at a higher abstraction level using the actor model. Programming the IoT with an appropriate development environment will increase quality and robustness of code and reduce code dependencies on specific hardware or systems, all of which is urgently needed in the IoT domain. We proposed a new, adaptive communication layer for the `C++ Actor Framework` to enable its use in constrained networks. An evaluation showed that the message-oriented programming style proposed in our software development platform fits the frame exchange layer provided by CoAP.

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. We working to enable `CAF` on RIOT. While simple applications do already work, the support is very limited for now. Hence, the next step is to provide full support for `libcaf_core`, which includes actors and local interactions.

Once the core of `CAF` is running, network capabilities are desirable. The network stack requires UDP and CoAP to build the transactional layer. There two approaches to implement this in `CAF`. Either, both can be implemented as separate layers which can be combined to the transactional layer later on, or the transactional layer is offered as an abstraction over both while hiding the implemented protocols. In both cases, the implementation should allow an substitution of CoAP for HTTP, which can offer similar

transactional semantics not targeted at IoT environments.

The UDP implementation is straight forward based on the Berkeley socket API. CoAP on the other hand is standardized, but lacks a goto implementation. For the prototype, we used the open source library libcoap. However, it required adjustments to be usable in our context. Further, it does not offer a `C++` wrapper and as such offers a very low-level API. Before implementing CoAP ourselves, research is necessary to see if newer libraries with a suitable interface are available by now. 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 [8]. 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) [7], which proposes an alternative congestion control mechanism for CoAP. 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.

References

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

[2] Agha, G., Mason, I. A., Smith, S., and Talcott, C. Towards a Theory of Actor Computation. In *Proceedings of CONCUR* (Heidelberg, 1992), vol. 630 of *LNCS*, Springer-Verlag, pp. 565–579.

[3] 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.

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

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

[6] Blumofe, R. D., and Leiserson, C. E. Scheduling Multithreaded Computations by Work Stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748.

[7] Bormann, C. CoAP Simple Congestion Control/Advanced. Internet-Draft – work in progress 01, IETF, February 2014.

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

[9] Charousset, D., Hiesgen, R., and Schmidt, T. C. 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, Oct. 2014), ACM.

[10] Charousset, D., Schmidt, T. C., Hiesgen, R., and Wählisch, M. 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, Oct. 2013), ACM.

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

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

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

[14] Hui, J., and Thubert, P. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282, IETF, September 2011.

[15] Kothmayr, Thomas and Schmitt, Corinna and Hu, Wen and Brünig, Michael and Carle, Georg. DTLS Based Security and Two-way Authentication for the Internet of Things. *Ad Hoc Netw. 11*, 8 (Nov. 2013), 2710–2723.

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

[17] Kwon, YoungMin and Sundresh, Sameer and Mechitov, Kirill and Agha, Gul. ActorNet: An Actor Platform for Wireless Sensor Networks. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* (New York, NY, USA, 2006), AAMAS '06, ACM, pp. 1297–1300.

[18] Mainland, Geoffrey and Morrisett, Greg and Welsh, Matt. Flask: Staged Functional Programming for Sensor Networks. *SIGPLAN Not. 43*, 9 (Sept. 2008), 335–346.

[19] Nagendra Modadugu and Eric Rescorla. The Design and Implementation of Datagram TLS. In *In Proc. NDSS* (2004).

[20] Nieminen, J., Savolainen, T., Isomaki, M., Patil, B., Shelby, Z., and Gomez, C. Transmission of IPv6 Packets over BLUETOOTH Low Energy. Internet-Draft – work in progress 12, IETF, February 2013.

[21] Rahman, A., and Dijk, E. Group Communication for CoAP. Internet-Draft – work in progress 19, IETF, June 2014.

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

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

[24] Shelby, Z., and Bormann, C. *6LoWPAN: The Wireless Embedded Internet*, 1st ed. Wiley Publishing, 2009.

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

[26] Turner, S., and Polk, T. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176, IETF, March 2011.

[27] Typesafe Inc. Akka. `http://akka.io`, March 2012.

[28] Vallentin, M., Charousset, D., Schmidt, T. C., Paxson, V., and Wählisch, M. Native Actors: How to Scale Network Forensics. In *Proc. of ACM SIGCOMM, Demo Session* (New York, August 2014), ACM, pp. 141–142.