Evaluation of CoAP Implementations for Live Streaming using CoAP-Observe

Arne Wall, Hannes Raddatz, Bala Vikram Reddy Gopu, Dirk Timmermann
University of Rostock
Institute of Applied Microelectronics and Computer Engineering
18051 Rostock, Germany, Tel.: +49 381 498-7271
Email: arne.wall@uni-rostock.de

Abstract—CoAP (Constrained Application Protocol) enables embedded devices to offer RESTful Web Services and exchange small binary message headers. Next to the transmission of sensor data and control information between smart home devices, there is an application field of streaming data using CoAP. There are CoAP implementations available in Java, that allow an execution on heterogeneous devices and operating systems due to the Java Virtual Machine and Java Runtime Environment. Because of the timing demands of live streaming applications in terms of low latency communication at application layer, we evaluate two different CoAP Java implementations. We compare the timing fluctuations of the packet processing with a CoAP C implementation. As a conclusion we identify the Java Garbage Collector to cause large fluctuations of the packet processing, which results in bad suitability for a live streaming scenario. Furthermore, we recommend the usage of native code applications because of better timing results.

I. INTRODUCTION

The Constrained Application Protocol (CoAP) [1] gained an increasing popularity in the field of Machine-to-Machine (M2M) communication of resource constrained devices. It enables devices to send application data with a small message overhead, due to the binary application header format and a lightweight congestion control of UDP packets over IPv4 and IPv6 networks. The application data contains, e.g., sensor data or control messages for actuators. Next to small data sizes, CoAP can be used for streaming application data at a high data rate. There are two principles to deliver streams, the CoAP block-wise transfer and the CoAP-Observe [2] mechanism. One concept using the block-wise transfer presented by [3] is evaluated with a ns-3 simulation regarding the suitability of the congestion control mechanism. The authors of [4] show a suitability of the CoAP congestion control mechanism to stream block-wise data over lossy networks using a test bed. In [5], another streaming approach based on CoAP-Observe is introduced. However, no practical evaluation on this concept and different CoAP implementations exists so far. In this paper we implement this streaming mechanism using three different CoAP implementations written in Java and C and evaluate them in terms of timing behaviour.

II. TECHNOLOGICAL BASIS

A. CoAP

CoAP is a protocol for M2M communication based on the server/client principle. The server offers a RESTful (Representational State Transfer) Web Service that can be invoked by CoAP clients. Every resource of the Web Service can be requested with the CRUD-operations GET, PUT, POST and DELETE, so that there is a compatibility to RESTful APIs that use HTTP. CoAP defines to set the application layer payload to 1024 bytes so that there is no IP packet fragmentation on the MAC layer, e.g., of Ethernet (1500 bytes MTU). There are two basic mechanisms enabling the transmission of high data rates using CoAP: One is the block-wise transfer that allows the server to send large data chunks packed into multiple CoAP messages that require one IP packet each. Another feature of CoAP is the asynchronous notification of a client through the CoAP-Observe mechanism. The client subscribes to resources and receives notifications about changes of, e.g., a temperature sensor. Next to small rates of sensor data it is possible to send application data at a higher rate like video streams.

B. Streaming using CoAP Block-wise Transfer

In [4] the authors present DASCo, a modification of the streaming protocol MPEG DASH [6]. To receive a video, a MPEG DASH client requests meta data of available video formats and qualities from the server. Next, the client will request video segments of a few seconds length via HTTP GET on a resource, which is extracted from the meta information. MPEG DASH allows a client to request a different video quality depending on the quality of the transmission channel. Usually Video on Demand services are requested to deliver low video quality segments first, because of the optimized user experience, which means a seamless start of the stream after requesting it via a Web GUI. DASCo substitutes HTTP through CoAP which entails the substitution of TCP through UDP. The authors show that while using a modified re-transmission timeout of the application-layer congestion control of CoAP a comparable or better performance can be achieved.

C. Streaming using CoAP-Observe

The principle of sending asynchronous notifications is exploited by the mechanism published by the patent [5]. The basic principle relies on the CoAP-Observe mechanism that enables clients to subscribe asynchronous events of a resource. In their proposed scheme the client observes a video resource which triggers the server to send the chunks of the video stream to the subscriber. We implemented this mechanism using different CoAP stacks. In this paper, we do an analysis of the timing behaviour of the CoAP Java implementations.
 Californium [7], jCoAP [8] and libcoap [9] (C implementation) on a test bed of different embedded devices.

III. Test Setup

In our test setup (Figure 1) we make use of two devices running a server and a client application. A client sends an observe request to a resource representing the live stream. The server starts to send chunks to the client immediately. The data source is implemented with a separate C application that periodically produces data of a fixed size. Via a pipe the test data is sent to the server application. In all test cases we implemented server and client with the same CoAP implementation (Californium, jCoAP and libcoap). To achieve significant experimental results we execute the CoAP implementations on different hardware platforms. Therefore, we make use of the Raspberry Pi 1 [10] as a single core system, the ZedBoard [11] as a dual core system and the Raspberry Pi 3 as a quad core system. Both devices, server and client, are connected with a wired 1 Gbit/s Ethernet switch, because we do not yet investigate the transmission via an unreliable channel. We evaluate the processing time of each CoAP implementation on different hardware. The JRE 1.8 build 171 was used for all Java applications on all devices to get comparable results. We take five time stamps for each message: Time \( t_1 \) indicates that data from standard input (coming from the test data source) is available. \( t_2 \) is the time point when the server has created the complete CoAP message right before sending it via the UDP socket. \( t_3 \) is taken after returning from the socket. On client side the time \( t_4 \) is taken when raw data is received by the UDP socket. After parsing the CoAP message and extracting the payload \( t_5 \) is measured.

IV. Evaluation

The first test case we created, describes a test data stream of a fixed data rate with a fixed chunk size. Therefore, we measured the creation of CoAP messages on the server and the parsing time of the client application with 10,000 messages each. Another experiment was used to estimate the latency and its fluctuation which is influenced by the garbage collector of the JVM.

A. Packet Processing at Fixed Data Rate and Chunk Size

First, we illustrate the duration to build a packet (Figure 2), to return from the socket (Figure 4) and to read a packet (Figure 5) from client side for jCoAP and Californium (Cf). We send test data at a fixed transmission interval of 100 ms and 10 ms on all platforms. In Figure 2 there is a comparison between the packet build time using jCoAP and Cf on three different platforms at a transmission cycle of 100 ms and 10 ms. In both cases and implementations, the performance of the Raspberry Pi 3 and the ZedBoard are comparable. The Raspberry Pi 1 has a much lower performance than the other platforms. This results in an increased packet build time on server side when an application with a higher throughput demand is executed. Building a packet with jCoAP within a 100 ms interval on the Pi 1 takes approximately 514 µs compared to 1655 µs within an 10 ms transmission interval. The same increased computational time can be observed when Cf is used to send packets. The build time increases from 763 µs at an 100 ms transmission interval to approximately 240 ms at an 10 ms interval. The large execution time of 240 ms is not displayed in Figure 2 due to better readability. Those nonlinear effects occur on all platforms at a certain threshold of the data rate to be sent.

Fig. 2: Build Time for each Packet created by the Server at 100 ms and 10 ms Interval

In Figure 3 we depict exemplarily the increasing packet build time on a Raspberry Pi 3 while executing the Cf implementation. There is a threshold of a certain data rate that leads to a maximum load of the machine, which can be measured in an increasing packet build time. After sending...
10,000 test messages every 1 ms, the build time continuously increases to 250 ms. We can observe that a higher rate results in a slightly decreased processing time. But only of the data rate is within the maximum threshold. Another observation is that the average execution time to build a packet is smaller if jCoAP is used compared to Cf.

Next to the applications, where the timings are determined by the underlying JVM, we investigated the C-implementation libcoap. Running native code is much faster than the variants with a JVM.

![Socket Time Graph](image)

**Fig. 4: Time to Return from Server Socket at 100 ms and 10 ms Interval**

In Figure 4 it can be seen that returning from the socket is quicker than building and parsing a packet. There is an internal thread in jCoAP and Cf that reads messages from a queue and sends them over the socket. Therefore, the socket time in Figure 4 is the theoretical limit for a data transmission interval.

![Packet Read Time Graph](image)

**Fig. 5: Time to Read a Packet on the Client at 100 ms and 10 ms Interval**

Figure 5 shows the time to read a packet on client side. Due to the overload situation at a 10 ms transmission interval there is no valid result for Pi 1. We can observe that Cf has a slower packet processing compared to jCoAP independent of the transmission rate. A statistical evaluation shows that all timings of Cf are slower than the timings of jCoAP with a confidence interval of larger than 99.9%. Comparing the creation and parsing times of a single packet the parsing procedure is significantly faster than the creation of a message.

![Packet Build Time Graph](image)

**Fig. 6: Build Time for each Packet created by the libcoap Server at 100 ms and 10 ms Interval**

In Figure 6, we can see that operating at a higher transmission rate results in a slightly lower packet build time. We assume that this effect is caused by a higher probability that the code, which is executed at a higher rate, is stored in the cache. This assumption needs further investigation. Furthermore, we can observe that the platforms Pi 3 and ZedBoard create CoAP messages 3-5 times faster than the Pi 1.

![Packet Read Time Graph](image)

**Fig. 7: Time to Read a Packet by the libcoap Client at 100 ms and 10 ms Interval**

Figure 7 shows the average time to read a message. The same speed difference between the platforms and test cases can be observed. While using libcoap the parsing time is slower than the build time of a single message, which is the opposite behavior compared to the Java applications.

In Table I we show the sum of building and parsing time of a single message. We can observe that all values for the Java implementations on all platforms range in the values of a few hundred µs. Only Cf on the Pi 1 is at 1332 µs. The
same streaming experiments are done to investigate the CoAP C-implementation libcoap. The results are ten times faster compared to the Java implementations. On a Pi 1 it takes 69µs to create and parse a message, on a Pi 3 and ZedBoard it takes 17µs and 18µs. Table II lists the overall timings measured using an application that requires to send a message every 10 ms. On the Pi 1 an increase of the time using jCoAP can be observed. As discussed in Figure 2 the overload situation results in large computational times if Cf is used on Pi 1. Native code (libcoap) on a Pi 1 is executed much faster at 60µs overall processing time, where no limitations occur at this data rate. The platforms Pi 3 and ZedBoard can also handle a transmission interval of 10 ms without any non-linear increase of the processing time. Using libcoap on those platforms, we measured the same processing time of 15µs.

### B. Fluctuation of Latency

Next to the average timings we evaluated the jitter. Running the Java implementations, some messages are delayed by a huge factor. For comparable results all applications (implemented with jCoAP and Cf) are executed on two ZedBoards.

Fig. 8: Fluctuation of Packet Build Time on ZedBoard using Californium

Figure 8 shows the build time of the messages sent by the server. Therefore, we measured the timings of 9000 messages. There are a lot of spikes which indicate large time values for single messages. Those needles do not occur for a burst of messages. It can be seen that after 1500 messages the processing time drops, which can be explained due to the run time compiler which optimizes the code after 1500 calls. This behavior can be also observed in all other Java measurements.

Fig. 9: Fluctuation of Packet Build Time on ZedBoard using jCoAP

On client side there are less needles regardless of the implementation (Figures 10, 11). Comparing both implementations with each other, we observe that jCoAP shows less needles (Figures 8, 9). The high timing values for single messages can be explained by the garbage collector of the underlying JVM. Two types of garbage collection are known [12], which result in different values of delay.

Fig. 10: Fluctuation of Packet Read Time on ZedBoard using Californium

We also measured the round trip time between two Pi 3 running jCoAP to estimate the latency and show the influence of the garbage collector (Figure 12). There, the CoAP client sends requests of 1024 bytes to the server, which responds with an echo message. After an echo is received by the client a new request is sent. The average round trip time is at 1.7ms, but there are single message round trip times that exceed this value by a factor of 10. 650 messages took longer than 3 ms, 56 messages where slower than 10 ms and 8 messages out of the set of 50,000 messages had a round trip time slower than...
A deeper investigation of the Garbage Collector induced interruptions of Java applications is done in [12]. The authors measured the RTT between a server-client application based on a medical DPWS stack (Devices Profile for Web Services), that was executed on an Intel Galileo Board and Raspberry Pi Version 1 - 3. They also observed two different kinds of garbage collections, the small and the large garbage collection. Next to this effect an RTT measurement over time has shown a significant drop in RTT after 1500 messages when the run time optimization of the JVM has been executed. In addition to our work the authors evaluated the "-Xcomp" flag of the JVM to force the compilation at start time. The use is not recommended because of highly increased start times ranging from 6 s to 13 s.

VI. CONCLUSION

The use of Java as a programming language and execution environment enables the deployment of applications to heterogeneous devices. There are JVMs for a range of architectures and operating systems available. Developers of distributed, e.g., smart home applications can use CoAP RESTful servers and clients that are implemented as open source software. We analyzed the performance of the CoAP Java implementation Californium to create a point-to-point streaming application using the asynchronous notification mechanism of CoAP. Due to high fluctuations of the packet processing within the Java application, large message queues have to buffer messages. The queues have to buffer up to 1s of incoming streaming data which results in an unacceptable latency of live streaming applications. The memory management of the JVM has to clean the heap because of memory allocations, which is also time consuming, for every single message. Similar qualitative results where observed using the CoAP Java implementation jCoAP, although the quantitative values of the packet processing where faster compared to Californium. In contrast to the Java implementations, we measured the packet processing times of the C implementation libcoap. Running native code on all platforms leads to much lower (factor 10) packet processing times. Furthermore, the fluctuations of these timings are much smaller even on a non-real-time Linux kernel and the achieved throughput is much higher compared to the Java applications (approx. 10s Mbit/s vs. 100s kbit/s). For timing sensitive applications like audio/video streaming we can conclude to use native code implementations.

ACKNOWLEDGEMENT

We would like to thank the German Federal Institute for Research on building, Urban Affairs and Spatial Development (BBSR) within the Federal Office for building and Regional Planning for their support in this project. This work is partially granted by BBSR within the scope of project “Zukunft Bau”.

REFERENCES


