Californium: Scalable Cloud Services for the Internet of Things with CoAP

Matthias Kovatsch
Department of Computer Science
ETH Zurich, Switzerland
Email: kovatsch@inf.ethz.ch

Martin Lanter
Department of Computer Science
ETH Zurich, Switzerland
Email: lanterm@student.ethz.ch

Zach Shelby
ARM Inc.
San Jose, CA, USA
Email: zach.shelby@arm.com

***Abstract*—The Internet of Things (IoT) is expected to interconnect a myriad of devices. Emerging networking and backend support technology not only has to anticipate this dramatic increase in connected nodes, but also a change in traffic patterns. Instead of bulk data such as file sharing or multimedia streaming, IoT devices will primarily exchange real-time sensory and control data in small but numerous messages. Often cloud services will handle these data from a huge number of devices, and hence need to be extremely scalable to support conceivable large-scale IoT applications. To this end, we present a system architecture for IoT cloud services based on the Constrained Application Protocol (CoAP), which is primarily designed for systems of tiny, low-cost, resource-constrained IoT devices. Along with our system architecture, we systematically evaluate the performance of the new Web protocol in cloud environments. Our Californium (Cf) CoAP framework shows 33 to 64 times higher throughput than high-performance HTTP Web servers, which are the state of the art for classic cloud services. The results substantiate that the low overhead of CoAP does not only enable Web technology for low-cost IoT devices, but also significantly improves backend service scalability for vast numbers of connected devices.**

I. Introduction – IoT and CoAP

In accordance with Moore's Law, the number of components per integrated circuit at minimal cost doubles approximately every two years. In the vision of the Internet of Things (IoT), however, this gain is not used to increase the computing power of devices, but to decrease power consumption, to integrate whole systems on a tiny chip, and in particular to minimize unit costs. Very low prices will lead to myriads of IoT devices deployed in homes, office buildings, factories, whole cities, and other environments of interest. This also means that most nodes in the IoT will remain resource-constrained and will require lightweight protocols. Yet with about 100 KiB of ROM and about 10 KiB of RAM, devices are capable of directly connecting to the Internet *in a secure manner*. These are so-called *Class 1* devices, for which lightweight IP solutions such as 6LoWPAN and RPL have been standardized in the Internet Engineering Task Force (IETF) [7], [19]. This allows for seamless integration of sensor and actuator nodes into the Internet, thereby connecting the virtual world of computers with the physical world of our daily lives.

For full convergence, however, devices and services must also interoperate at the application layer. When it comes to mashing up different services, the World Wide Web has proven most flexible and scalable. This motivated the *Web of Things* initiative, which advocates using the REST architectural style to design IoT applications and the ubiquity of HTTP to interact with devices [5]. HTTP over TCP has problems in constrained environments, though, in particular with the small frame sizes and the lossy links of low-power wireless communication. Instead of adding the problems of compression to the problems of HTTP, the IETF designed a new Web protocol from scratch: the *Constrained Application Protocol* (CoAP) [15]. CoAP follows REST, but is tailored to the requirements of low-cost devices and IoT application scenarios. It uses a compact binary format and runs over UDP (or DTLS when security is enabled), which also enables multicast communication. A messaging sub-layer adds a thin control layer that provides duplicate detection and reliable delivery of messages based on a simple stop-and-wait mechanism for retransmissions. On top, the request/response sub-layer enables RESTful interaction through the well-known methods `GET`, `PUT`, `POST`, and `DELETE` as well as response codes that are defined in accordance to the HTTP specification. CoAP resources are addressable by URIs, and Internet Media Types are used to represent resource state. RESTful caching and proxying enables network scalability. Yet CoAP offers features that go beyond HTTP 1.1, and hence make it a better fit for the IoT:

1) Resources are *observable*, that is, extra responses continuously push state changes to all registered clients [6].
2) The extension to a request/multiple-response pattern also enables RESTful *group communication* where multiple servers respond to a request that is sent to an IP multicast address [13].
3) CoAP includes a machine-to-machine *discovery mechanism* to find matching resources based on Web Linking [10], [14]. It uses either multicast or resource directories [16] where devices register on start-up.
4) *Application-layer fragmentation* allows blockwise on-the-fly processing of messages that would otherwise exceed the maximum transmission unit (MTU) of 1280 bytes or the potentially even smaller buffers of highly resource-constrained devices [3].
5) Finally, CoAP supports *alternative transports* such as Short Message Service (SMS) or Unstructured Supplementary Service Data (USSD), while maintaining interoperability at the application layer [17].

Given the low unit costs and open Internet standards, analysts expect more than 200 billion IoT devices by the end of 2020 [1]. While many research efforts have targeted the challenges of constrained environments with little resources and lossy radio links, the impact of a myriad of IoT devices on the existing Internet infrastructure so far has gained little attention. Nonetheless, IoT traffic is quite different from human-centered Web applications and file sharing. Instead of bulk data, IoT nodes will primarily exchange real-time sensory and control data in small but numerous messages. A typical scenario is using a cloud service to manage a large number of devices, process their data, and orchestrate their actuation. In this paper, we study the benefits of CoAP's low overhead for IoT cloud services, in particular scalability to handle huge numbers of concurrently connected IoT devices. Our contributions are:

- A system architecture that outperforms state-of-the-art Web servers as well as other CoAP solutions
- The first performance evaluation of CoAP in unconstrained environments such as the IoT service backend
- A re-implementation of our Californium (Cf) CoAP framework, which is explicitly designed for scalable IoT cloud services and is publicly available at the Eclipse Foundation: http://www.eclipse.org/californium

II. Scalable Systems for the IoT Backend

Our system architecture for CoAP-based IoT cloud services is inspired by previous work for highly concurrent Internet services, in particular the *Staged Event-Driven Architecture (SEDA)* [18] and the *PIPELINED* architecture [4]. Other related work is discussed in Sec. IV.

SEDA splits the message-handling process into multiple stages. Each stage consists of an incoming event queue, a thread pool, and an event handler that executes the logic of the stage. The threads pull events from the event queue and invoke the event handler, which can dispatch new events to the next stage through their connecting queue. Each stage is managed by a controller that can dynamically change the configuration according to a policy. Therefore, each stage of a SEDA server can self-tune itself to have an optimal number of threads.

PIPELINED can be considered a special form of SEDA, as a pipeline is a chain of single-threaded stages. The threads belong to the same pool, though, which results in better cache behavior when switching between stages. Usually, a server creates one pipeline per core to achieve good scalability.

We propose a 3-stage architecture as depicted in Fig. 1. It is mainly based on the lessons learned from our initial Californium (Cf) implementation [9]. Like SEDA, each stage is decoupled by queues and has its own concurrency model. The thread pool size does not depend on a dynamic scheduling policy, though, but only on the application requirements and the execution platform. This reduces complexity and the overhead of monitoring tasks. By default, the number of threads equals the number of cores and multiple messages can traverse our processing chain in parallel, similar to PIPELINED. We re-designed our CoAP framework from scratch and made the new Californium available under EPL+EDL dual-licensing.

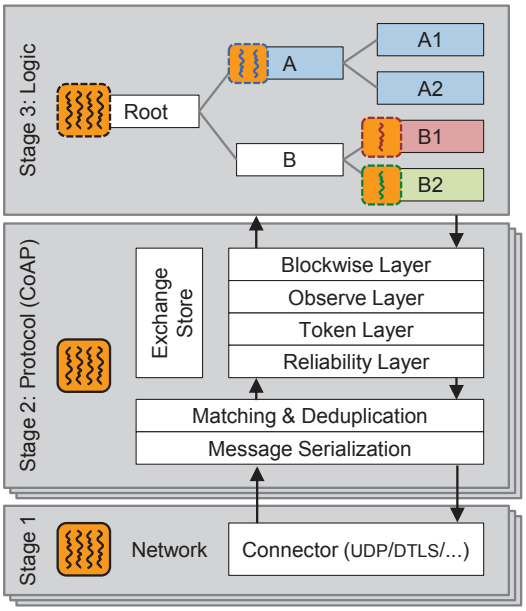


Fig. 1. Our architecture has three decoupled stages with individual thread pools. The CoAP protocol is executed in the second stage: the Blockwise Layer fragments requests and responses for blockwise transfers (in atomic fashion [3]); the Observe Layer handles observe relationships and orders notifications; the Token Layer ensures the unique matching of responses to open requests; the Reliability Layer manages retransmission timeouts. Matching incoming messages to their state in the Exchange Store happens outside this stack, so that it becomes a pure processing pipeline without synchronization overhead.

The **network stage** (see Fig. 1 bottom) is responsible for receiving and sending byte arrays over the network. It therefore abstracts the transport protocol, which is typically UDP or DTLS for CoAP. Micro-benchmarks show that using more than one thread to move data through the socket can increase the throughput on some platforms, but also decrease it on others. Since we wrap the process for receiving and sending in its own independent stage, the server can chose an optimal number of threads for a specific platform without affecting other stages. For our Windows 7 quad-core reference system, for instance, using four receiver and four sender threads (4/4) instead of one each (1/1) almost doubles the achievable data rate of the provided UDP socket. On Linux (RHEL6), however, increasing to two threads (2/2) causes a 40% setback in throughput. Thus, the default configuration uses one sender and one receiver thread per core on Windows and a single one each on Linux. Other configurations can further improve the performance depending on the platform. On a Xeon-based server system with 10-gigabit Ethernet, for instance, a 1/2 configuration achieved best results for both operating systems.

The **protocol stage** executes the CoAP protocol and has a thread pool with as many threads as cores by default. We kept the multi-layer CoAP stack of the initial Cf design for its advantages in understandability, maintainability, and extensibility of the code. Originally, each layer managed its own state to execute the protocol, e.g., the Transaction Layer stored all messages and timers to perform retransmissions and the Token Layer did the housekeeping for open requests and their tokens. This makes the stack perfectly modular, but

at the price of a high synchronization overhead and risk of memory leaks because of the scattered data. Therefore, we separated bookkeeping and processing: An `Exchange` object now holds all data and timers necessary for a request/response exchange and our stack is a pure processing pipeline. When a message arrives, a matching step outside the stack accesses the Exchange Store (a `ConcurrentHashMap`) only once to retrieve the necessary state or to create a new exchange. The associated exchange is then passed along with the current message, layer by layer. Once an exchange completes, the system removes its reference to the exchange and the garbage collector automatically reclaims the memory.

The **business logic stage** depends on the role: (1) Servers host Web resources that are structured in a logical tree. By default, the server stage has no thread pool and a thread from the protocol stage invokes the resource handler, which takes the request and produces the corresponding response. Developers can, however, choose their preferred concurrency model (e.g., to prioritize or balance their Web resources) by optionally configuring thread pools at individual resources (using a Java `Executor` in our implementation). If a resource does not define a thread pool, the thread pool of its parent, transitive ancestor, or eventually the protocol stage will be used. (2) Clients can also define their own concurrency model and use our `CoapClient` API, which supports synchronous and asynchronous requests. Synchronous calls hand the request over to the protocol stage and block until the response is delivered by a protocol-stage thread. Asynchronous API calls return immediately and by default the protocol-stage thread will execute the response handler registered for this exchange—similar to a Web resource handler. Both roles can be combined by acquiring a `CoapClient` that is associated with a specific Web resource, allowing an application model similar to Actinium [8].

We encapsulate the protocol and network stages into `Endpoint` objects. Thus, alternative transports (UDP, DTLS, SMS, etc.) can provide their variations of the CoAP stack in a modular way. A server can also have multiple endpoints to make resources available over multiple channels and also to easily distinguish between different network interfaces (a problem introduced by many OS datagram APIs). Our system already comes with endpoint classes for UDP and DTLS; a multicast endpoint is currently under development.

III. Evaluation

We evaluate our system for scalable IoT cloud services through benchmarks and comparison to the state of the art, both high-performance HTTP Web servers and other CoAP solutions. IoT applications can have different communication models, which results in different evaluation scenarios:

1) *Cloud service as server*: This is the typical scenario for Web 2.0 and HTTP-based IoT applications. Since HTTP is missing an efficient server push mechanism, IoT devices are usually programmed as HTTP clients that `POST` their data to the service (and are able to sleep in between). For CoAP, this scenario applies for resource directories that are used for device management and discovery [16]. IoT devices register on start-up and periodically update their status using requests while other devices or services perform look-ups.
2) *Cloud service as client*: With CoAP's push mechanism, the server role has become practical for IoT devices, which enables an application-agnostic IoT device infrastructure (cf. [9]). The cloud service is a client and takes over the role of a Web mashup engine that sends requests to the devices for actuation and observes resources for monitoring and sensing tasks.
3) *Cloud service as both*: Complex business logic requires both roles by the service, e.g., to observe device resources and to provide computed results again as resources for other services. For instance, the *OMA Lightweight M2M (LWM2M)* specification is built around this scenario. The LWM2M server is a resource directory that receives registration and look-up requests from devices, but also issues requests to their resources. This means that IoT devices are hybrids as well: they are primarily CoAP servers, but use requests to register with the service (and thereby open ports in firewalls).

We use the *cloud-service-as-server* scenario because it allows for a direct comparison with HTTP servers, the current state of the art for scalable cloud services. For Californium, the results for this scenario can be transferred to the other two scenarios (in particular the handling of incoming observe notifications), since clients and servers use the same stack due to the shared message format and equal processing of requests and responses.

A. CoAPBench

While there are plenty of benchmark tools available for HTTP, to the best of our knowledge, there is none for CoAP. Therefore, we developed *Cf-CoAPBench*, a tool similar to ApacheBench. It uses virtual clients to meet the defined concurrency factor. To have enough resources to saturate the server and keep all collected statistics in memory, CoAPBench can be distributed over multiple machines. A master controls the benchmark by establishing a TCP connection to all slave instances. We designed this master/slave mechanism to be able to execute third-party benchmark tools as well. Thus, we can run ApacheBench distributed and synchronized over multiple machines and bring even very powerful HTTP servers into saturation. Note that master and slaves only communicate before and after the experiment, so that the network traffic is not influenced by our tool.

CoAPBench adheres to basic congestion control, that is, each CoAP client sends Confirmable requests and waits for the response before the next request is issued. We disable retransmissions, though, to not blur the numbers of sent and successfully handled requests. In case of message loss, a client times out after 10 seconds, records the loss in a separate counter, and continues with a new request.

B. Setup

The evaluation focuses on the performance and scalability of the protocol handling by the systems—not the business logic. Thus, CoAPBench issues simple `GET` requests to a "/benchmark" resource, which responds with a short "hello world."[1] CoAP and HTTP requests and responses are semantically equal (including header field information).

We run CoAPBench distributed on three machines as depicted in Fig. 2. The hardware for the system under test is specified in the respective experiments below. We increase the concurrency factor stepwise from 10 to 10,000 and stress the server for 60 seconds, followed by a 15 seconds cool-down period before continuing with the next step. To have deterministic results, we also disable Hyper-Threading and Turbo-Boost on the machine executing the systems under test.

C. Multi-core Support

Our new architecture design specifically focuses on the utilization of multiple CPU cores. We evaluate this by measuring the throughput with different processor affinity settings (one run each) and comparing the results of the new Californium to the initial implementation (*Initial-Cf*) for reference. For these benchmarks, the CoAP servers are running on a laptop machine with an Intel i7-3720M processor at 2.6 GHz, 24 GiB RAM, and Intel 82579LM Gigabit Ethernet adapter.

Multi-threading support was added rather late in the initial Cf design. A single thread was receiving messages from the socket and executed the CoAP stack upward. Only at the top, the request was handed over to a fixed-size thread pool, which dispatched one of its ten workers to execute the resource handler and send the response down the multi-layer stack and through the socket. This caused significant synchronization overhead, since up to eleven threads worked on several hashmaps that were distributed over all layers of the protocol stack.

Fig. 3 shows that our proposed architecture scales better with the number of available cores. Using two instead of a single core almost exactly doubles the throughput. On four cores, we perform about 3.4 times better than on one core, which is reasonable since not all tasks can be parallelized. Socket I/O, for instance, is partly done in the kernel and always runs on 'Core 0' on Windows machines. Our maximum throughput (on this machine) is 137,592 requests per second versus 71,255 requests per second for the initial Cf.

[1]To achieve the best results, we request the 'natural' resource of each server: for Apache, this is `/benchmark/index.php`, for Tomcat and Node.js `/benchmark/` (with slash), and `/benchmark` for the remaining servers.

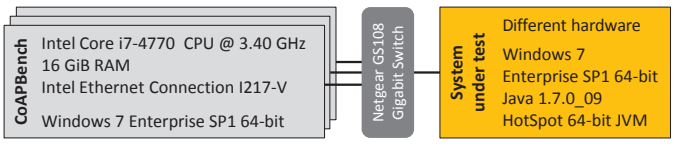
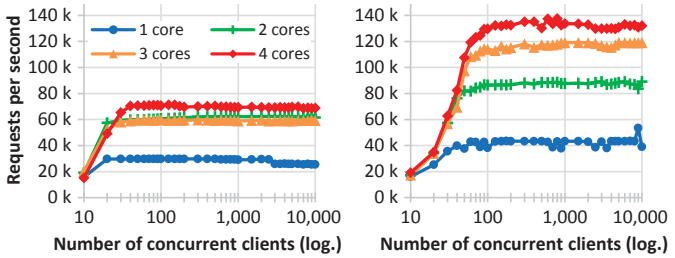


Fig. 2. All CoAP and HTTP servers are tested in this setup.



(a) Initial-Cf (b) Californium

Fig. 3. We evaluate the utilization of the available CPU cores using an affinity tool. Both CoAP systems stay stable at high concurrency factors. Our new system architecture achieves a much higher throughput, though.

D. Comparison with the State of the Art

For this experiment, we compare our system to six state-of-the-art HTTP servers (*Apache HTTP Server (2.4.6)*, *Tomcat (7.0.34)*, *Node.js (0.10.20)*, *Grizzly (2.3.6)*, *Jetty (9.1.5)*, and *Vert.x (1.3.1-final)*) as well as four CoAP implementations, our *Initial-Cf*, the *Sensinode NanoService Platform*, *nCoAP*, and the *OpenWSN Python Library* (see Sec. IV for more details). The servers run on a machine similar to the CoAPBench machines described in Fig. 2. We run Californium in the default configuration for Windows (4/4/4) and also optimize the configurations of the other servers for this platform. The reported results are averages over ten runs.

1) With Keep-alive: First we evaluate the performance with the keep-alive option of HTTP/1.1, that is, a client re-uses a single TCP connection for all subsequent requests. This saves costly round-trip times for the handshake and remedies the slow-start mechanism of TCP. Here, Vert.x performs best among the HTTP solutions as seen in Fig. 4. It impressively solves the so-called 'C10K problem,' being able to maintain 10,000 concurrent TCP connections to its clients at high throughput. Vert.x is the only server with high standard deviation, though, which is indicated by error bars ($\pm 1\sigma$). Jetty shows stable performance for high concurrency factors as well, which is why it is also popular for IoT applications. Tomcat has good performance on its first run, but automatically disables keep-alive once it experiences high concurrency factors and the throughput drops for all subsequent runs (indicated by the dotted light blue line). Thus, we limit the number of concurrent clients to 200 to keep Tomcat in the range it is originally designed for. Its successor Grizzly scales better and only gives in at around 5,000 simultaneous clients.

The Initial-Cf and Sensinode CoAP servers exhibit similar performance as Jetty and Grizzly. Although nCoAP is connectionless like all CoAP servers, it drops from a good first run (indicated by the dotted gray line) down to about 30,000 requests per second on average. Overall, Californium performs best with up to almost 400,000 requests per second and stable throughput for high concurrency factors. We also indicate the standard deviation for our system; it is so small, however, that the error bars are mostly hidden behind the data points. We pay for the high throughput under heavy load with a slightly slower growth in the beginning: Due to the context switch between the network stage and the protocol stage, the

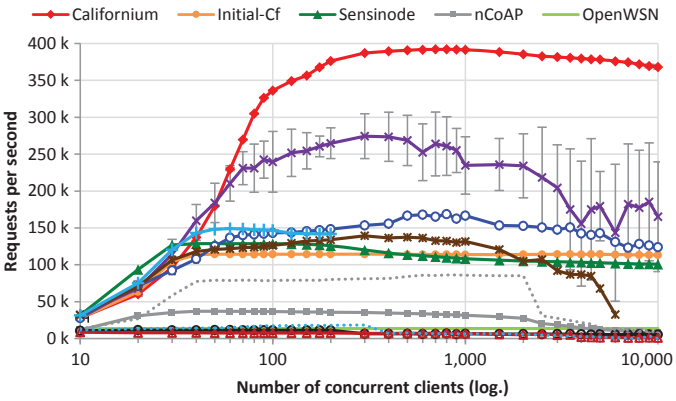


Fig. 4. Throughput with HTTP keep-alive: Modern architectures for HTTP servers can handle a high number of parallel TCP connections. However, CoAP in general scales better for high concurrency factors and our system can handle the most requests per second.

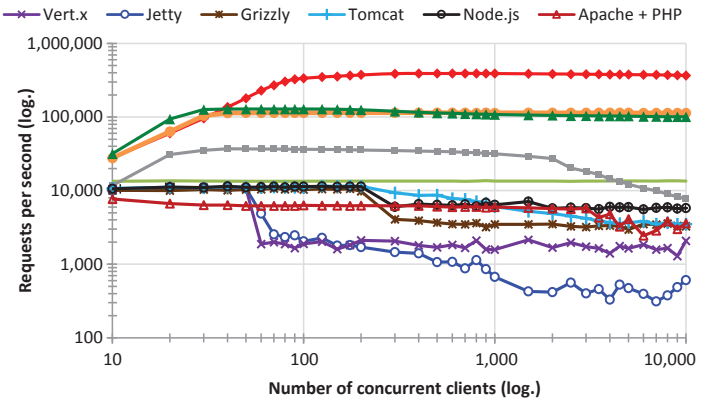


Fig. 5. Throughput without HTTP keep-alive: HTTP servers mainly suffer from TCP overhead: the three-way handshake and slow-start mechanism. Yet short-lived message exchanges from myriads of devices is the expected traffic pattern in the IoT.

processing time is longer than, for instance, the one of the very efficient Sensinode implementation. Since CoAPBench always waits for a request to finish before issuing the next one, the clients spend more time waiting for the response and hence a few clients cannot saturate our system. Yet for low concurrency factors, 99% of all requests still finish in under half a millisecond.

The Apache Web server with PHP is not designed for highly scalable cloud services. However, Apache is still the most popular Web server and is included for reference. It achieves a steady throughput at about 7,000 requests per second until 3,500 concurrent clients. Beyond that point, the performance slowly declines.

2) Without Keep-alive: In an IoT scenario, devices often close the connection after each request to resume sleep. Furthermore, we expect the high message rates to originate from tens of millions of IoT devices sending alternately in minute or hour intervals, rather than tens of thousands sending constantly at very high rate. Scenarios for this are sensors deployed throughout a smart metropolis or smart metering. As a consequence, we focus on the throughput behavior without the keep-alive option (i.e., a new TCP connection for each request), since a cloud service simply cannot maintain a standing connection to every device. This does not affect the CoAP results, since it is connectionless, and we can use the measurements reported above for comparison.

Fig. 5 shows that HTTP suffers from the overhead of TCP, whose avoidance was in fact one of the design goals behind CoAP. Here, we use a logarithmic scale on the y-axis to cover all results in a single graph. Apache actually performs similarly with and without keep-alive, so its series can be used for reference when comparing Fig. 4 and Fig. 5.

Having a stable throughput at high concurrency factors, Node.js now performs best among the HTTP servers. The cluster mode has good scalability for short-lived TCP connections and can handle almost 6,000 requests per second at 10,000 simultaneous clients. Node.js is followed by Tomcat, Apache, and Grizzly, which all converge toward about 3,500 requests per second at the end. Without keep-alive, the servers designed to overcome the C10K problem, Vert.x and Jetty, drop from 10,000 to about 2,000 requests per second already for more than 50 concurrent clients.

In highly concurrent scenarios as found in the IoT, CoAP solutions perform much better than HTTP-based cloud services in terms of scalability. The logarithmic scale shows that our system also performs an order of magnitude better than other CoAP implementations. Compared to state-of-the-art HTTP solutions, CoAP can achieve a 33 times higher throughput for conservative concurrency factors (up to 200). For concurrency factors as expected in IoT scenarios, it even has a 64-fold increase in throughput compared to HTTP. This is mostly due to the 3-way handshake and tear-down of TCP connections, which leads to at least 9 messages to execute a single request. The overhead of the verbose HTTP headers slows the systems down further, especially considering the small payloads that are typical for IoT traffic.

IV. Related Work

Researchers have been enhancing architectures for Web servers since the advent of the Web itself to scale with the ever-growing traffic. Because of the similarity between CoAP and HTTP (both are REST implementations), we took HTTP-based designs into account, in particular since almost no CoAP backend system architectures exists so far in practice.

A. HTTP Server Architectures

The *Multi-Threaded (MT)* server architecture assigns each incoming TCP connection to one worker thread that handles the request. This allows to serve each request without delays and best performance is achieved when the number of threads equals the number of expected concurrent requests [2]. When the number of concurrent clients grows, however, the large number of threads causes significant synchronization overhead and high memory usage due to the separate stacks. This is the main drawback of MT, especially when HTTP/1.1 keeps connections alive for a long time. The MT architecture is used in the *Apache HTTP Server* and the *Tomcat* server for Java Servlets and JavaServer Pages (JSP) when using the original blocking I/O connector, which is the default.

A more recent trend is to use non-blocking I/O in a *Single-Process Event-Driven (SPED)* architecture. A single event-dispatching thread keeps popping tasks from a global queue and executes them one after the other. Thus, no synchronization is required, context-switches can be saved, and data are always cache-local. However, SPED cannot directly benefit from multiple cores and many operating systems do not provide suitable support for non-blocking operations [11]. As a result, SPED has been augmented with helper threads to wrap blocking I/O. This extension is called Asynchronous Multi-Process Event-Driven (AMPED) [11]. A multi-core variant is Co-AMPED, which runs one AMPED process per core [12]. *Node.js*, which introduces server-side JavaScript for cloud services, is a good example for the pure SPED architecture. We use its cluster mode, though, which enables replication similar to Co-AMPED. Project *Grizzly* is originally a component of the GlassFish Java Enterprise Edition application server that provides the HTTP server interface. It uses Java's 'New I/O' API for higher scalability, also following the concepts of SPED. Using a central thread pool that can use all cores, *Jetty* connects different kinds of connectors with handlers. *Vert.x* is staged and has an event-driven architecture. It combines non-blocking network I/O (using the Netty project) and support for several languages to implement the business logic in so-called 'verticles.' To become scalable on multi-cores, the default is to run multiple instances of the same verticle in parallel (one per core for our benchmarks).

B. CoAP Implementations

There exist a number of CoAP implementations, most of them targeting resource-constrained environments, though. In this paper, we focus on the use of CoAP in the IoT service backend. Here, the *Sensinode NanoService Platform*[2] is a commercial solution that offers good support for industry-relevant features such as OMA Lightweight M2M support and in-memory data grid caching for big data. *nCoap*[3] and *jCoAP*[4] are open-source Java projects that are best comparable to our framework. The latter, however, only implements a deprecated draft version of CoAP at the time of writing, and hence is not included in the benchmarks. The OpenWSN project[5] provides a Python library, which primarily targets easy interaction with devices, though. Thus, it is benchmarked non-competitively.

V. Conclusions

This paper presents a system architecture for scalable IoT cloud services based on CoAP. It is inspired by proven architectures for Web servers that have evolved over time. Our evaluation shows that our 3-stage architecture fully utilizes the resources of today's multi-core systems. Our Californium (Cf) reference implementation, whose source code is publicly available, outperforms other CoAP systems with a three times higher throughput.

As a more general result, we show that CoAP's low overhead also has significant advantages over HTTP in the IoT service backend. With up to 64 times higher throughput than state-of-the-art Web servers, CoAP-based cloud services can handle the expected myriad of IoT devices in an efficient way. Thus, we propose to limit the use of CoAP-HTTP cross-proxies to the transitional period. In the long-run, IoT cloud services and Web integration platforms need to speak CoAP directly to be able to scale to vast numbers of concurrently connected devices. Note that when an IoT cloud service interacts with a small number of other services or a load balancer, long-lasting TCP connections using HTTP/2.0 or the upcoming CoAP-over-TCP binding are still a good choice.

This work focuses on the essential networking and backend support technology to implement the vision of the IoT. In future work, we want to take security aspects into consideration: A DTLS handshake, for instance, poses similar problems as establishing a TCP connection. To achieve optimal security profiles for the IoT, similar experiments need to be run with CoAPS and HTTPS once the (D)TLS v1.3 specification stabilizes.

[2]http://www.sensinode.com/

[3]https://github.com/okleine/nCoAP

[4]https://code.google.com/p/jcoap/

[5]http://www.openwsn.org/

References

[1] Worldwide Internet of Things (IoT) 2013–2020 Forecast: Billions of Things, Trillions of Dollars. Market Analysis 243661, IDC, 2013.

[2] V. Beltran, J. Torres, and E. Ayguade. Understanding Tuning Complexity in Multithreaded and Hybrid Web Servers. In *Proc. IPDPS*, Miami, FL, USA, 2008.

[3] C. Bormann and Z. Shelby. Blockwise transfers in CoAP. draft-ietf-core-block-14, 2013.

[4] G. S. Choi, J.-H. Kim, D. Ersoz, and C. R. Das. A Multi-threaded PIPELINED Web Server Architecture for SMP/SoC Machines. In *Proc. WWW*, Chiba, Japan, 2005.

[5] D. Guinard, V. Trifa, and E. Wilde. A Resource Oriented Architecture for the Web of Things. In *Proc. IoT*, Tokyo, Japan, 2010.

[6] K. Hartke. Observing Resources in CoAP. draft-ietf-core-observe-14, 2014.

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

[8] M. Kovatsch, M. Lanter, and S. Duquennoy. Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications. In *Proc. IoT*, Wuxi, China, 2012.

[9] M. Kovatsch, S. Mayer, and B. Ostermaier. Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In *Proc. IMIS*, Palermo, Italy, 2012.

[10] M. Nottingham. Web Linking. RFC 5988, 2010.

[11] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proc. USENIX*, Monterey, CA, USA, 1999.

[12] S. Palchaudhuri, R. Kumar, and A. K. Saha. A Web Server Architecture for Symmetric Multiprocessor System. Project Report for Comp520, Department of Computer Science, Rice University, 2000.

[13] A. Rahman and E. Dijk. Group Communication for CoAP. draft-ietf-core-groupcomm-19, 2014.

[14] Z. Shelby. Constrained RESTful Environments (CoRE) Link Format. RFC 6690, 2012.

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

[16] Z. Shelby, S. Krco, and C. Borman. CoRE Resource Directory. draft-ietf-core-resource-directory-01, 2013.

[17] B. Silverajan and T. Savolainen. CoAP Communication with Alternative Transports. draft-silverajan-core-coap-alternative-transports-04, 2014.

[18] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *Proc. SOSP*, Banff, Canada, 2001.

[19] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. P. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC6550, 2012.