Loosely Coupled Communication in Actor Systems

Master Seminar II

Raphael Hiesgen

Hamburg University of Applied Sciences

August 31, 2014

Contents

1 Introduction

The Internet of Things (IoT) describes a network of interconnected nodes often connected to the Internet. These nodes rely on machine-to-machine communication to perform a common task [1]. Applications include sensor networks as well as smart home devices. While individual nodes can only process simple jobs, a highly distributed work flow enables the processing of complex duties. The IoT enables machines to upload data to the Internet, a task that originally required human interaction, and thus allow tracking of data anywhere and anytime.

Developing applications for this distributed work flow strongly depends on coordinating the participating nodes. This includes the exchange of data, synchronization as well as the propagation and mitigation of errors. Handling these problems is a complex and error-prone task. This process can be eased by using a framework that handles the communication infrastructure. The actor model is a natural fit for the message-driven work flow in IoT applications and an efficient middleware layer based on this model can serve as a productive and scalable development environment. We contribute the `C++ Actor Framework` [1], previously named `libcppa` [2].

The actor model can not be applied to this new domain without adjustments [3]. It does not address the lossy links or low-powered hardware that is often present in the IoT. New challenges must be handled, such as connection failures due to erroneous transmissions or sleep cycles. Further, participants cannot rely on a strong coupling that was originally part of the actor model. The error handling mechanisms known from the actor model are also not applicable as they do not The distributed error handling capabilities were not designed with these constraints in mind and require adjustment. Finally, the actor model leaves security considerations to the runtime environment. Since wireless communication is a given, messages exchanges should be secured and authenticated accordingly. To meet these challenges we implement a new network stack for `CAF` based on open standards for the IoT.

This work presents three papers that explore different aspects of this topic. Section 2 explores the aspects of loose coupling. The following Section 3 examines an extension to the Constrained Application Protocol (CoAP), that implements an improved congestion control algorithm. Section 4 presents an Erlang-based framework with a similar goal, to raise the abstraction level for software development in the IoT. Finally, Section 5 draws a conclusion and presents our next steps.

2 Why is the Web Loosely Coupled?

The paper "Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design" was written by Pautasso et al. in 2009 [4]. Loose coupling is often stated to be a desirable characteristic and associated with a limited impact of change and the possibility for services or components to evolve independently. However, a specific definition is missing. In their paper, the authors explore the origins of loose coupling and identify twelve different facets to better define it. They evaluate their findings by examining the Web service technologies RESTful HTTP, RPC over HTTP and WS-* with regard to these facets.

[1]http://www.actor-framework.org

2.1 The Origins of Loose Coupling

The concept of loose coupling first appeared in 1967 in the context organization research. From there on it was adopted by the IT industry and boosted by the positive image it had previously. The authors argue that loose coupling addresses similar problems in both areas: the contrast between constant structure and the need to react to change and new input. Furthermore, it improves the reliability of complex systems because local failures are not propagated as far.

With a reference to computer science, the concept appeared early in the areas of software engineering and distributed systems. In software engineering coupling was used to describe the ability to compose different modules in a software architecture. Loose coupling provides an easier understanding of the system and supports its evolution. In distributed systems coupling is a characteristic of the communication mechanisms between processes or nodes. While the use of shared memory is an example for strong coupling, loose coupling is associated with message passing. Other aspects are reflected by the publish/subscribe pattern, which decouples components in space, time and synchronization.

2.2 Facets of Coupling

Coupling between system or services has different aspects that influence each other. The authors chose the term facet to stress this common bond and identified twelve different facets of coupling.

Discovery The facet of discovery describes how resources are addressed and can be located in a system. Participants of the same system require a shared model of resources and their addresses. A tradeoff in this facet is a central registration vs. a decentralized referral.

An example can be found in the Web. The approach to establish a central registry failed in the past as it could not keep up with the fast growth and change of the web. Instead, hyperlinks prevailed. Search engines can discover content by crawling the web and following hyperlinks to find resources.

Service discovery is an essential component to build a loosely coupled distributed system. In the presence of connection and infrastructure failure, relying on a central registry is not a viable option. As seen in the evolution of the Internet, a central instance also hinders scalability.

Identification When connecting to local or remote systems the representations of entities must be tranlateable between these systems. This includes namespaces, assigning identities and services for lookups, binding and comparison. Similarly to the discovery facet the tradeoff is a central identification service opposed to a specified identification scheme. The strong coupling of a central service provides context to the identities it manages. Outside of this context the rules how these entities are handled are lost. In contrast, loosely coupled approaches do not couple context and identification. A well known example is the Uniform Resource Identifier [5], which is commonly used with the http scheme on the Web.

Binding Resolving names into identifiers is called binding. The difference in coupling is the point in time when the names are resolved. Strong coupling resolves names early, e.g., at deployment time or even at compile time. However, resolving names dynamically and only if necessary leads to loose coupling and is known as late binding. If it is performed before each lookup, it may lead to a bottleneck and thus can turn into strong coupling.

Platform The platform facet stems from dependencies to a specific programming language or platform such as an operating system or framework. Strong coupling imposes requirements or limits the interaction between heterogeneous platforms. These requirements can be loosened through standardized interfaces and protocols.

Interaction The interaction between entities is classified in either synchronous or asynchronous communication. Synchronous interaction requires both entities to be available at the same time and is associated with tight coupling. Asynchronous interaction does not impose this requirement and corresponds to loose coupling. An example for a synchronous protocol is HTTP, which does not work if the addressed web server is not available. However, caches and (reverse) proxies loosen this requirement and allow HTTP to work if the server itself is not available.

Interface Orientation Interfaces can be oriented either horizontally or vertically. A horizontal interfaces is used for the interaction between local components and available through APIs. Although an API may wrap an interaction with a remote system, the interfaces itself is still is a local component that hides locality. In contrast, a horizontal interfaces defines the communication with a component through a protocol. These interfaces can be layered and are often available to the user through a vertical interfaces.

Vertical interfaces favor loose coupling because they do not dictate the local abstraction. On the other hand, horizontal interfaces often depend on a specific middleware.

Model Exchanging data between applications may require a common data model specified by the design. This leads to a tight coupling and often results in a simple serialization of the data. Loose coupling can be implemented through self-contained messages that are exchanged in a standardized format. As a result, the internal model can be easily created from the known format. Although the mapping process includes overhead, it is required to decouple the internal model from the exchanged data format.

Granularity This facet denotes the number of interactions required to perform a task through an API and their complexity. Coarse-grained interfaces require fewer interactions and provide a loose coupling. Changes are not visible to the user as quickly and allow an easier evolution of the API. Furthermore, this minimizes the total latency in high-latency environments. Fine-granular interfaces quickly require adoption when they evolve, but can avoid overhead if the interfaces match the performed task.

State A lot of services require information to keep track of interactions. This facet differentiates between stateless and stateful design. Stateful design indicates tight coupling and creates a shared state by keeping track of interactions at the service. It hinders high throughput and long running transactions when scaling to many parallel interactions. In addition keeping track of these information can lead to a bottleneck. Stateless design keeps track of these information in the exchanged messages to relieve the service of this task and avoid a shared state. Furthermore, management tasks and error handling, i.e., recovering from failure, are easier to implement. In some cases stateful design can be valuable to reduce message size.

Evolution The evolution of services does not only lead to change in the service itself, but can affect user interfaces as well. This facet discusses interface compatibility between different versions. A service is backward compatible if clients with an old version can use new versions of the service and forward compatible if clients with a new version can use an old version of the service. In this context, loose coupling provides forward

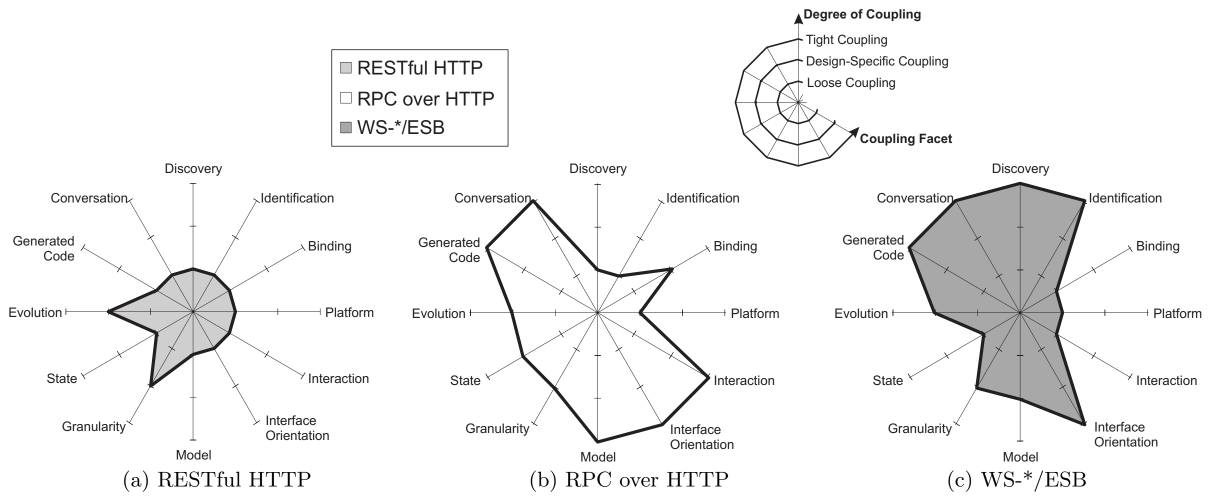


Figure 1: The degree of coupling with regard to the twelve facets found in different Web services, from [4].

and/or backward compatibility while tight coupling offers neither. This facet does not only include the evolution of the service itself, but also its extensibility.

Generated Code Models and services with well-defined interfaces can be used to generate code, e.g., to handle communication. However, this introduces a tight coupling as the code may no longer work if the description changes and has to be generated anew. Further, it may have dependencies to the platform for which it was generated. Instead of static code generation, a loosely coupled design favors declarative mechanism and leaves it to participants to make assumptions or handle communication dynamically. An example for loose coupling is the extension mechanism for MIME content types in Web browsers.

Conversation The interaction with a service or server consists of a sequence of messages forming a conversation. A specific functionality can be achieved by the according exchange of messages. In a tightly coupled design the service tries to ensures that the user can only follow the correct path. This can be achieved through metadata which enables static checking of possible paths. In contrast, a loosely coupled approach allows the dynamic discovery of possible paths at runtime. RESTful Web services use hyperlinks to navigate conversations, which is an example of a loosely coupled design.

2.3 Evaluation

The evaluation examines three Web services with respect to the twelve facets in a quantitative analysis. Figure 1 depicts the degree of coupling for each facet for (a) RESTful HTTP, (b) RPC over HTTP and (c) WS-* based messaging[2] on a enterprise service bus (ESB) [6]. Coupling can be tight, design-specific or loose, ordered from outside to inside. RESTful HTTP shows loose coupling for most facets with only two design-specific facets and ten loosely coupled ones. RPC over HTTP and WS-*/ESB both show five facets with tight coupling, but WS-*/ESB has less design-specific coupling giving it a slight edge over HTTP with RPC.

See the paper for a detailed explanation of the differences between the services.

[2]WS-ReliableMessaging provides reliable message transfer for SOAP messages.

2.4 Discussion

Coupling is of used with the connotation that loose coupling is positive and tight coupling is negatice. However, the authors identified multiple facets of coupling, each one a ranging between loose an tight coupling. Services or systems rarely depict either loose or tight coupling for all facets, but include both for different aspects. Neither loose or tight coupling is simply bad. Depending on the scenario, either can bring an advantages. While loose coupling provides advantages for evolution and agility, tight coupling can improve performance as it relies on fewer indirections.

The evaluation and examples in that paper often reference Web services, which are not in the focus of my work. Of much more interest are the identified facets. They show a broad range of the aspects to consider when design systems as well as discussing about them. The paper brings structure to the topic and enables a better analysis for future considerations.

3 Congestion Control in Reliable CoAP Communication

The Congestion Application Protocol (CoAP) [7] is a draft specified by the IETF and designed machine-to-machine (M2M) communication. 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 is designed to work via datagram protocols such as UDP. It offers reliable message transfer via Confirmable messages (CON) in addition to non-reliable Non-Confirmable messages (NON).

The paper “Congestion Control in Reliable CoAP Communication” was published by Betzler at al. in 2013 [8]. Congestion occurs when the network is overloaded with traffic and leads to longer delays and even packet loss due to buffer overflows. The task of congestion control is to detect and counteract congestion. The authors assume that congestion is a common problem in constrained environments due to limited hardware and link capabilities. Although CoAP does implement a basic mechanism to handle retransmissions for reliable messages and counteract congestion, the CoRE Working Group is developing an alternative approach called CoAP Simple Congestion Control/Advanced (CoCoA) [9] to enhance these capabilities. This paper presents the workings of CoCoA and evaluates its performance in comparison to basic CoAP congestion control.

3.1 Congestion Control in basic CoAP and CoCoA

The confirmable (CON) message type specified in CoAP implements reliability through retransmits. A packet is retransmitted when no acknowledgement (ACK) is received within the retransmission timeout (RTO). However, it is not possible to retrieve a reason for the failure—it may be caused by congestion or the lossy nature of wireless links. The initial timeout is chosen at random from the interval of $[2\,s, 3\,s]$. Since CoAP allows ACKs to include piggy-backed answers, the RTO accounts for the round-trip times (RTT) as well as processing time to handle the request. While choosing a long interval may result in idle processors as they wait too long before retransmitting, a short interval may lead to redundant request and congestion. Retransmission applies a binary exponential backoff

(BEB) to the next RTO in a similar way to TCP. Thus, the interval in which new packets are sent is increased, which leads to fewer packets in the network.

The approach specified in basic CoAP is not flexible. Choosing the timeout from an interval is necessary to avoid simultaneous retransmits due to collisions. However, each new transaction uses the default RTO and thus does not adjust to the network.

CoCoA aims to adapt to the network characteristics. Instead of using the same initial value for each new transaction, it uses two timeout estimators per endpoint, a strong and a weak estimator. The strong estimator tracks the RTT of transactions that required no retransmission and the weak estimator tracks the RTT of retransmitted transactions. Both estimates are used to calculate the final RTO using an exponentially weighted moving average. These formulas were introduced in RFC 6298 [10], but are slightly adjusted in the CoCoA draft. CoCoA changes the initial estimators to $2\,s$ and suggests to weight the weak estimator less than the strong one. Further, the draft proposes to change from the BEB to a variable backoff factor if the initial RTO is outside the interval $[1\,s, 3\,s]$. The BEB is truncated at $32\,s$ and the variable backkoff factor is truncated at $93\,s$.

In addition to the two approaches presented above, the paper uses a variant of the CoCoA calculation for their evaluation. Instead of relying on the two estimators from CoCoA, they present CoCoA Strong (CoCoA-S) which only use the strong estimator to calculate the RTO.

Basic CoAP uses the value NSTART to define the maximum number of parallel transaction with one endpoint. Per default this is constrained to 1 as higher values can lead to congestion. Aiming to provide a better congestion control, CoCoA defines the mechanisms for NSTART greater than 1. Instead of the default $2\,s$ RTO, each additional transaction uses the value of $2\,s$ times the number of parallel transactions.

3.2 Evaluation

The evaluation compares the congestion control in basic CoAP to CoCoA and CoCoA-S by measuring throughput and dropped packets for different network topologies.

Instead of a physical setup the nodes are simulated using the Cooja Simulator. Each node runs Contiki OS, an open source operating system for the Internet of Things (IoT). The deployed network stack consists of IEEE802.15.4, IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN), the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL), UDP and CoAP.

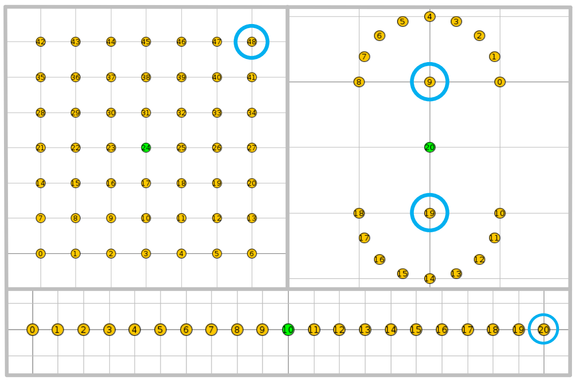
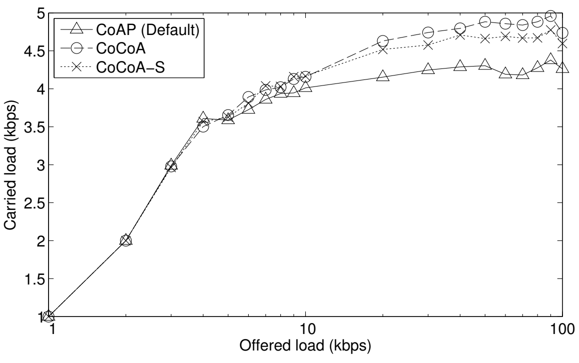
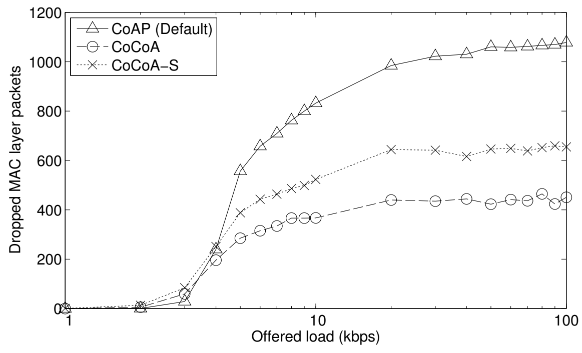


Figure 2: The network topologies for the evaluation (grid, dumbbell and chain). Blue circles are sinks and green nodes are RPL border routers, from [8].

Figure 2 shows the three network topologies used for the measurements. The grid consists of 10 by 10 squares, while each node has a transmission range of 10 and an interference range of 20. Packets are forwarded towards the nodes marked with a blue circle. Nodes that have no routing entry for their destination forward packets towards the green-marked border routers, which are located to be equally accessible by all nodes. The topologies directly affect the amount of neighbors and the interference suffered from them, thus affecting packet loss caused by congestion or interference.



(a) The measured throughput.



(b) Dropped MAC layer packets.

Figure 3: Evaluation of the chain topology with NSTART=1, from [8].

Nodes periodically create CON CoAP messages of 95 bytes size and send them to the sink. Messages that can not be sent due to the parallel transaction limit of NSTART are dropped.

Figure 3 presents the measurements for the chain topology with a value of NSTART=1. Both graphs use a logarithmic scale for the offered load. Unlike the gird or dumbbell topology, the chain topology has longer paths. Further it exhibits an unequal usage of the participating nodes. Nodes that are located closer to the source are used more frequently.

The achieved carried load is depicted as a function of the offered load in Figure 3a, both in kilobits per second (kbps). As long as the network is not fully loaded, which happens around 3.5 kbps, the carried load rises linearly. It shows a slightly higher carried load for basic CoAP until 5 kbps. From there on, the curves diverge showing the best performance for CoCoA, followed by CoCoA-S with basic CoAP performing worst.

The dropped MAC layer packets are shown in Figure 3b as a function of the offered load in kbps. MAC layer packets are dropped when the buffers overflow, i.e., when more packets are created than the network can handle. This is a symptom of congestion. Until the offered load reaches 3 kbps only few packets are dropped. Subsequently, the dropped packets rise rapidly. CoCoA shows the best performance, meaning the least amount of dropped packets. While CoCoA-S performs slightly inferior to CoCoA, basic CoAP show the worst performance.

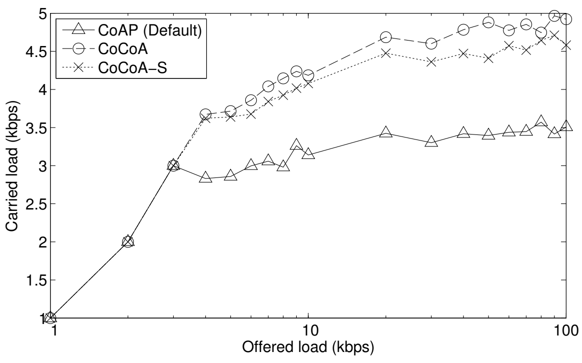


Figure 4: The throughput measured in the chain topology setup for NSTART=4, from [8].

Increasing the NSTART leads to more congestion in the network, as packets are not dropped when a second transaction with same endpoint is initiated. Figure 4 depicts the carried load as a function of the offered load for NSTART=4. The behavior for CoCoA and CoCoA-S is similar to the NSTART=1 graphs, while basic CoAP performs significantly worse once the network if fully loaded.

Overall, the graphs show that CoCoA provides a better congestion control than both other algorithms. The paper presents the evaluation for the topologies grid and dumbbell, which offer similar results. However, it should be considered that the new algorithm introduces overhead, such as tracking additional information for each destination.

3.3 Discussion

The paper presented the improved congestion control algorithm CoCoA for reliable messages in CoAP, which is currently an IETF draft. Furthermore, the authors suggested a variant of CoCoA that requires less overhead, called CoCoA-S. They evaluated the algorithms using a simulated environment that includes a network stack for IoT environments built from IETF standards. Overall, CoCoA outperformed the basic CoAP algorithm as well as CoCoA-S. However, CoCoA-S still performed better than basic CoAP.

I plan to use CoAP to adjust the network stack of the `CAF` for IoT environments. Actor systems are heavily dependent of message exchange and as a result congestion should be considered. Furthermore, CoAP is relatively new and there are not many accomplished libraries around that implement it. At least parts of it will be implement by us. Hence, drafts that improve the protocol and its performance are highly relevant.

In addition to the draft presented above, the CoRE Working Group has an active draft that implements block messages in CoAP [11]. They allow the transfer of data that does not fit into a single IEEE 802.15.4 frame. While IP does specify fragmentation, all fragments have to be retransmitted if a single fragment is lost. In contrast, CoAP block messages require only the retransmission of the lost packet.

4 Drop the Phone and Talk to the Physical World

The authors of the paper “Drop the Phone and Talk to the Physical World” by Sivieri at al. [12] show a similar motivation to our own, see Section 1. They use Erlang to approach these challenges and present the Erlang-based framework ELIoT. Erlang offers an actor-like concurrency model that fits the the distributed nature of the IoT. To my knowledge this is the first paper that connects the actor model and the IoT.

4.1 Erlang for the Internet of Things (ELIoT)

The authors chose Erlang because it contains many desirable features for the development of distributed systems. While the actor model is never mentioned, the languages contains processes which offer a concurrency model with the same characteristics. Processes do not share memory and can only communicate via location-independent, asynchronous message passing, thus masking distribution. Furthermore, the language includes a function core which includes pattern-matching and dynamic typing. The pattern-matching does even work for bit-streams and not only types, a feature designed for embedded systems as it allows a high-level handling of network packets. Finally, code can be hot-swapped, allowing easy updates on remote nodes.

However, Erlang is not suited for the IoT as it is. The communication between processes relies on reliable point-to-point communication based on TCP/IP. These are not always available or desirable in constrained environments. In addition, the runtime of Erlang is extensive and includes functionality that is not needed in IoT scenarios. Since memory is a constrained resource a large library can hinder deployment.

The ELIoT framework consists of three parts: 1) a library that provides functionality for decentralized networks, 2) a custom interpreter enhanced for constrained devices and 3) a simulator that provides partially or fully simulated environments.
Instead of relying on a full TCP/IP stack, the new library provides functions to communicate more power efficiently with remote processes. To stress the difference in cost and functionality between communication between local and remote processes, new functions

Algorithm	TinyOS	Contiki	ELIoT
Opportunistic flooder	495	187	100
Trickle	219	194	61
CTP	2169	1470	303

Figure 5: Lines of code for three different algorithms in TinyOS, Contiki and ELIoT, from[12].

are provided instead of overloading existing ones. These new functions allow the best-effort communication to one-hop neighbors. Furthermore, instead of addressing process identifiers, a process can be registered under a specified name and later identified by it. Furthermore, the option to create—spawn—processes on remote nodes is provided, but in difference to Erlang does not return the identifier of the create process, as the registered names should be used for identification.
Over the years the standard interpreter of Erlang has gained a lot of new functionality and libraries. However, this leads to a big memory footprint. Since some functionality is not required in embedded systems, such a support for Corba, ELIoT provides a slimed down interpreter that focuses on libraries and functionality for these environments. Their current work requires around 5 MB for a basic example.
Testing software for distributed embedded systems is a critical task. Deploying software to a large number of nodes in a testbed is a time consuming task and does not provided visibility into the running nodes. ELIoT provides a custom simulator that allows the simulation of multiple virtual nodes, each running unmodified ELIoT code. The simulator uses traces of wireless communication to model the communication between nodes. At runtime, the Erlang shell can be used to interact with the system, e.g., by injecting new messages. Finally, the virtual nodes can be deployed alongside physical ones to achieve a mixed setup. This allows moving from a fully simulated setup to a fully physical setup progressively. Some functionalities, such as sensors, are hard to simulate and easier to validate on real nodes. In a mixed setup this functionality can be provided by real hardware while the simulated nodes allow to retain a look into the system.

4.2 Evaluation

The authors implemented three different distributed applications that represent typical IoT scenarios. The first one is an opportunistic flooding protocol, the second one is the Tickle protocol used to distribute data in the network and last one is the Collection Tree Protocol (CTP) which transports data in the network to the closest source. Evaluating the advantage of abstraction is not an easy task. To measure the programming effort the authors counted the uncommented lines of code. Figure 5 shows the comparison for all three protocols. The more code the implementations for TinyOS and Contiki require, the greater is the advantage gained through ELIoT.

4.3 Discussion

The paper presented an Erlang based framework called ELIoT, designed to develop applications for the IoT. It offers a new library with IoT specific functionality, a lightweight interpreter striped of unsuitable libraries as well as a simulator that allows mixed as well as fully-simulated testbeds. Finally the authors examined programming effort by comparing lines of code for different embedded operating systems. ELIoT has the most compact code, which improves code readability and maintainability.

The paper addresses a similar problem to my work: raising the abstraction level for the development of embedded applications. As such, it is interesting to see their approach and focus for this task. In contrast to `CAF`, ELIoT relies on a interpreted language while `CAF` is build with a native language. Relying on a interpreted languages brings some features, that we are harder—or at the moment not possible—for us to achieve, such a code hot-swapping. On the other hand, an interpreted languages relies on an additional layer to run, which induces some overhead.

However, the paper leaves some questions open. Although the authors mention that they use an adjusted network stack, they do not explain how the stack is implemented and what protocols they use. Furthermore, the actor-like concurrency model of Erlang allows for network transparent communication, but is broken by ELIoT in introducing specific functions to communicate with remote processes. While this is motivated by the difference in cost it contradicts the abstraction model.

Finally, the code has not been published yet. As the authors are still active, I hope to see more of their work in the future.

5 Conclusion

This work presented three papers from the area of loose coupling and the Internet of Things. The first paper identified twelve facets that define loosely coupling, providing a foundation for further discussions. It showed that most system are not either loosely or tightly coupled, but exhibit different degrees of coupling for the facets.

Network congestion can have a heavy impact on the performance of IoT applications due to their highly distributed workflow. The second paper evaluated the congestion control in CoAP. In addition to the enhanced congestion control algorithm CoCoA developed by the IETF, the authors suggested a slimed down version called CoCoA-S. To compare their performance to the basic CoAP congestion control, they simulated different topologies and monitored throughput and packet loss. While CoCoA and CoCoA-S show better performance than the basic algorithm, it is left for future work to evaluate the overhead they induce.

The last paper has a similar motivation to ours, a better abstraction over distributed, concurrent systems for the IoT. It presents an Erlang-based framework for developing IoT applications, that includes a library with IoT specific functionality, a slimed down interpreter and a simulator. Erlang was chosen for its actor-like concurrency model and functional core. Although the paper presents the basic concepts and workings of ELIoT, it does not provide implementation details such as the composition of their network stack.

We are working to adjust the network stack of `C++ Actor Framework` (`CAF`) to the meet the challenges of the IoT. Out new stack is similar to the one used for the evaluation of the congestion control algorithms. It consists of IEEE 802.15.4, 6LoWPAN, UDP, DTLS for encryption and CoAP. We implemented a proof-on-concept using Ethernet, UDP and CoAP for a first flow analysis. It showed a reduced bit rate and less packages compared to the TCP implementation in `CAF`. Once the stack is fully implemented we can evaluation how it handles packet loss and small frame sizes. Further enhancement such as CoCoA or CoAP block messages have to be consider for the implementation.

Finally, we are working to run `CAF` on RIOT [3], an operating system for the IoT.

[3]http://www.riot-os.org

References

[1] Atzori, Luigi and Iera, Antonio and Morabito, Giacomo, “The Internet of Things: A Survey,” *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[2] 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.

[3] 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.

[4] Pautasso, Cesare and Wilde, Erik, “Why is the Web Loosely Coupled?: A Multi-faceted Metric for Service Design,” in *Proceedings of the 18th International Conference on World Wide Web*, ser. WWW ’09. New York, NY, USA: ACM, 2009, pp. 911–920. [Online]. Available: http://doi.acm.org/10.1145/1526709.1526832

[5] T. Berners-Lee, R. T. Fielding, and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” IETF, RFC 3986, January 2005.

[6] D. Chappell, *Enterprise Service Bus: Theory in Practice*, ser. Theory in practice. O’Reilly Media, 2004.

[7] Z. Shelby, K. Hartke, and C. Bormann, “Constrained Application Protocol (CoAP),” IETF, Internet-Draft – work in progress 18, June 2013.

[8] 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.

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

[10] V. Paxson, M. Allman, J. Chu, and M. Sargent, “Computing TCP’s Retransmission Timer,” IETF, RFC 6298, June 2011.

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

[12] 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.