2015 IEEE 18th International Symposium on Real-Time Distributed Computing

# Real-Time Communication for the Internet of Things using jCoAP

Björn Konieczek, Michael Rethfeldt, Frank Golatowski and Dirk Timmermann
Institute of Applied Microelectronics and Computer Engineering
University of Rostock
Rostock, Germany
Email: bjoern.konieczek@uni-rostock.de

***Abstract*—The term Internet of Things (IoT) describes a scenario where embedded systems are integrated into everyday objects, turning them into smart devices to assist the user in his everyday life. Each of these smart objects only offers a very limited amount of computational power since it is only specialized in a limited set of tasks. In order to achieve complex goals, the devices have to interact with each other. Therefore, they do not only need to be interconnected either by wire or through wireless technology but also need a set of common protocols to enable vendor-independent communication. In the past years, various protocols pursuing this objective have emerged. One of the most promising approaches is the Constrained Application Protocol (CoAP) because it offers high interoperability and very low communication overhead at the same time. Typical IoT applications include the observation and manipulation of their environment through sensors and actuators. Since the physical world is continuous in time and does not wait for calculations to finish, it is essential that the execution times of the applications stay within certain boundaries. These timing constraints are referred to as real-time requirements. However, current protocol implementations do not consider real-time requirements for IoT applications. In this paper, we introduce the jCoAP communication stack as a lightweight Java implementation of CoAP. We give a brief introduction to real-time communication and CoAP and provide insight in the design concept of jCoAP and the offered functionalities. Furthermore, a performance evaluation is done in order to point out the suitability of the jCoAP framework for real-time IoT applications.**

## I. Introduction

In the past few years, the idea of the Internet of Things (IoT) has become more tangible. Every day new technologies emerge that enable devices to communicate with each other in a vendor-independent manner. In order to maximize the benefit gained from interconnecting devices, it is desirable to integrate these devices into the already existing network infrastructure [1]. Since the Internet Protocol (IP) is widely used in this infrastructure, it seams preferable to use communication protocols that rely on IP. One well known IP-based approach in device communication is the Devices Profile for Web Services (DPWS) [2]. It offers high interoperability through Web Service (WS*) technologies and high integrability because it relies solely on protocols that are well-established on the internet. However, due to the usage of heavyweight protocols like SOAP and HTTP, the communication overhead is rather high [3]. To avoid this dilemma, the CoRE working group of the IETF specified the Constrained Application Protocol (CoAP). CoAP is a RESTful web transfer protocol that is ideally suited for the use in resource-constrained environments [4]. It offers high interoperability, low header overhead and a stateless HTTP mapping which enables easy integration. Still, for many IoT applications interoperability and integrability alone are not sufficient. These applications often involve the observation and manipulation of the physical world through sensors and actuators. Yet, the physical world is continuous in time and does not stop to wait on a calculation to finish. In order to avoid danger for life or property, the applications must satisfy certain timing constraints (deadlines). Applications that underlie these requirements are referred to as real-time applications and can be divided into three categories. In hard real-time applications any deadline miss will lead to a system failure. In firm real-time applications on the other hand, the value of information that is delivered after the deadline is zero, whereas in soft real-time applications the sporadic violation of deadlines is tolerable and the value of the delivered information degrades with the time it arrives after the deadline. Due to the distributed nature of IoT applications, it is not sufficient for every device to finish their part of the application within the deadline boundaries. In fact, the impact of the communication between the devices on the overall timing behavior is rather high. Hence, it is necessary to enable real-time behavior on the communication layer. In this paper we introduce the jCoAP communication stack and compare its timing performance with the well-known Californium CoAP implementation [5]. Furthermore, we will evaluate the influence of the used operating system and Java Virtual Machine (JVM) on the predictability of the latencies of transactions between interconnected nodes.

The main contributions of this paper are:

- Introduction and description of the design concept of jCoAP.
- Evaluation of the performance in a real world testbed.
- Revelation of measures that need to be taken to enable real-time communciation with jCoAP.

The remainder of this paper is organized as follows. In Section two we will give a short overview of the related work in this area. Section three gives a brief introduction to CoAP and its core features. In Section four, the design concept of the jCoAP framework will be explained. Section five describes

1555-0885/15 $31.00 © 2015 IEEE
DOI 10.1109/ISORC.2015.35



IEEE computer society

the experimental setup for the performance evaluation that was done. Section six draws conclusions from the experimental results and provides an outlook on future work.

## II. Related Work

### A. Device Communication

In the area of device interconnection, many IP-based approaches can be found. The two mainly discussed design principles are Service Oriented Architectures based on DPWS and RESTful communication. In [6], [7] it is shown that DPWS offers a high degree of interoperability. However, due to the utilization of XML based SOAP and HTTP messages it needs a considerable amount of computational power on the device side and causes a lot of communication overhead. It has been shown in [8] and [9] that DPWS can be optimized to better fit the needs of resource-constrained environments by applying compression techniques to the SOAP messages and programming optimizatons to the protocol implementation. These optimizations mainly involve the reduction of memory footprint. The smaller memory footprint is bought with a loss of features and hard to develop services though. Still, DPWS relies on HTTP which has a high message overhead and is based on TCP. The use of TCP involves a handshake procedure and implies the automatic retransmission of lost packets. This behavior is undesireable as it causes unnecessary network load and unpredictable timing behavior.

CoAP can be used to overcome this trade-off. The first specification of CoAP was released in 2010 by the CoRE working group as an internet draft [4]. Since then, this draft evolved to an approved standard and many different CoAP implementations have emerged. One of the most important ones is the Californium stack, which was developed by Kovatsch et. al. [5]. It is written in Java and known as the most fully-featured implementation. In 2013, Mathias Kovatsch suggested to include Californium into the Eclipse Project as CoAP reference implementation. Besides Californium, Kovatsch et. al. built two other implementations of CoAP that are widely used. The first one to mention is the Erbium Stack which is a C implementation [10]. Erbium is the built-in CoAP stack of the Contiki Embedded Operating System. It is very suitable for heavily resource-constrained devices such as TelosB mote and easy to use. However, it is part of the Contiki OS and therefore only available for a very limited number of platforms. The second one is a JavaScript based browser plug-in for Firefox called Copper [11]. It enables the user to visualize and manipulate CoAP resources on a remote server through the Web Browser. Since it can only be used as a client, it is widely used for testing purposes. Another important C implementation is the libcoap stack by Bergmann et. al., which is used as part of the TinyOS Embedded Operating System [12]. There are various other CoAP implementations of small relevance due to their lack of updates or their poor usability. However, none of the aforementioned CoAP implementations have been evaluated regarding their suitability for real-time applications.

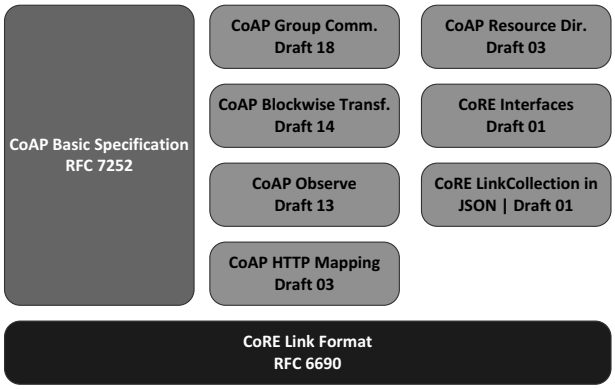


Fig. 1. Development status of CoAP

### B. Real-Time Communication

For distributed real-time applications it is crucial that the communication protocols on all layers show deterministic behavior. In the past, a variety of field bus systems has been used to achieve this behavior. These systems lag some abilities that are needed in today's IoT applications though. Firstly, the number of participants in the network is drastically limited. Secondly, these field bus systems are usually based on proprietary interfaces and hence are hard to integrate into current network infrastructures. This leads to the conclusion that field bus systems are not suitable for IoT applications. Mainly driven by the limitation in the number of devices, several real-time Ethernet solutions have emerged in the industrial area, also referred to as Industrial Ethernet (IE) [13]. This allows the easy integration of devices horizontally (with other devices) and vertically (with a company's remaining network infrastructure). Thus, it becomes possible to optimize controlling and monitoring of industrial plants. [14] gives an overview of the currently available IE solutions. However, most of these solutions require either specialized hardware or a specialized process data protocol on top of standard Ethernet. This leads to higher acquisition costs and degrades the integrability. To avoid this, Skodzik et. al. developed the HaRTKad framework [15]. HaRTKad is a peer to peer based approach that achieves hard real-time communication over standard GBit Ethernet through a time-slot-based medium access strategy. However, it still needs protocols on top to process the data.

## III. The Constrained Application Protocol

CoAP is a specialized web transfer protocol, designed as a lightweight RESTful communication protocol for heavily resource-constrained devices; thus enabling high interoperability between devices and increasing the ease of integration of these devices into already existing network infrastructures.

CoAP consists of the base specification and several sub-specifications [4]. The base specification describes the basic flow of the communication process, including the packet and

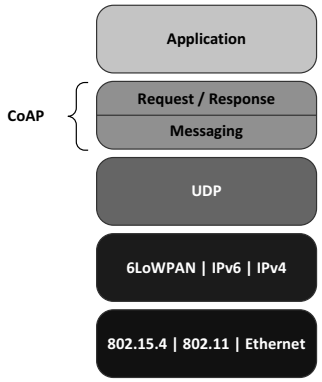


Fig. 2. Basic CoAP stack

header structure as well as header options, whereas the sub-specifications extend the base specification with additional but not mandatory functionalities that can be used with CoAP. Amongst others, these extensions include a publish/subscribe mechanism, group communication and the transfer of big amounts of data with a simple stop-and-wait algorithm called Blockwise Transfer. As it can be seen in Fig. 1, only the base specification and the CoRE Link Format, which is used to describe resources in CoAP, are confirmed standards until now. The sub-specifications are still under development.

CoAP follows the client-server-principle, where a client sends a request to a server in order to invoke a service on, send data to, or retrieve data from the server. The CoAP stack is depicted in Fig. 2. CoAP is based on UDP to avoid the TCP handshake and automatic retransmission of packets since this behavior is cost-intensive and not necessarily desirable in resource-constrained environments. CoAP itself consists of two sub-layers. First, there is a messaging layer. It defines four types of messages: confirmable (CON), non-confirmable (NON), acknowledgement (ACK) and reset (RST). Since CoAP utilizes UDP, there is no reliable packet transmission per se. To overcome this disadvantage, CoAP offers optional reliability through CON-messages. If a CON-message is received, the receiving node must respond with an ACK-message in order to confirm a successful transmission. If the sender of a CON-message does not receive an ACK within a certain timeout, it will resend the message. The mapping of ACK-messages to the corresponding CON-message is done via a message ID. However, in the real-time domain the retransmission of data is undesired because the value of the information is zero if it is not delivered in time. To enable a non-reliable communication, NON-messages can be used, as they do not need confirmation. RST-messages are used as a response to requests that can not be interpreted. The request/response layer is placed on top of the message layer. It enables the request/response pattern between the network nodes. In this layer, the four basic methods GET, PUT, POST and DELETE are performed. In order to match responses to the corresponding requests, a message token can be used. Fig. 3 shows the basic CoAP header.

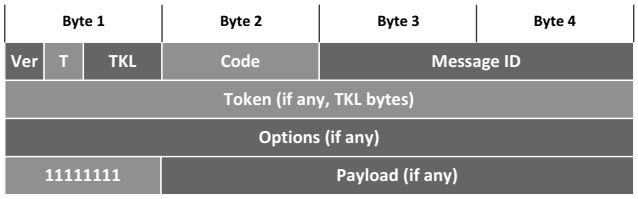


Fig. 3. Structure of the CoAP header

The first byte contains the protocol version (V), the message type (CON, NON, ACK or RST) and the length of the message token (TKL, can be zero). The second byte contains a message code that determines whether the message is a request (GET, PUT, POST or DELETE) or a response, in which case it contains a response code that depends on the corresponding request. Byte three and four contain the message ID. The following bytes contain the message token of the length specified in the TKL field and the options. Both fields are not mandatory. An option consists of an option number, the option length and the option value. The option number for every option type is fixed and defined in [4] or the sub-specification that introduces the option type to CoAP. Every option can occur multiple times. The options are ordered by their option number. In case a message contains payload, a 1-byte payload marker must succeed the options. The header is binary encoded to minimize the header size and the parsing effort. The CoAP header is at least 4 Bytes long. However, it is most likely to be larger, as numerous options can be used to trigger special behavior. Options are also used to indicate the URI of the target resource on the server. In order to enable an efficient device communication, several special features, e.g. service discovery, need to be provided. Subsequently, it will be described how CoAP realizes the three most important features.

### A. Resource Discovery

A server in the network can be discovered either through learning an URI that references a resource on the server or through the ”All CoAP Nodes” multicast address. To discover the resources that are provided by a server, the client sends a GET request for the ”/.well-known/core” resource. The server must respond to this request with a list of all provided resources including their URI and the supported content types (optional). This request can either be send to one particular server or to all nodes in the network. Clients that receive such request will either respond with a RST message or just drop the request message.

### B. CoAP Observe

A typical use case for IoT applications is to initiate actions depending on the environmental conditions. In these cases, the environment has to be observed through sensors. Actors then ask frequently for the current value. However, this behavior causes unnecessary network load, as the clients request the new sensor value even though it has not changed. Therefore,

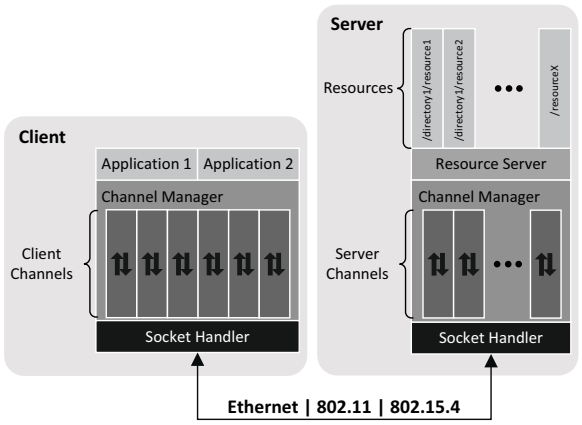


Fig. 4. jCoAP client and server

CoAP offers a publish/subscribe mechanism called ”CoAP Observe” [16]. Here, a client sets the observe option in its first GET request for a resource. When a server recognizes an observe option in a request, it adds the client to the list of observers for this particular resource. Every time the resource changes its value, the server notifies every client in the list of observers with the new value. To indicate that the new message received by the client is a notification, the server also uses the observe option. The option value is a 24 Bit long sequence number for reordering purposes.

### C. Blockwise Transfer

Since CoAP omits the use of TCP, it needs other mechanisms to transmit large amounts of data. Unlike TCP, that uses a streaming approach, CoAP offers a simple stop-and-wait algorithm called ”Blockwise Transfer” [17]. On a request, the server first divides the data into smaller blocks. Then, only the first block is sent in the response and the block option is set to indicate that more blocks will follow. The client buffers the data, reads the block option in the response and then sends a request for the next data block to the server, also using the block option. This is repeated until the whole data is transmitted. CoAP provides two types of block options, Block1 and Block2. The usage of the block options depends on whether data is going to be retrieved from or sent to the server. If a device wants to store data on a server through a PUT request, it needs to use the Block1 option; whereas the Block2 option is used if data is retrieved from a server through a GET request. Both block options contain the block number, the block size and a flag that indicates whether there are more blocks to follow. The block size can vary between 16 Bytes and 1024 Bytes per block.

## IV. The Structure of jCoAP

jCoAP is designed as a very lightweight Java CoAP framework. It can be used to implement both, a client and a server. The basic structure is depicted in Fig. 4. A client consists of one or more applications. Each application can initiate a connection to a remote server. For every connection, a client channel is created to map the incoming packets to the corresponding client application. The channels are identified through a channel key (CK). This key is created as a combination of the server's IP address and port through Formula 1.

$$CK = P * (P + Hash_{serverIP}) + Port_{server} \quad (1)$$

Here, $P$ is an arbitrary factor and $Hash_{serverIP}$ the hash value of the server's IP address. The hash value is calculated through the built-in function of the standard Java InetAddress class. When a client sends a request to a server, the packet is forwarded through the channel to the SocketHandler. The SocketHandler manages open ports and all outgoing and incoming packets. All outgoing packets are stored in a send buffer and are sent one after another through the specified port. When a server receives a message, it is forwarded to the right server channel. The channel is selected through the channel key that can be calculated from the sender's IP address and port. If no channel with the specified key exists, the server creates it. The channel forwards the message to the ResourceServer component which inherits the main functional part of the server. It parses the received request for the operation type and header options. Afterwards, it selects the target resource and checks whether the desired operation is allowed for this resource. A resource can be any application, file or value. After the operation on the resource has finished, the resource server creates the proper response and sends it back to the client. Again, the client receives the message in the SocketHandler. The SocketHandler checks whether the message is a valid response and selects the corresponding channel. In contrast to the server, the client drops the packet if no channel can be found, for a server cannot initiate a connection. To notify the client application of the newly received response, the client invokes the onResponse() method of the application. Afterwards the application can process the received data. JCoAP not only offers standard CoAP functionality according to [4], it also implements parts of the extended functionalities like Blockwise Transfer which will be subsequently described. However, some substandards, e.g. Group Communication, have not been implemented yet, for this is still work in progress.

### A. CoAP Observe

Alongside of the basic CoAP functionality, jCoAP also provides the CoAP Observe feature. The ResourceServer contains a list of all resources offered by the server. Each resource maintains a list of server channels. These server channels correspond to the clients that want to observe this resource. When a server receives a GET request that contains the observe option, it calls the addObserver() method on the resource specified in the request and passes the channel as argument. The channel will then be added to the resources list of observers. Every time a value change occurs on a resource, its changed() method is called. In this method a notification message for the value change is generated. Afterwards, it will

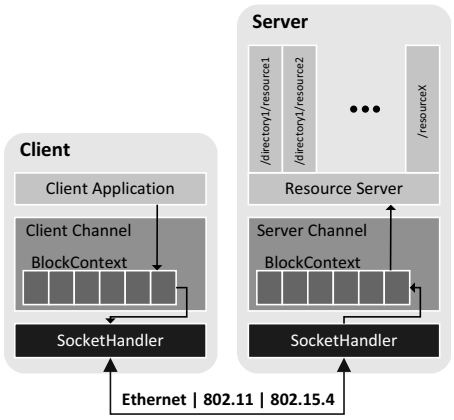


Fig. 5. Blockwise Transfer in jCoAP

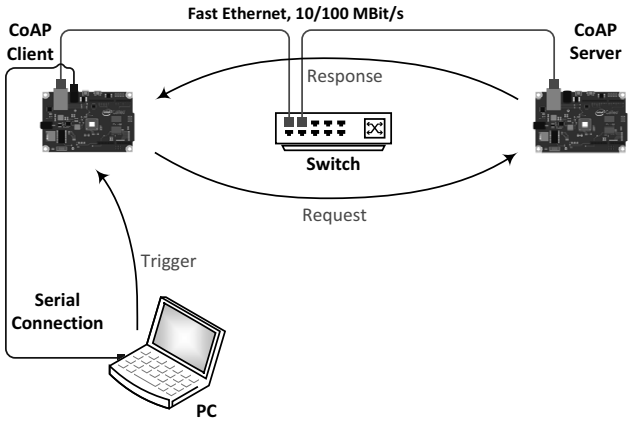


Fig. 6. Test setup for the performance evaluation.

iterate through the list of observers and send the notification through every channel in the list. To cancel a subscription, the client will either send a RST message as response to a notification or a GET request that again contains an observe option.

### *B. Blockwise Transfer*

The jCoAP framework implements the Blockwise Transfer feature in a very simple way.

Fig. 5 shows the work flow of a client and a server in a blockwise transaction. In the following, a blockwise GET and PUT request will be explained briefly. If a client wants to retrieve data from a server in a blockwise manner, it sets the Block2 option in the message header of the GET request. On the server side, every message is checked for the presence of any block option. In this case, a Block2 option is recognized. The channel then checks whether this message belongs to an already ongoing transaction. If the received request is the first one in this transaction, the message is forwarded to the ResourceServer. Here, the requested data is retrieved from the resource. This data is then used to create a BlockContext in the channel which is mainly a data buffer. On subsequent get requests the channel will automatically generate the matching response and retrieve the requested data block from the buffer in the BlockContext. When a client wants to send data to the server, it will add the Block1 option to its PUT request. When a Block1 option is recognized by the server channel, it will again check whether this request belongs to an ongoing transaction. If this is not the case, the channel creates a BlockContext and buffers the payload of the received request. Afterwards, it will generate a matching response. The payload of all subsequent PUT requests from this client will be buffered in the created BlockContext. When the transaction is done, the channel creates a single message object from the buffered data and forwards it to the ResourceServer. The ResourceServer will then invoke the PUT operation with the complete data on the specified resource.

## V. Performance Analysis

To analyze the performance of the presented jCoAP framework, an example client-server application was created. Fig. 6 shows the test setup.

Both, the client and the server, are deployed on an Intel Galileo Board of the first generation. The Intel Galileo is equipped with the QuarkX 1000 SoC which is based on the x86 architecture and has a clock frequency of 400MHz. The board additionally supplies 256 MB RAM, a 10/100 MBit/s Fast Ethernet adapter and a host USB interface. As operating system, a customized Embedded Linux created with the Yocto Build Tools has been used. The boards are interconnected through 10/100 MBit/s Fast Ethernet and a switch. The server offers an echo resource which contains a byte array as value. The PC triggers the start of the test. The client then generates a 16 Byte random payload and sends a PUT request as CON message to the server in order to assign a value to the echo resource. When the client receives the corresponding ACK message, it starts to retrieve the value of the echo resource with a GET request 100 times in a loop. After each loop, a new random payload which is 16 Bytes longer is created and sent to the server. Subsequently, a new loop of GET requests is initiated. This will be repeated until a 160 Byte payload is reached. For every transaction, the time is measured using the System.nanoTime() method. This means, only the wall-clock time which can vary depending on the system load caused by other applications is measured. In order to ensure comparable results, the tests were executed under similar conditions on the system load. A transaction begins when the send method is called by the client and ends when the retrieved data is accessible to the client application. Subsequently, we compare the performance of jCoAP with the widely used Californium framework. Afterwards, we describe the evolution to deterministic timing behavior step by step. Additionally, we show that the use of blockwise transfer is not necessarily preferable for larger amounts of data.

For our first experiment, we have realized our test application with both CoAP implementations, jCoAP and

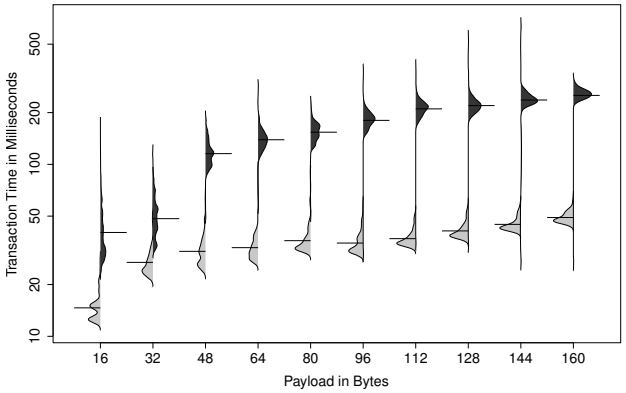


Fig. 7. Transaction Time (logarithmic scale) for CoAP test application implemented with jCoAP and Californium. The light gray density curve on the left of each payload size belongs to jCoAP. The dark grey density curve on the right belongs to Californium.

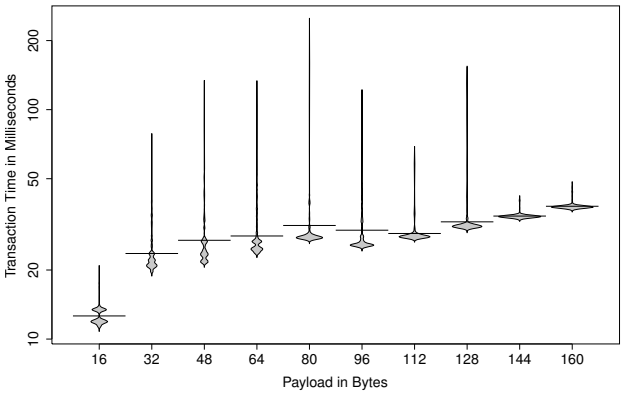


Fig. 8. Transaction Time (logarithmic scale) for jCoAP test application using Oracle JVM and preemptive Linux.

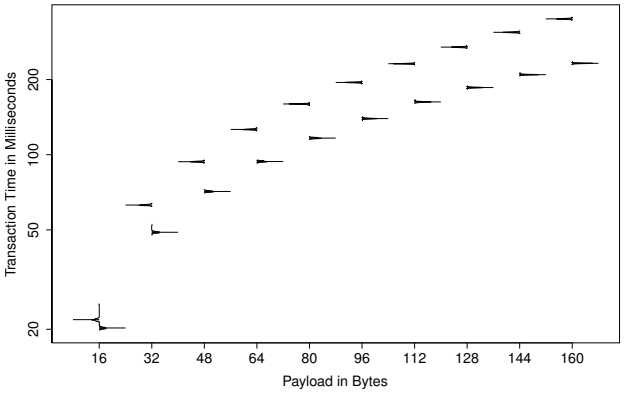


Fig. 9. Transaction Time (logarithmic scale) for jCoAP test application run with JamaicaVM and JamaicaBuilder on preemptive Linux. The light gray density curve on the left of each payload size belongs to the JamaicaVM. The dark gray density curve on the right belongs to the JamaicaBuilder.

Californium. We have used a standard Linux kernel version 3.8 and the Oracle Java HotSpot Client Virtual Machine version 25.11-b03 with its default configuration to run the test application. Fig. 7 shows the transaction time in milliseconds for a block size of 16 Byte per message. The black lines mark the median transaction time while the colored areas indicate the density distribution of the measured transaction times. The dark gray density curves on the right side of each payload size belong to Californium whereas the light gray density curves on the left belong to the jCoAP version of the test application. It can be seen that the performance with the jCoAP framework is much better than with Californium. However, the density distribution reveals that in both cases the measured transaction times are subject to large fluctuations. This is due to the indeterministic nature of Oracle's JVM and the standard Linux kernel in the sense of the timing behavior. Furthermore, it can be seen that the timing fluctuations are higher for smaller payloads. The larger the payload, the more packets need to be exchanged for a single transaction. Hence, the fluctuations for the single packets are more likely to cancel each other out. The reasons for the observed indeterminism mainly originate from the Garbage Collection of the JVM and concurrent allocation attempts for system resources by other processes [18].

### *A. Deterministic Timing Behavior*

In this section we describe how we achieved deterministic timing behavior with jCoAP. Our first step towards this goal was to use a fully preemptable Linux kernel as real-time operating system. In this way, we are able to assign higher priorities to the Java Virtual Machine (JVM). The JVM was run with a priority value of -77, so that it was able to even interrupt kernel tasks [19].

Fig. 8 shows the median transaction time and the density of the measured time values achieved with this approach. It can be seen that outliers with very long transaction times appear, while most of the measured transaction times are below average. We believe that these strong outliers mainly originate from the Java Garbage Collection (GC). Although the JVM is highly prioritized and thereby has privileged access to system resources, it is not predictable when the GC will start or how long it will take. To verify this thesis, we used a real-time JVM with deterministic GC. The chosen real-time JVM was Aicas' JamaicaVM because it fully supports the Real-Time Specification for Java (RTSJ) [20]. The JamaicaVM offers two possible ways to run Java applications as real-time tasks. Firstly, the Java application can be directly run with the JamaicaVM. The second way is to use the JamaicaBuilder to cross compile a standalone binary directly from the Java application's source files. The results achieved with both approaches are shown in Fig. 9.

The density curve and average bar on the left of every payload size belong to the version directly run with the JamaicaVM whereas the right density curves belong to the cross compiled version. It can be seen that the median transaction time of both approaches increases linearly with the number of sent messages per transaction and is much larger than the median transaction time that could be achieved with

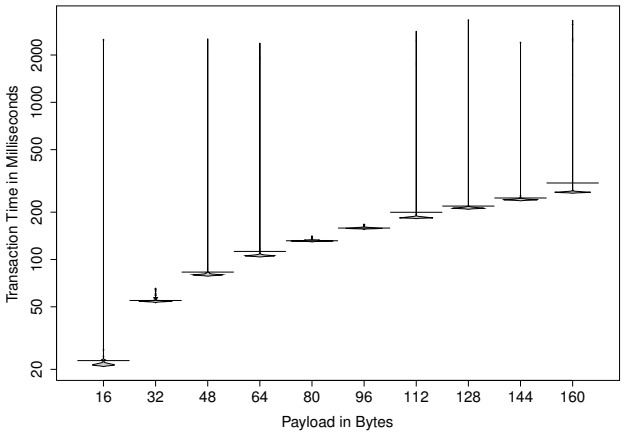


Fig. 10. Transaction Time (logarithmic scale) over WLAN for jCoAP test application run with JamaicaVM on preemptive Linux.

Oracle's JVM and a preemptive Linux kernel. Furthermore, it can be seen that the cross compiled version of the the test application is slightly faster. This is due to optimizations that are performed during the compilation process. The resulting binary is specifically build for a particular target architecture and needs to be recompiled for different devices though. Yet, the transaction time for both approaches is still suitable for smart environments in building or home automation [21]. More importantly, the density of the measured time values reveals that not only the outliers caused by the undeterministic GC can be eliminated, but also the fluctuations of the time values are minimized. This leads to a deterministic timing behavior of the transactions between two CoAP devices.

### B. Wireless Communication

For the performance evaluation of jCoAP in wireless environments, we equipped both Galileo Boards with a USB WLAN adapter through their host USB interface. The WLAN adapter utilizes a Ralink RT 2870 USB chip set. The client and the server board were interconnected via a wireless 802.11g ad-hoc network. Fig. 10 shows the obtained results.

It can be seen that the median transaction time in wireless and wired networks (compare Fig. 9) are much alike. However, it also shows that in wireless networks few outliers with very high transaction times appear. These are caused by the high susceptibility of wireless communication to interferences. Interferences can either be caused by external radio sources or other participants of the network that try to access the communication medium at the same time. In order to avoid concurrent access attempts on the medium, 802.11 uses a Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) technique. Here, the sender of a message checks whether the communication medium is already in use and only sends the message if this is not the case. When the medium is in use, the sending process is postponed for a random period of time. After this period is over, the medium is checked again for a concurrent access. Every time a concurrent medium access is detected, the sending process is postponed for an exponentially increasing period of time. This leads to unpredictably long transaction times whenever interferences appear. To minimize the impact of this effect, a Time Division Multiple Access (TDMA) method can be applied. Hereby, a time slot is assigned to every network participant, in which it has exclusive access to the communication medium. In this way, the probability for concurrent medium access attempts of other network participants can be drastically reduced. However, external radio sources can still interfere with the communication. Since there are only few outliers, it can be stated that jCoAP can at least satisfy soft real-time requirements in wireless environments.

TABLE I
TRANSACTION TIME AND STANDARD DEVIATION FOR DIFFERENT EXPERIMENTAL SETUPS.

| | **Transaction Time (median)** | **Transaction Time (avg.)** | **Standard Deviation** |
|---|---|---|---|
| **Ethernet & Standard Linux Kernel** | | | |
| Oracle JVM | 14.691 ms | 14.885 ms | 2.937 ms |
| **Ethernet & Preemptive Linux Kernel** | | | |
| Oracle JVM | 12.154 ms | 12.584 ms | 0.972 ms |
| JamaicaVM | 21.818 ms | 21.856 ms | 0.142 ms |
| JamaicaBuilder | 20.223 ms | 20.238 ms | 0.202 ms |
| **WLAN & Preemptive Linux Kernel** | | | |
| JamaicaVM | 21.476 ms | 46.436 ms | 246.176 ms |

Table I gives an overview of the median and mean transaction time and the corresponding standard deviation for the evaluated jCoAP setups. It shows the trade-off between average speed and time predictability of the communication, whereby the combination of JamaicaBuilder with the preemptive Linux kernel achieves the best results.

### C. Blockwise vs. Non-Blockwise Transfer

In our previous experiments we used the blockwise transfer functionality of jCoAP to deliver the desired data. In this way, it is possible to interrupt the communication after every data block that is transferred. This comes in handy when TDMA techniques are applied so that every device has only a limited amount of time to send data. However, blockwise communication in CoAP causes additional communication overhead and increased transaction times because every data block has to be requested separately by the client. Therefore, it is important to choose the right block size to minimize the communication overhead while maintaining the ability to pause the communication at any given time. In order to evaluate this, we ran our test application with a block size of 1024 Bytes. This block size was chosen in regard to the maximum ethernet frame size of 1514 Bytes so that the CoAP packet would fit into a single Ethernet frame, even with the maximum payload. In this way, it is still possible to pause the communication at any given time. Fig. 11 shows the obtained results. It can be seen that the transaction time increases only slightly when the payload in a single packet is maximized. This is due to the increased time that is needed to process the incoming messages on the server and client side before the data is available to the application. However,

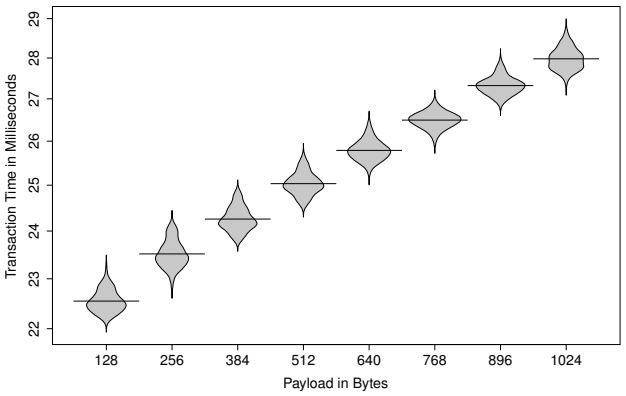


Fig. 11. Transaction Time for jCoAP test application run with JamaicaVM and increasing payload in a single packet on a preemptive Linux kernel.

as previous experiments have shown, the increase in the transaction time that is caused by the usage of multiple data blocks instead of a single larger packet is much higher. Yet, the transaction time would also increase if the size of a single packet exceeds the size of the maximum transfer unit (MTU) that depends on the used communication technology due to fragmentation. Fragmentation would also prohibit to pause the communication at any given time. Therefore, we conclude that the block size should always be chosen in regard to the communication technology, so that it is as large as possible but never exceeds the MTU.

## VI. Conclusion

In this paper, we have presented a lightweight Java implementation of the Constrained Application Protocol called jCoAP. We have shown that jCoAP enables CoAP-based communcation for embedded devices with comparably small latencies. Furthermore, we have shown that a slower but more predictable communication can be established even for wirelessly interconnected devices by using a real-time operating system and JVM. Additionally, suggestions on when and how the blockwise communication feature of CoAP should be used were made. Yet, in our test cases only two devices were interconnected, hence no simultaneous communication occurred. However, real operational scenarios include a multitude of devices. This will cause more concurrent medium access attempts. Thereby, the predictability of every single transaction will be reduced, especially in wireless environments. To minimize these effects, time division multiple access (TDMA) is a promising approach to be evaluated in future work. Therefore, a common time basis for all devices is needed. In our future work, we will evaluate different TDMA and time synchronization approaches to enable real-time communication with jCoAP for multiple devices.

## Acknowledgment

We like to thank Jens Schiebel and Marcel Borutta from Aicas GmbH for their assistance in the deployment of the Jamaica real-time Java VM on the Intel Galileo Board. We also like to thank Christian Lerche and Nico Laum for their spadework with the jCoAP implementation. This work was partially financed by the German Research Foundation (DFG) within the graduate school Multimodal Smart Appliance Ensembles for Mobile Applications (MuSAMA, GRK 1424).

## References

[1] L. Atzoria, A. Ierab, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54 Issue 15, p. 27872805, October 2010.

[2] T. Nixon, A. Regnier, D. Driscoll, and A. Mensch, *Devices Profile for Web Services Version 1.1*, Online, OASIS Std.

[3] Z. Shelby, "Embedded web services," *IEEE Wireless Communications*, pp. 52–57, December 2010.

[4] Z. Shelby, K. Hartke, and C. Bormann, *RFC 7252: The Constrained Application Protocol*, online, IETF Std.

[5] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: Scalable cloud services for the internet of things with coap," in *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*, 2014.

[6] G. Cndido, F. Jammes, J. Barata, and A. W. Colombo, "Generic management services for dpws-enabled devices," in *35th Annual IEEE Conference of Industrial Electronics (IECON '09)*, Posto, Portugal, November 2009, pp. 3931–3936.

[7] S. Iarovyi, J. Garcia, and J. L. M. Lastra, "An approach for osgi and dpws interoperability: Bridging enterprise application with shop-floor," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, Bochum, Germany, July 2013, pp. 200–205.

[8] R. Kyusakov, P. P. Pereira, J. Eliasson, and J. Delsing, "Exip: A framework for embedded web development," *ACM Transactions on the Web (TWEB)*, vol. Volume 8 Issue 4, no. 23, October 2014.

[9] C. Lerche, N. Laum, G. Moritz, Z. Elmar, F. Golatowski, and D. Timmermann, "Implementing powerful web services for highly resource-constrained devices," in *Pervasive Computing and Communication Workshops (PerCom Workshops), IEEE International Conference*, Seattle, WA, USA, March 2011.

[10] M. Kovatsch, "A low-power coap for contiki," in *8th IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS)*, Valencia, Italy, October 2011, pp. 855–860.

[11] ——, "Coap for the web of things: from tiny resource-constrained devices to the web browser," in *UbiComp '13 Adjunct, Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, Zurich, Switzerland, September 2013, pp. 1495–1504.

[12] O. Bergmann. libcoap: C-implementation of coap. [Online]. Available: http:// libcoap.sourceforge.net

[13] (Panel Building & System Integration) Ethernet adoption in process automation to double by 2016, 2013. [Online]. Available: http://www.pbsionthenet.net/article/58823/Ethernet-adoption-in-process-automation-to-double-by-2016.aspx

[14] P. Danielis, J. Skodzik, V. Altmann, E. B. Schweissguth, F. Golatowsk, D. Timmermann, and J. Schacht, "Survey on real-time communication via ethernet in industrial automation environments," in *19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2014)*, Barcelona, Spain, September 2014, pp. 1–8.

[15] J. Skodzik, P. Danielis, V. Altmann, and D. Timmermann, "Hartkad: A hard real-time kademlia approach," in *11th IEEE Consumer Communications & Networking Conference (CCNC)*, 2014, pp. 566–571.

[16] K. Hartke, "Observing resources in coap," *draft-ietf-core-observe-15*, 2014. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-core-observe/

[17] C. Bormann and Z. Shelby, "Blockwise transfers in coap," *draft-ietf-core-block-15*, 2014. [Online]. Available: https://tools.ietf.org/html/draft-ietf-core-block-15

[18] P. C. Dibble, *Real-Time Java Platform Programming*, J. H. Schwartz, Ed. Prentice Hall Professional, 2002.

[19] R. Love, *Linux kernel development*. Pearson Education, 2010.

[20] *JamaicaVM 6.3 User Manual - Java Technology for Critical Embedded Systems*, aicas GmbH, May 2014. [Online]. Available: https://www.aicas.com/cms/en/reference-material

[21] G. P. Fettweis, "The tactile internet: Applications and challenges," *Vehicular Technology Magazine, IEEE*, vol. 9, Issue: 1, pp. 64–70, March 2014.