ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

RESEARCH GROUP FOR
Distributed Systems

Master's Thesis

at the Institute for Pervasive Computing, Department of Computer Science, ETH Zurich

Scalability for IoT Cloud Services

by Martin Lanter

Spring 2013

ETH student ID:	08-928-368
E-mail address:	lanterm@student.ethz.ch
Supervisors:	Dipl.-Ing. Matthias Kovatsch
	Prof. Dr. Friedemann Mattern
Date of submission:	15 October 2013

Declaration of Originality

Zurich, 15 October 2013

Martin Lanter

Acknowledgements

Many thanks to Matthias Kovatsch for his dedicated support and the inspiring discussions we had about the Internet of Things. Special thanks for the opportunity to write on a scientific paper and our proposal for the blockwise transfer of CoAP messages.

Special thanks to the IT Service Group (ISG) of the Computer Science Department at ETH Zurich for the provision and setup of two NUMA machines for my server benchmarks.

I would like to gratefully and sincerely thank my parents for their lifelong support for everything I do. Your support allowed me to focus on my studies, projects, and in particular this thesis.

Abstract

The Internet of Things (IoT) is envisioned as a global network of billions of smart devices. Traditional protocols known from the Internet do not fit well with the constrained environment these devices typically are found in. Thus, the IETF has standardized the Constrained Application Protocol (CoAP). Devices can communicate over CoAP and be integrated into Cloud services. We focus on the backend of such services that run on an unconstrained machine. High numbers of services and especially clients call for efficient processing and scalable services with high throughput. HTTP server architectures that perform well in regards to these properties have been researched for over 15 years. We analyze HTTP server designs thoroughly and apply the lessons learned to a new CoAP framework for IoT Cloud services, a re-implementation of Californium (Cf). We present a CoAP server design that is scalable and able to fully utilize a modern multi-core computer. We evaluate our new and the old version of Californium and five HTTP servers on a 4-core SMP and a 16-core NUMA system. Our implementation achieves a throughput of about 140,000 requests per second and even outperforms state-of-the art HTTP servers by 15%-45%. The results substantiate that CoAP is superior to HTTP for IoT services not only in constrained but also unconstrained environments. Furthermore, we propose a powerful, yet simple API to develop scalable services for the IoT Cloud in an efficient and productive way.

Contents

Acronyms

AMPED	Asynchronous Multi-Process Event-Driven
API	Application Programming Interface
CoAP	Constrained Application Protocol
DTLS	Datagram Transport Layer Security
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IoT	Internet of Things
JVM	Java Virtual Machine
MID	message identifier
MP	Multi-Process
MT	Multi-Threaded
NUMA	Non-Uniform Memory Access
PIPELINED	Multi-Threaded Pipelined
REST	Representational State Transfer
SEDA	Staged Event-Driven Architecure
SPED	Single-Process Event-Driven
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Objectives

The student will evaluate alternative API designs in terms of learnability, usability, maintainability, extendibility, and efficiency. The most promising solution shall be implemented to improve functional scalability of the framework.

The student shall re-design Cf's internal architecture to improve load scalability, i.e., the throughput of handled requests.

The student shall make use of an appropriate testbed that enables stress testing the system.

CoAP shall be compared to HTTP in an M2M scenario. Throughput, latency, and reliability shall be evaluated using the testbed and a deployment of real devices, which will be provided.

1 Introduction

The vision of the Internet of Things (IoT) is a world in which billions of everyday objects are interconnected and extend today's Internet pervasively into our daily lives. Sensors and actuators of all forms and purposes and even the infrastructure of whole cities are expected to be globally accessible and controllable in real-time [31]. Driven by Moore's law, the density of transistors continuously increases. Nevertheless, chip manufacturers are more likely to utilize technological advances by reducing the size, costs and energy-consumption of devices instead of their computing power [7]. Therefore, these devices and their ability to communicate with others are constrained and traditional protocols known from the Internet do not fit well. The *Internet Engineering Task Force (IETF)* has standardized the Constrained Application Protocol (CoAP) which is designed for the constrained environment that we find within the IoT [42].

CoAP implements a subset of HTTP's RESTful [18] operations (GET, POST, PUT and DELETE) and allows for a straight-forward mapping between the two protocols. Therefore, they are often referred to as being cousins. However, CoAP has its own mechanisms for (optional) reliability, observation, group communication, and discovery of resources.

An essential part of the IoT is the autonomous, smart behavior of its devices. A middleware allows building services that run in the Cloud and interoperate with devices. Interoperability can be bidirectional in the sense that services access and control devices and vice versa. Services can be combined to build new services and top-level applications can be built to export the system's functionality to the end-user [2, 9]. We focus on the backend of services. We assume services to run on a strong, unconstrained machine and simultaneously processing requests in the role of a server or send requests in the role of a client.

1.1 Background

In 2011, the Californium (Cf) project started at ETH Zurich. Cf is a modular, open-source framework that facilitates deployment of backend services [25]. Cf serves as intermediary between the logic of a service and the IoT, i.e., it abstracts the communication over CoAP and provides the developer with a powerful API to develop services in an efficient and productive way. A first version of Californium was intended as prototype and "running code" implementation of the CoAP standard and has progressively grown to test CoAP extensions. In 2012, support for DTLS [22, 39] and CoAP and HTTP proxies [14] were added to Cf. Actinium (Ac) [24] is a runtime container for JavaScript services built upon Cf. Ac exposes a RESTful interface to install, configure, and update JavaScript services, which can build an arbitrary resource structure to export their functionality.

Originally not designed for multi-threading, Cf does not scale well in the presence of a multi-core processor. The belated addition of multiple threads has led to race conditions, e.g., with non-thread-safe caches. The design of a static message gateway, called a "communicator," limits Cf to one port only, and therefore to only one transport protocol, one CoAP processing implementation and one tree of resources. In particular, the integration of DTLS and proxies was not seamless. Furthermore, Cf contained several bugs and inconveniences that required incisive reimplementation. Therefore, it appeared more economical to start over and use the learned lessons from the first version to design and implement a new, second version with a better architecture, higher efficiency, and scalability and an improved API. In this thesis, we call the first version "Old Californium (OCf)" and our new solution "Californium (Cf)".

Hosting a high number of services that all communicate with a large amount of devices can lead to significant demand in computing power and message throughput. Our goal is to design and implement a server that is able to utilize today's prevalent multi-core processors to perform well in both regards. Architectures that improve the throughput of servers have been subject to research for over 15 years, albeit with focus to CoAP's cousin HTTP. We believe that insights into architectures for HTTP servers are invaluable for the design of a CoAP server. Therefore, this thesis gives a detailed survey in the field of server-architecture design and discusses implications that go inherent with CoAP and HTTP. With the knowledge gained from OCf and HTTP servers, we design a new scalable CoAP server architecture for Californium and compare its throughput to OCf and five state-of-the-art HTTP servers.

1.2 Related Work

Several CoAP implementations already exist and either target constrained or unconstrained platforms. CoAPBlip for TinyOS [36], CoAPSharp[1] for the .NET Micro Framework, SMCP[2], libcoap [26], and Erbium (Er) for Contiki [23] are optimized for embedded devices. Although these libraries could also be deployed on an unconstrained platform, they are not designed for scalability and are not suitable for developing and maintaining complex service systems. Sensinode[3] has developed the "Sensinode NanoService Platform" that includes libraries in C and Java for devices. These libraries are commercial and not

[1]`http://www.coapsharp.com/`, October 10, 2013
[2]`https://github.com/darconeous/smcp`, October 10, 2013
[3]`http://www.sensinode.com/`, October 10, 2013

publicly available. For unconstrained platforms, there is the CoAP Python library by UC Berkeley[4], which primarily targets easy interaction with devices, though. The two Java frameworks jCoAP[5] and nCoap[6] both target unconstrained platforms and are best comparable to Californium. However, both do not implement the latest CoAP draft version 18 and are still in an early optimization stage. Therefore, we cannot yet compare Californium to those systems. None of these projects focus on scalability to the extent that this thesis does. To the best of our knowledge, there is no related work on this topic yet. Commercial solutions might have similar goals but there are no results published yet.

1.3 Outline

The particular, the contributions of this thesis are:

1. An extensive scalability evaluation of CoAP for service backends in comparison to HTTP solutions.
2. The design of a scalable, efficient and extensible architecture for CoAP servers and a simple open-source CoAP implementation.
3. A novice-friendly API for synchronous and asynchronous CoAP operations
4. A generic, distributed benchmark tool for CoAP servers

Chapter 2 explains and compares CoAP and HTTP in detail and discusses different requirements for respective servers. In Chapter 3, we present six server architectures and variations that were proposed during the last 15 years. We will use the gained knowledge to design a new architecture for Californium in Chapter 4. Design decisions and implementation details are covered in Chapter 5. In Chapter 6, we evaluate Cf, OCf and five HTTP servers on a commodity quad-core notebook and on a many-core NUMA system and discuss the results with respect to scalability in terms of multiple cores and large number of concurrent clients. Chapter 7 points out some optimizations that improved Californium's performance on different platforms. Chapter 8 presents our new novice-friendly API to develop CoAP services and to send asynchronous and synchronous CoAP requests. Finally, we conclude in Chapter 9.

[4]`http://www.openwsn.org/`, October 10, 2013
[5]`http://code.google.com/p/jcoap/`, October 10, 2013
[6]`https://github.com/okleine/nCoAP`, October 10, 2013

2 CoAP vs. HTTP

The *Constrained Application Protocol (CoAP)* understands itself as “a specialized web transfer protocol for use with constrained nodes and constrained [. . .] networks” [42]. In the IoT, constraints on nodes typically emerge in terms of limited power supply, manufacturing costs, RAM, ROM, and generally low processing capabilities. Yet, constrained nodes, i.e., devices, are powerful enough to send and receive network packets and benefit from a connection to the Internet as they can be integrated into a distributed service. A protocol such as the *Hypertext Transfer Protocol (HTTP)* [5,17], however, is too expensive in terms of implementation code space and network resource usage [7]. Therefore, CoAP has been designed to serve as a powerful, yet simple communication protocol in such an environment. Since CoAP can easily be mapped to HTTP and since both follow the *Representational State Transfer (REST)* architectural style, CoAP and HTTP applications can be connected via transparent proxies and integrated into the same system. HTTP is based on the reliable *Transmission Control Protocol (TCP)* [35]. Its reliability encompasses not only the autonomic retransmission of unacknowledged messages but also duplicate detection, message ordering, segmentation of large messages, and flow and congestion control. CoAP instead is based on the unreliable *User Datagram Protocol (UDP)* [34], and therefore has to implement certain guarantees of TCP on its own. Since TCP was developed for the unconstrained Internet, CoAP uses different techniques that better suit the constrained environment.

An entity participating in the CoAP protocol is called an endpoint [42]. An endpoint lives on a network node and is identified by its IP address, port, and security association. CoAP can be thought of as having two sublayers: the request/response-layer and the message-layer. Messages are either confirmable (CON), non-confirmable (NON), acknowledgements (ACK) or resets (RST). Confirmable and non-confirmable messages carry requests or responses. When an endpoint receives a confirmable message, it replies with an acknowledgement. The response to a confirmable request can be sent piggybacked with the ACK or in a separate confirmable response. An endpoint retransmits confirmable messages with an exponentially increasing back-off timer until it receives an acknowledgement, a reset or the maximum retransmission count is reached. [1] If an endpoint receives a CON or NON that it does not know how to process, it rejects it with a RST. A message is identified by a message ID (MID) and an endpoint needs to temporarily remember incoming MIDs to detect duplicates. On the request/response-layer, requests have a method code (GET, POST, PUT, or DELETE) and responses have a response

[1]Typically, the maximum retransmission count is 4, which allows for 5 transmissions in total.

code (either of class 2.xx (success), 4.xx (client error), or 5.xx (server error)). A token, chosen by the client, serves as identifier for a request. The server endpoint must include the request token in the response so that the client endpoint knows to which request the response belongs to. Additionally, CoAP requests and responses can be accompanied by simple options, similar to HTTP header options. For example, options may describe the content format or destination URI.

It is important to note that CoAP is more than just a compressed form of HTTP and moreover provides several features that are beneficial in an M2M application. In this section, we discuss how CoAP + UDP are superior to HTTP + TCP in constrained environments.

2.1 Conceptual Comparison

Fewer messages: A typical CoAP exchange consists of 2 messages, i.e., a request and a response. In contrast, an HTTP request first requires the client to establish a TCP connection and later terminate it. This results in at least 9 messages for only one request [11]. Note that this argument is not necessarily true for large payloads. After TCP's slow-start phase, it is able to send multiple packets at once and acknowledge all of them with a single acknowledgement. CoAP's blockwise transfer [8] though, requires an acknowledgement for each block and leads to more messages and higher transfer time. Since we expect the majority of CoAP messages to be rather short, this is of less importance. However, CoAP's blockwise mechanism allows a constrained server to not only receive but also process a large request block-by-block. This would not be possible if we used HTTP and TCP.

Compressed format: CoAP encodes option values in binary format while an HTTP request is one large, verbose text. This saves the effort for converting numbers as text to integers and makes the encoding more compact in general. A minimum CoAP header is only 4 bytes long and a minimum UDP header is only 8 bytes long. In contrast, a minimum TCP header alone is 20 bytes long plus what comes from HTTP[2]. As a result, a message occupies less memory in a buffer or a network packet. A bare CoAP request is not human-readable though.

Observe pattern: The observe pattern is a well-known architectural design pattern: a client declares its interest in the occurrence of a specific type of event to a server and is notified by the server when such an event occurs. In CoAP, a client can establish such an observe relation with a resource which sends a notification when its state changes. This mechanism is highly efficient, in particular compared to HTTP's

[2]To give a rough idea: According to http://serverfault.com/questions/163511/what-is-the-mandatory-information-a-http-request-header-must-contain (Sept. 25, 2013), a minimum legal HTTP request header consists of 14 bytes. According to Google, http://dev.chromium.org/spdy/spdy-whitepaper (Sept. 25, 2013), today's request headers typically vary between 200 bytes and 2 KB in the unconstrained Internet.

polling where the same GET request must be resent over and over. There exist workarounds for HTTP [28] but still "[...] generate significant complexity and/or overhead and thus are less applicable in a constrained environment." [21]

Resource discovery: CoAP defines a well-known URI `/.well-known/core` which lists the URIs to available resources on a CoAP endpoint. URIs and descriptions of resources are encoded in the Core Link Format [41] and can be requested by a GET request from a client. The server can complement URIs with further attributes that describe the resource. This mechanism allows autonomous devices and services to efficiently discover other CoAP resources in a uniform and standardized way. In contrast, the most prevalent technique for HTTP is crawling. A crawler starts at a starting point, e.g., `index.html`, and analyzes all its content to find and follow links to other resources and successively building up a resource tree. The client basically needs to download and analyze the whole content of the server to have a complete overview, making this mechanism highly inefficient.

Group communication allows a client to address multiple servers at once. This can obviously save some effort for the client and can especially be useful for discovery. CoAP features IP multicast [38]. Nodes can join a group represented by an IP address and receive messages addressed to it. The source does not even know who the destination nodes are. This is a best-effort approach and therefore unreliable. Since IP multicast is very efficient [37], it fits the constrained environment well and can in principle be implemented with existing (routing) protocols.[3] Reliability for IP multicast has been discussed [37, chapter 3.3.5.3] but later been dismissed [38]. IP multicast violates TCP's connection oriented paradigm, and is therefore not applicable for HTTP.

Deduplication: One disadvantage of CoAP is that it has to detect and filter duplicates on its own, unlike HTTP, which inherits the reliability guarantees from TCP. A CoAP server identifies a message by the pair of its source and *message identifier (MID)* and has to remember it for a specific time[4]. Under heavy load, this becomes a considerable overhead in terms of memory consumption and book-keeping effort. TCP has comparable costs for managing connections. Within a connection, however, TCP assigns a one-by-one increasing sequence number to each message, which allows for a much simpler duplication detection mechanism. Obviously, TCP's advantage fades away when the number of clients, and therefore connections, grows large as we expect in the the *Internet of Things (IoT)*.

[3]`https://www.ietf.org/proceedings/81/slides/core-11.pdf`, August 28, 2013
[4]The duration is 247 seconds for confirmable messages and 145 seconds for non-confirmable messages.

2.2 Server Specific Comparison

The executions of the two protocols CoAP and HTTP share similarities, yet have fundamental differences. To properly use knowledge about HTTP servers for an efficient CoAP server, it is important to identify such similarities and differences and outline uncertainties. Here, we only discuss the details of the two protocols in the context of a powerful server running on an unconstrained platform. Servers on constrained platforms such as a simple device have to address memory limitations or energy consumption instead.

Traditionally, HTTP has been used to access static content on a Web server. Typically, a server had to load the content from a hard disk or a cache. Workloads for such Web servers clearly were disk-bound, meaning the disk was the bottleneck and throughput and response time heavily depended on the disk and I/O performance. Furthermore, disk accesses are a source for synchronization overhead. With the emergence of CGI[5] scripts and servlet-like technologies to create dynamic Web sites, workloads shifted towards being more dependent on the computational performance of a Web server. Yet, HTTP servers still need to serve static content as well and should therefore perform well for I/O intensive workloads. HTTP benefits from the rich infrastructure the Internet provides with powerful network nodes and high-bandwidth links. For a CoAP server, the expected workload is rather uncertain. Since CoAP is used in a constrained environment where the message size is much more critical than for HTTP, it is rather uncommon that a CoAP server needs to load hundreds of KB from a disk and send them over the network. Instead, responses should be rather small and easily fit into a cache or be newly computed when requested [7].

The typical structure of steps for handling CoAP and HTTP requests look quite similar as shown in Table 2.1. In step 3, both servers need to find the exact destination of the request. If the amount of files on an HTTP server or the amount of resources on the CoAP server is small, the paths might be cached and the cache entries might even contain a ready-to-use file descriptor or pointer to the resource. Otherwise, an HTTP server might have to find the file on the disk or the CoAP server in the data structure that holds its resources. Step 4 of the HTTP server is included in step 7 of the CoAP server. In HTTP this step is useful because the client can already start to process the header while the server is copying large amounts of data from the disk to the network interface controller. The biggest difference is step 5 in which an HTTP server might read a large file from the disk if it is not a dynamically generated Web page, while the CoAP server rather loads data from memory or computes a response.

For an HTTP server thread, it might be much more important to be independent of the data fetching (loading from disk) than it is for a CoAP server thread to be independent of a resource that generates a response. An HTTP server might benefit from exporting data loading to another thread, process or, in case of unblocking I/O, the OS. While the OS

[5]http://www.w3.org/CGI/

	CoAP	**HTTP**
1	Get the datagram from the socket	Accept connection
2	Interpret the request	Interpret the request
3	Translate the path and find the resource **a)** From the cache **b)** Search in the resource tree	Translate the path and find the requested file (location) **a)** From the cache **b)** Search on the disk
4	-	Send the response header
5	Handle the request and prepare a response	Read the file to the cache (if necessary)
6	Send the response	Send the response body

Table 2.1: Structures of the processes for handling CoAP and HTTP requests.

loads data, the HTTP thread is able to compute something else. In contrast, a CoAP server might even suffer from unnecessary context switches. Let us assume, we have one thread to processes a request and generate the response and another thread that traverses through the CoAP stack and ultimately sends the request. The first thread might load the response from the main memory into the cache or newly compute it. Either way, since the other thread might be running on another core, not only do we need a context switch but also does the response data need to be transferred to the other core's cache. Both penalties can be avoided if the same thread processes the request and sends the response back.

3 Server Architectures

Concurrent servers have been subject to research for more than fifteen years, in particular HTTP Web servers. Originally, concurrent handling of requests was introduced to allow servers to accept connections form multiple clients at the same time. The goal was to better utilize the CPU during I/O operations but was not intended to scale servers over multiple cores. At the latest with the advent of HTTP/1.1 in 1999, clients were allowed to keep connections alive for multiple consecutive requests. A typical Web page contained several links to images, flash content and other pages. Instead of repeatedly connecting to the server for each link, a temporary connection allowed sending multiple consecutive requests, and therefore avoiding many 3-way handshakes and TCP slow-starts. In subsequent years, engineers came up with different server architectures that attempt to increase server efficiency by mitigating bottlenecks such as synchronization. When servers were equipped with multiple cores, the very same techniques of parallelizing concurrent computations were used to utilize their multiplied computational capacity. In this chapter, we present the most significant server architectures found in the literature:

1. *Multi-Process (MP)*
2. *Multi-Threaded (MT)*
3. *Single-Process Event-Driven (SPED)*
4. *Asynchronous Multi-Process Event-Driven (AMPED)* (1999)
5. *Staged Event-Driven Architecure (SEDA)* (2001)
6. *Multi-Threaded Pipelined (PIPELINED)* (2005)

Note that MP, MT, SPED, and AMPED originally have been proposed for single-core systems.

3.1 Multi-Process

The Multi-Process (MP) architecture distributes the workload over multiple processes. Most operating systems support multiple processes and make use of a sophisticated strategy to schedule processes on one or more cores to optimally utilize their computational power and guarantee fairness (especially the absence of starvation). Multiple processes are rarely able to share global information such as a cache, though.[1] When a client establishes a

[1]Unix systems provide the `mmap` operation that allows sharing memory among processes.

new TCP connection, the server forks a new process which processes all messages for that connection. The MP architecture is shown in Figure 3.1.

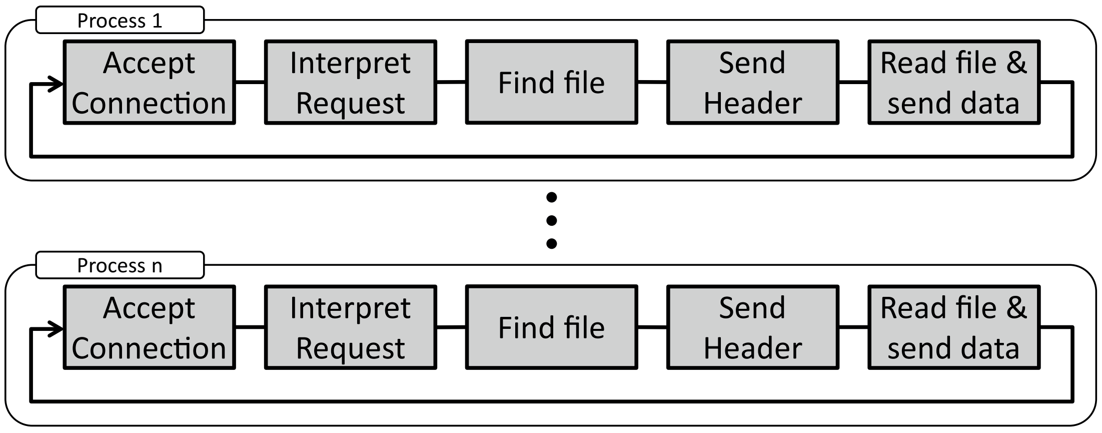


Figure 3.1: Multi-process architecture: Each process processes all protocol steps.

3.2 Multi-Threaded

The Multi-Threaded (MT) architecture distributes the workload over multiple threads with a shared memory address space as shown in Figure 3.2. It is crucial that the underlying operating system supports kernel-threads. Since an application manages user-level threads in user-space, they cannot be distributed over multiple cores. Worse, since user-level thread scheduling is a form of cooperative scheduling, the kernel's interrupt-handler cannot preempt a user-level thread that invokes a blocking I/O operation and schedule another user-level thread. The I/O operation blocks the whole application.

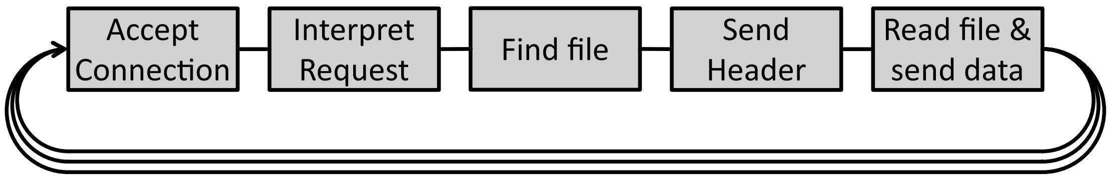


Figure 3.2: Multi-threaded architecture: Each thread processes all protocol steps.

Threads are able to access a shared cache concurrently. Modern programming languages provide powerful synchronization tools and there exist whole libraries for efficient caching. Yet, the synchronization of a large number of threads can lead to a significant overhead. In a typical multi-threaded HTTP server, a welcome socket accepts new TCP connections, creates a new socket and assigns a thread to it, thus, there is one thread per connection. Usually, a thread closes its connection after a timeout has expired with no events on the socket. As a result, there are two main factors that influence the server performance: the amount of available threads and the connection timeout. Beltran et al. [4] have found to get the best performance for a server when setting the amount of threads to the amount of expected concurrent connections and the timeout to infinity. Obviously, the amount of concurrent connections is hard to estimate but can be adjusted at runtime with some effort.

A common disadvantage of MP and MT is that a large number of processes or threads is required to support a large number of concurrent clients. If the allowed number of processes (threads) is limited and all processes (threads) assigned, no further connections can be accepted until a process (thread) closes its connection and is ready for a new one. Creating and terminating a process (thread) is expensive. Therefore, most implementations use a pool of reusable processes (threads). Processes (threads) allocate a considerable amount of memory for their state and especially their stack. On a 64-bit Java Virtual Machine, the default stack size of a thread is 1 MB. [2] Nonetheless, MP and MT have been the architectures of choice for Apache's HTTP server at the end of the 1990ies and remained prevalent up to the present day.

The large number of threads is MT's greatest weakness and all the following architectures specifically try to avoid it. Notice that this is different for a CoAP server because there are no connections. In CoAP, the requests of all clients arrive at the same socket and no thread is blocked by an idle but still open connection. An MT solution for CoAP could concurrently serve many clients with only few threads.

3.3 Single-Process Event-Driven

The Single-Process Event-Driven (SPED) architecture splits up the message handling process into small tasks. Tasks use non-blocking I/O operations. When an event is triggered (e.g., a new connection arrives, a file operation completes or a client socket has received data or has space in its send buffer) a resulting task is added to the event queue. A single event dispatching thread keeps popping tasks from the queue and executes them one after another as shown in Figure 3.3. SPED is able to parallelize CPU, disk, and network operations despite having only one thread. Since there is only one thread, no synchronization is required, context switches can be saved, and data are always cache-local. In return, SPED cannot benefit from multiple cores and, unfortunately, many operating systems do not provide suitable support for non-blocking operations [30]. However, when the new 1.4 release of the J2SE introduced the NIO (New I/O) API, Beltran et al. showed in 2004 that an event-driven Web server written in Java and using NIO "[...] scales as well as the best of the commercial native-compiled Web server, at a fraction of its complexity and using only one or two worker threads" [3]. In 2007 Pariag et al. proposed an extension to SPED called Symmetric Multi-Processor Event Driven (SYMPED) which forks multiple SPED processes [33]. Whenever a (SPED) process is stuck in a blocking I/O operation, the OS switches to a process that is able to run.

[2] `http://www.oracle.com/technetwork/java/hotspotfaq-138619.html`, July 02, 2013

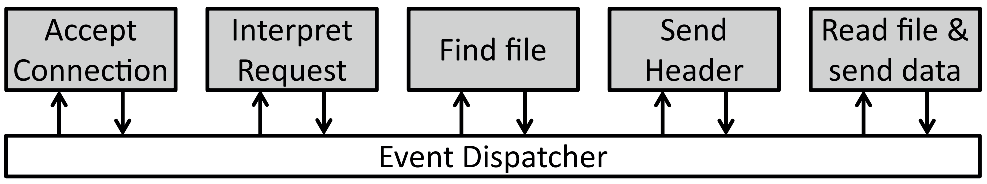


Figure 3.3: Single-threaded event-driven architecture: A single thread jumps from task to task and uses non-blocking I/O operations.

3.4 Asynchronous Multi-Process Event-Driven

The Asynchronous Multi-Process Event-Driven (AMPED) [30] architecture uses, similarly to SPED, a single dispatching thread as shown in Figure 3.4. The dispatcher, however, only serves cache-hit requests. If there is a cache-miss, the dispatcher forwards the request to a helper process (or thread) which fetches the data. Basically, AMPED wraps blocking I/O operations in a separate process (or thread) to make them asynchronous. In 1999 (still the age of unicore processors), Vivek at al. implemented AMPED in a server called Flash [30]. In 2000, PalChaudhuri et al. released a Co-AMPED version of Flash for multiprocessors that used one AMPED process per core and outperformed Apache's MP implementation [32].

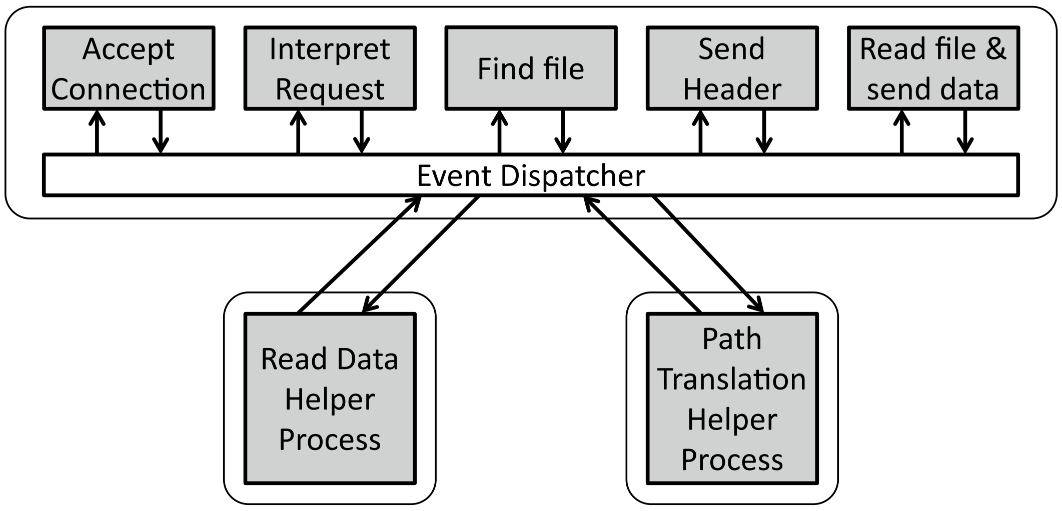


Figure 3.4: Asynchronous Multi-Process Event-Driven architecture: Outsources blocking operations to separate processes.

3.5 Staged Event-Driven Architecture

The Staged Event-Driven Architecture (SEDA) [47] splits the message handling process into multiple stages as shown in Figure 3.5. For instance, one stage is responsible for interpreting a request and another one for loading data from the disk. Each stage consists of an incoming event queue, a thread pool, and an event handler. The event handler can be thought of as a function that executes the logic of the stage. The threads pull events from the event queue and invoke the event handler which can forward new events to a next stage. A stage is managed by a controller that can dynamically allocate new threads according to a policy. Therefore, each stage of a SEDA server can self-tune itself to have the optimal number of threads to exploit the benefits of parallel execution without the drawbacks of a too large number of threads.

Welsh et al. invented SEDA and implemented an HTTP server called Haboob. Haboob easily outperformed MP Apache and AMPED Flash [47]. Unfortunately, they compared SEDA only to Apache's MP version but not MT. In 2002, Larus at al. showed that a SEDA server can even further increase its throughput when threads not only pull one but a batch of events from the event queue at once and therefore improve cache-locality [27]. However, in 2003, von Beren et al. suggested using compiler support to improve synchronization and memory stack management of MT servers and presented results that outperformed Haboob once again [46].

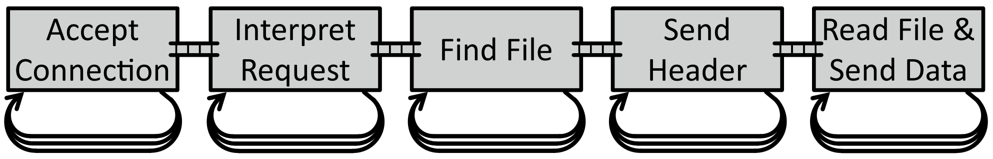


Figure 3.5: Staged Event-Driven Architecture: Each protocol step is a stage. Each has its own thread-pool and forwards processed messages to the next stage over a queue.

3.6 Multi-Threaded Pipelined

A chain of stages can be seen as a pipeline. In contrast to SEDA, Multi-Threaded Pipelined (PIPELINED) only has one thread per stage. Within a pipeline one thread forwards results to the next thread. A PIPELINED server creates one pipeline per core as shown in Figure 3.6. Additionally, a pipeline can use helper threads to which it outsources blocking I/O in case of cache-misses, similar to AMPED [12]. In 2005, Choi et al. have shown that MT and PIPELINED outperformed all other architectures in terms of memory usage and throughput [12]. In 2007, Pariag et al. showed that SYMPED (see above) and PIPELINED can achieve an 18%higher throughput than MT [33].

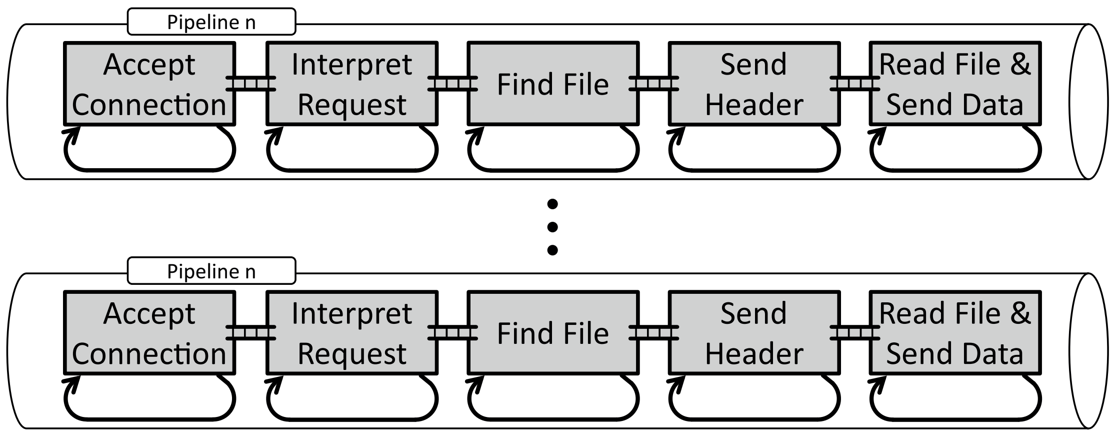


Figure 3.6: Pipelined architecture: The protocol steps form a pipeline where each step is processed by one thread. Blocking operations can be outsourced to additional threads (not in the image).

3.7 Comparison

The main difference between SEDA and PIPELINED is that a thread in SEDA forwards an event to any thread of a next stage where a thread in PIPELINED always forwards an event to the same thread in the next stage. An advantage of SEDA and PIPELINED over MT is that fewer threads are necessary to handle a large amount of parallel connections, and therefore reduce synchronization overhead. However, they both need to synchronize threads between each stage. Furthermore, the time a request spends in the system can increase due to traveling from stage to stage. Each stage change for a request means a thread change which implies a context switch and a potential loss of cache-locality. This can increase the latency a client experiences. In MP and MT, there should be much fewer context switches while a request is processed. In 2010, M. Welsh, the author of SEDA, reviewed his design and wrote in his blog: “Most stages should be connected via direct function call.” [3] This basically means to merge two stages together, since too many stages are just not worth it.

Genevès compared the MT architecture to AMPED and SEDA on an eight core machine [19]. When varying incoming connections per second, all models reached the peak throughput simultaneously at 9000 connections per second. Eight cores achieved only 2.3 times the performance of a single core and the difference between four and eight cores was only marginal. He found that with eight cores, the memory bus between cores was totally overloaded, allowing no further performance improvement when using even more cores. Beltran et al. showed in 2008 that a hybrid Tomcat server [10] that combined MT and SPED was able to outperform MT Tomcat on CPU-bound workloads if the number of concurrent clients exceed 2000 (but they perform equally up to that point) [4]. In 2012,

[3] `http://matt-welsh.blogspot.ch/2010/07/retrospective-on-seda.html`, August 02, 2013

Harji et al. extensively benchmarked multiple Web servers on quad-core SMP systems and concluded "[. . .] that implementation and tuning of Web servers is perhaps more important than server architectures." [20]

In summary, there exist a plethora of models, implementations, workloads, and experiments. Results depend heavily on the underlying platform (caching protocol, non-blocking I/O support, etc.) and used workloads (disk-bound, memory-bound, computation-bound, small/large files, cache-hit rate, etc.). Results of different papers are often contradictory. Our evaluation in Chapter 6 is going to show, whether Californium exhibit similar behavior as their HTTP cousins and whether the simpler CoAP protocol allows better scalability than HTTP.

3.8 Alternative Scalability Strategies

Using multiple processors and cores in one machine is a form of vertical scaling. Papers that suggest horizontal scaling, i.e., using multiple machines with load balancing, have been published as well [29]. Since horizontal scaling is out of the scope of this thesis and CoAP application workloads are not expected to be disk-bound, these approaches are less important for us. On the other side of the spectrum, the HTTP protocol has been translated into a finite state machine to ease HTTP integration into constrained devices [13]. There have been many further approaches to increase Web server performance. Since the main purpose of most HTTP servers is to only serve HTML Web pages or media files to clients and therefore need to read a lot of data from the hard disk, engineers have presented approaches to boost disk-bound workloads. For instance, the server scans the HTML files served to a client for links to local files, e.g., images, and starts to preload these files into the cache as it is to be expected that the client will request them right away [1]. Some approaches focus on dynamic content. In 2009, Hop Web Server [40] was proposed, a server that compiled third-party software at runtime and merged it into the server's runtime environment. Hob Web Server outperforms Web servers that rely on CGI or FastCGI protocols.

4 Design

In this chapter, we discuss our proposal for the architecture of our CoAP server, the re-implementation of Californium (Cf). We first highlight the most relevant goals that have influenced our design decisions. Second, we describe how we structure the elements that make up CoAP and how they exchange information. Finally, we propose a concurrency model that specifies how the elements are parallelized.

4.1 Design Goals

In the field of software engineering, the list of expressions for software qualities is long. ISO/IEC 25010:2011 [1] alone defines 40 of them—from "accessibility" to "user-interface aesthetics." Some of them are rather obvious, e.g., correctness; others are synonyms or at least do overlap, e.g., maintainability, extensibility, and adaptability. This section gives a non-exhaustive list of non-obvious goals and explains why we believe them to be relevant for Californium.

4.1.1 Completeness

Californium enables an application to appear as both CoAP client and CoAP server. First, it enables the application to send CoAP requests to CoAP resources and receive the response. Second, as server, the application is able to export CoAP resources which process incoming CoAP requests and respond to them. Since Cf is able to not only receive but also send requests, it can also serve as intermediary such as a caching proxy. We identify the following three main high-level workflows in Cf:

1. Construct a server and add resources. The server should be able to be started and stopped.
2. Receive a request, find its target resource, let the resource process the request and send the response back to the client.
3. Issue requests as client and wait for responses to allow for proxies or more complex services that use and aggregate Web services themselves.

[1] `https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en`, August 22, 2013

Californium must be designed so that these workflows can be implemented.

4.1.2 Scalability

A traditional indicator for server performance is throughput, i.e., processed requests per second. It is not our prior goal to achieve a throughput as high as an optimized C library for CoAP. Instead, we favor vertical scalability when multiple processor cores are available. Cf's architecture should allow the whole processing chain to be efficiently distributed over multiple cores. To this end, we will need a concurrency model that specifies which and how processes are parallelized and synchronized. Concurrency is often a source for race conditions and causes the need for synchronization. Overhead due to synchronization among parallel processes is the nemesis of scalability. Therefore, our goal is to introduce as little synchronization as possible but as much as required for satisfying the CoAP protocol.

4.1.3 Flexibility

It is clear that there is no silver bullet that scores highest in all aspects. For instance, synchronization that comes with multi-threading clearly is an adversary to understandability. In general, optimizations often come in form of shortcuts that lessen flexibility, extensibility, and in particular maintainability. Typically, a design decision affects different machines and applications much differently. Therefore, our goal is to keep Cf as flexible as possible and give the programmer instruments to modify Cf's behavior as much as possible.

4.1.4 Usability

Californium should be easy to learn to use. Developers should be able to compose services that export their functionality in resources, build up resource trees, and connect servers to the network. It should be simple to synchronously or asynchronously send requests and react to responses or failures. It should be easy for the developer or even the end-user to dynamically configure the parameters of Californium. Therefore, we need a well-designed API that is simple and powerful.

4.1.5 Understandability

Even though the core of CoAP is on the verge of becoming an RFC, CoAP is still subject to extensions. Cf's design should accommodate for that. It is natural to understand CoAP as a combination of multiple semantic components such as reliability of transactions, deduplication of messages, blockwise transfer and so on.[2] These components are not necessarily independent of each other. For instance, when a confirmable notification of an observed resource fails to be transmitted to the client and timeouts, the observe relation should be canceled. Furthermore, blockwise transfer splits up the exchange of a single request and a single response into an exchange of a potentially much larger amount of requests, responses, and acknowledgements. Some components require information whether a given message is the full message or only a block of it.

The proposed architecture should define the place of these separate conceptual components in the system and how they interact so that the required dependencies can be satisfied. With such an architecture at hand, we can reason about whether the system satisfies the semantics of CoAP and the understanding of the system eases maintenance and leverages further extensions. Therefore, the placement of these components and the way they exchange information must be chosen carefully. The more logical and simpler CoAP's conceptual components are connected, the easier a developer, who looks into the code, can understand the system and the less prone to errors it becomes.

4.2 Architecture

We propose to logically split up a CoAP server into three stages: resources, endpoints, and connectors. Figure 4.1 visualizes how these three stages build up a Cf server. A program that appears only as client would only operate with endpoint and connector. We explain these three stages in the following sections.

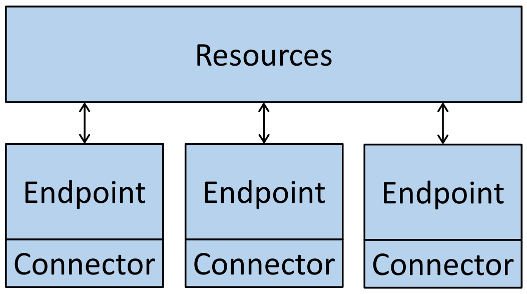


Figure 4.1: We split up Californium into three stages. The server has a resource tree, one or more endpoints and each endpoint has exactly one connector.

[2]Unfortunately, not only the environment CoAP is meant for but also our human brain has limitations so that we are unable to grasp the complete CoAP protocol as one piece but require dividing it into simple human-brain-understandable concepts.

4.2.1 Resource

Resources are the main building blocks that the developer is going to use to export the functionality of its application over CoAP to the world. A CoAP resource provides a RESTful interface to clients. It makes itself accessible and modifiable by reacting to requests that carry one of the four request codes defined in CoAP: GET, POST, PUT, or DELETE. Each server holds a tree structure where each node is a resource. Each resource is identified by a URI that is composed of the URI of its parent plus its own name. When a request arrives at the server, it searches the resource tree for a resource that corresponds to the destination URI of the request. If the server finds the resource, the resource processes the request and responds with an adequate response code, options, and payload according to the CoAP protocol. If the server cannot find the destination resource, it responds with a 4.04 (Not Found) error code.

4.2.2 Endpoint

An endpoint wraps the implementation of the CoAP protocol, i.e., it contains the processing chain that processes incoming and outgoing CoAP messages. A server has one or more endpoints that are connected to its resource tree. When a CoAP request arrives, the endpoint processes the required steps according to the protocol, e.g., decoding the datagram into a request-object, duplication detection, etc. Finally, the endpoint forwards the request to the resource tree, which will handle it and respond with a response over the same endpoint. An application can also use the endpoint to send requests to another CoAP server and will later receive the response.

As described in section 4.1.5, we understand CoAP as a composition of conceptual components; some of which depend on information from each other and some do not. The structure in which we place these components has to satisfy all these dependencies. We identify the following conceptual components:

1. Token management to generate an ID for new requests.
2. Observe relations from clients to a resource.
3. Blockwise transfer to split up the transmission of one large message into multiple smaller ones.
4. Transfer reliability through retransmitting unanswered confirmable messages.

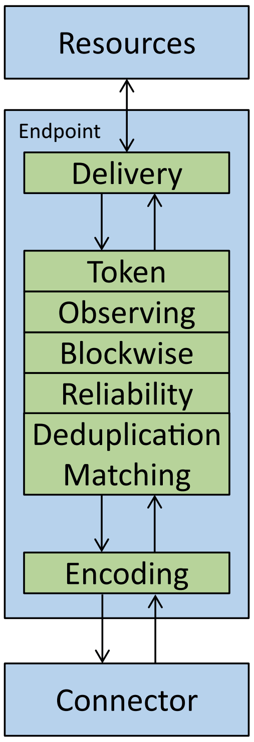


Figure 4.2: Structure of the CoAP stack in an endpoint. Each component (green) is implemented as a layer.

5. Matching of incoming responses, acknowledgements, and resets to former outgoing messages and duplicate detection.

6. Encoding messages to byte strings and decoding byte strings into an internal representation of a message.

Notice that not all these components can be properly associated to one of the two logical sublayers [42, p. 10] that CoAP describes (request/response layer and message layer). The layer for blockwise transfer, for instance, needs to receive acknowledgements (part of message layer) and responses (part of request/response layer) to initiate the transmission of a next block.

In the first version of Californium (OCf), it was proposed to implement these components as layers stacked upon each other and using a common interface to forward messages upwards through the stack as well as downwards. The main advantage of such a layered structure is understandability and extensibility. The resulting workflow is linear by design and the order in which these conceptual components are invoked is strict for all messages. This is simple and better understandable than a finite state machine, for example. We have adopted this design from OCf and will refine its implementation in Chapter 5 to achieve even better understandability, maintainability, and support for concurrency. The stack of layers that Californium uses is shown in Figure 4.2.

Using a stack of layers leads to a few implications: First, the order in which messages are processed is static and strict, not dynamically modifiable. When sending a message, the order in which it is processed by the components is exactly the reverse of when the message is being received. Furthermore, forcing the components to implement a common interface and letting them only call the layer above and below using only methods defined in this interface is a restriction. The interface for a layer must be expressive enough. These restrictions might not be reasonably applicable to any protocol. Our implementation, however, shows that we can implement CoAP in spite of these restrictions and benefit from its simplicity.

4.2.3 Connector

A connector abstracts how the server sends and receives messages, i.e., the transport protocol. A connector has no knowledge about the format of CoAP messages. Instead, an endpoint passes encoded messages as opaque bit strings together with the destination address and port to the connector to send it over the network. When a connector receives a message in form of a bit string, it passes it to the endpoint together with the source address and port. Typically, the connector uses the UDP protocol and sends and receives datagrams over a socket. Other connectors might use *Datagram Transport Layer Security (DTLS)*, TCP or any other transport protocol. For example, we use some modified connectors in our JUnit tests to circumvent the OS and network and neglect undesirable delays by forwarding a message from a client directly to the server.

A connector should not be confused with a socket. An endpoint does not query the connector for receiving another message. Instead the connector calls its associated endpoint and hands over new messages. Vice versa, an endpoint hands outgoing messages over to the connector. The methods of the connector should be non-blocking and the connector has its own concurrency-policy. Rich functionality can be implemented in a connector. Scandium[3], for example, is a DTLS implementation for secure communication between endpoints.

4.3 Concurrency Model

One of the main goals of Cf is scalability over multiple cores of a machine. Therefore, Cf's architecture has to define which of its elements can be parallelized. A parallel element defines its own concurrency model, e.g., whether it uses a single thread or a thread pool to execute its code. An element could even implement an event-driven or multi-process concurrency model but for our implementation, we have not found these models useful. One encounters different conflicting factors when parallelizing the elements of a server. Naively, the more elements run in parallel, the more can they exploit large numbers of cores and achieve better performance. In practice this hardly holds true. In fact, the information flow between two parallelized elements becomes much more costly. Making a caller thread of a method also executing the logic behind that method is straight forward in programming, i.e., a normal method call. Such method calls are synchronous by design and easy to reason about. If the callee element is supposed to run concurrently to the caller element though, and therefore the caller thread has to get a thread of the callee element to process the logic behind the method, the effort for such a method call significantly increases. First of all, a context switch is required before the callee thread can execute the method's logic. Second, the callee thread might run on another core than the caller thread so that the memory of the method parameters is not in the cache of the core. Finally, to pass parameters from one thread to another, one typically uses a shared queue to which the caller writes and from which the callee reads and which causes further synchronization overhead. As a result, the total amount of computation and latency per message increases and the performance could decline. There are further issues such as the impact of a large number of threads to the system, e.g., in terms of additional memory consumption. Finally, more fine-grained parallelism leads to higher complexity, potential race conditions and hampers maintainability.

One advantage of parallelization is that it allows protecting the system from failures in elements that are not relevant to the whole system. Assume, for example, a resource that spends five seconds' time in a critical section to respond to requests. If the threads

[3]`https://github.com/mkovatsc/Scandium`, October 12, 2013

that process requests for other resources also invoke the handler of this ‘slow’ resource, they might all get stuck waiting to enter the critical section. In this case, a broken resource breaks the whole system. We can solve this issue by modeling a ‘slow’ resource as an independent element with its own thread. The threads that process requests in the endpoint-stage no longer invoke the handler but pass the requests to the specified thread of the resource. This makes the resource non-blocking and congesting requests will no longer delay the threads of the remaining system.

As stated above, a parallel element defines its own concurrency model, e.g., the number of threads it uses to execute the programming code. Using only a single thread for all computations frees the element internally from concurrency issues such as race conditions. Using a thread pool allows to run independent computations concurrently, e.g., processing independent requests. The number of threads can be chosen at runtime and even adjusted dynamically. Depending on the application, it might be advantageous to prioritize certain elements, e.g., certain resources or endpoints, to find an optimal balance. The server behavior becomes more flexible. Furthermore, we can optimize the server for a specific platform it runs on, for instance depending on the number of cores it provides. On a quad-core machine running Windows, we have measured that using four threads, we can receive almost twice as many request per second through a UDP socket than if we use only one thread. On a 16-core NUMA Red Hat Linux system, we found quite the opposite, having a 40% lesser throughput using two threads instead of one.

Our concurrency design opts for finding the optimal tradeoff among these advantages and disadvantages. The most coarse-grained design would be to use one single concurrency model for the whole server. The most fine-grained design defines a separate concurrency model per object, e.g., each CoAP component of an endpoint defines its own threads. We believe the optimum to be between these extremes. We propose to let each connector, each endpoint, and each resource choose its own concurrency model. That means, each of our three defined stages has its own concurrency model, similar to SEDA. We have chosen a stage-based model and subdivided the server into the aforementioned stages for the following reasons.

Since our benchmarks have shown that on certain platforms one thread instead of multiples can receive more packets per second through a socket, such a thread should be burdened with as less different code as possible to achieve the best performance. Since a connector defines its own concurrency model, the number of threads reading from and writing to a socket can be configured independently from the rest of the server.

The main advantage for giving each endpoint its own concurrency model is that it allows us to make them independent from the resource-stage. Resources that block threads compromise the whole system, and therefore, giving the developer instruments to protect endpoints from ‘slow’ resources is important. The endpoint consists of the processing chain of the CoAP protocol. Each incoming or outgoing message is processed by one of its components after another. Although it is possible to parallelize the components of an endpoint as well, it introduces too many context switches to be efficient. Each component is relevant for the whole endpoint, and therefore the balancing argument does not apply.

Resources are imaginable in many flavors. A resource could be very simple in responding to requests but just as well wrap a complex procedure. In particular, some resources might be able to process requests concurrently while others consist of critical sections. Orthogonally, the potential effort to process a given request ranges from a few instructions up to several seconds' time of computation. Since each resource is able to define its own concurrency model, i.e., thread pool, with an arbitrary number of threads that process requests, the developer has high flexibility. If a resource does not define a thread pool, the thread pool of its parent or transitively its ancestor will be used. If no resource on the path to the root defines a thread pool, the same thread that has processed the CoAP protocol will also process the code of the resource. In this case, there is no context switch for passing the request to the thread pool which benefits performance. If the request processing code of a resource contains critical sections, there are two simple ways how to guarantee mutual exclusion. First, the developer can choose a resource with a single-threaded pool. As

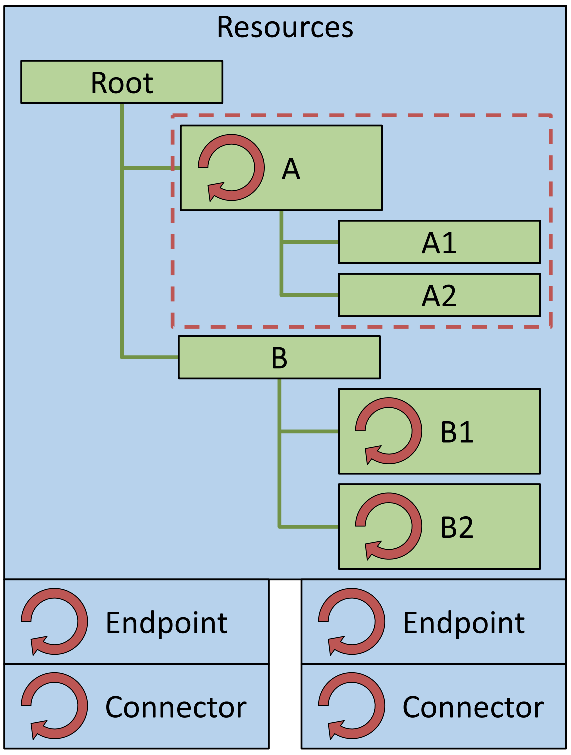


Figure 4.3: Each Connector, endpoint and, resource (green) is able to define its own concurrency model (represented by a rotating, red arrow). In this example, requests to A, A1, and A2 will be all handled by A's thread pool. Requests to B1 and B2 will be handled by their respective thread pools. Requests to B and the root will be handled by the thread from the endpoint that forwards the request to the resource.

a result, all requests for the resource will be processed by the one and only thread and concurrency is not an issue in the first place. Second, the developer can simply add the keyword ‘synchronized’ to the method declaration that processes requests or to the critical section. Figure 4.3 gives an example of a server with two endpoints and a resource tree where resources A, B1, and B2 define their own concurrency model. Requests to these resources or their children will be processed by their respective thread pool.

The staged architecture can be seen as a combination of several architectures from Chapter 3. Each stage can create its own thread pool which is what the MT model postulates. Threads have to wait until a message comes through the queue, which makes it event-driven as well. Therefore, expertise from these architectures also improves the staged architecture. Since we use Java, any multi-process architecture would not be appropriate. We also do not consider the PIPELINED architecture applicable for a CoAP servers. The CoAP protocol processing chain has components that cannot simply be replicated; the replicas had to communicate and synchronize their state, e.g., the component for duplicate detection. Since an HTTP server benefits form TCP’s reliability guarantees, it is more eligible for such an architecture. Typically, the OS provides a TCP implementation, and therefore already solves the problem how to achieve these guarantees. CoAP, on the other hand, has to do it in the application layer, which the server architecture has to account for.

5 Implementation

In the previous chapter, we have explained the three-stage architecture of Californium and how we structure the CoAP processing chain as a stack of layers within an endpoint. In this chapter, we give a more detailed discussion, how Californium implements these layers and how they communicate with each other.

5.1 Messages

A CoAP message is a collection of values: message type, message identifier (MID), message code, token, options, and payload. These values are defined in the CoAP specification [42]. The MID is needed to match an acknowledgement to a message and the token is needed to match a response to the origin request. It is impractical to keep messages encoded in the binary format CoAP describes. Dealing with bit strings is cumbersome and error prone. Since Cf is written in Java, we also opt for an object-oriented model for messages. With an appropriate internal representation not only have we convenient object-oriented methods to read and modify a message's values but also benefit from type-checking at compile time. High-level programming languages provide modeling instruments such as subtyping and inheritance. We do not want to over-engineer our class-model, though, and make only carful use of such instruments.

One of the most important values of a message is its message type: confirmable (CON), non-confirmable (NON), acknowledgement (ACK) or reset (RST). The message code defines whether a message carries a request, a response or is empty. Request codes are further divided into GET, POST, PUT, and DELETE while response codes are divided into the three classes Success, Client Error, and Server Error which again are divided into multiple status codes. Depending on the type of the message and whether it carries a request or response, different other properties can hold, e.g., we can only respond with a status code 2.05 (Content) to a GET but no other requests. However, many properties still hold for most messages. We believe it is worth only implementing the code for these properties once in a class and let all messages use it either due to inheritance (for example *MID* and token) or due to having an object representing them (for example options). Table 5.1 shows what message type can carry a request or response. Multiple reasonable division strategies for messages are imaginable.

	CON	NON	ACK	RST
Request	Yes	Yes	No	No
Response	Yes	Yes	Yes	No
Empty	Yes[1]	No	Yes	Yes

Table 5.1: Not all message types and message codes can be legally combined.

No division: We might just use one class to represent them all. However, we feel this design is too coarse. We believe that different classes can hint the developer to what actions and properties belong to a message and ultimately improve the developer's understanding of Californium.

Division according to message type: This allows giving different methods to messages according to their message type. Confirmable messages would provide acknowledge() and reject(), non-confirmables only reject() and empty messages neither. However, most processing parts in CoAP depend more on whether a message carries a request or a response than what message type it has. The most inconvenient drawback of this division is that it requires a lot of confusing looking code for checking what a message actually is. We believe that in general, the CoAP message layer is of little interest to the application developer and prefer keeping it transparent in our API.

Division according to message code: We believe that it is most natural for the developer to think in terms of requests and responses about CoAP and our goal is to hide as much complexity as possible. We observe that a lot of code for the different message types is unitary while code for requests, responses, and empty messages often differ. Therefore, we propose to division all messages into three classes for requests, responses, and empty messages. Values that belong to all messages independent of their code (for example *MID* and token) are collected into a common superclass `Message` as shown in Figure 5.1. Methods that depend only on the message type, e.g., retransmission depends only whether it is CON but not whether it is a request or response, can work with the common supertype `Message` so that no duplicate code must be written.

We could even further division requests into one class per request code or the responses into one class per response code. This, however, yields a large amount of classes with little to no difference. Therefore, we consider this over-engineering and decide against it.

[1]Can be used to trigger an RST from an endpoint to test if it is reachable

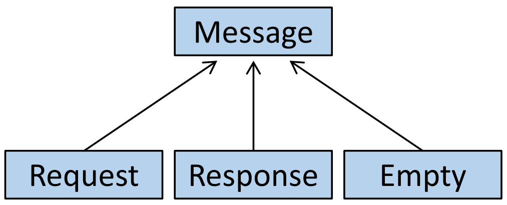


Figure 5.1: Message model: the classes for requests, responses, and empty messages have a common superclass.

5.2 Exchange

We call the process of sending a request and receiving one or more corresponding responses and empty messages an exchange. In its simplest case, an exchange consists only of one request and response. If a request must be split up into multiple blocks each block belongs to a single exchange stemming from the origin request. The same applies to all responses. This section explains how the state of an exchange can become complicated and our solution how Cf keeps track of messages and manages its state of exchanges. In CoAP, The reasons for most complexity are the three following CoAP extensions that lead to multiple responses.

a) A response is split up into multiple blocks due to its large size (block-12 [8]).
b) In case of an observe relation, we receive multiple notifications (observe-08 [21]).
c) In case of a multicast request, we receive responses from multiple endpoints (groupcomm-16 [38]).

Reasons a) and b) can occur within the same exchange but should not at the same time. Blockwise transferred notification should be transferred one after another. Reasons a) and c) could but should not occur at the same time. Endpoints should not respond with a blockwise transfer to a multicast request. The same is true for multicast requests because they must be sent non-confirmable while a blockwise transfer requires confirmable messages. Finally, reasons b) and c) can never occur at the same time. “If multiple subjects are of interest to an observer, the observer must register separately for all of them” [21, p. 3].

The state of an exchange seen by a client and seen by the server is not always the same but must be synchronized to eventually reach consensus. On first view, CoAP may seem to be a protocol where always only one participant triggers a new action but this is not true. A timeout at an endpoint, for instance, might cause it to resend a message, and therefore an endpoint must expect to receive duplicate messages that belong to the same exchange and only process one of them. A client application might just cancel a request in the middle of sending it blockwise. An observed resource on a server might cancel a notification that is being transferred blockwise and send a new one. A cancelation of a large message must be propagated to its blocks and must be recognized by both endpoints so that both can clean up their state. As a result, the state of an exchange can be modified concurrently due to numerous events. Endpoints must be able to deal with an arguably complex space of possible states that an exchange might be in. If an endpoint uses only a single thread to modify its state of a specific exchange the concurrently triggered events automatically linearize and the program code becomes easier. In a multi-threaded environment, however, judicious use of synchronization is required.

The implementation of OCf has shown one major flaw: the state the endpoint sees of an exchange is distributed over all layers. Each layer has one or more hash maps that contain mappings of tokens to the partial state of exchanges that the layer requires. The layer for blockwise transfer, for instance, uses an inner class to hold the current status of the blockwise transfer. Whenever a block arrives, the layer finds the corresponding blockwise transfer status in its hash map and uses is for its next actions. Drawbacks of this approach are

Difficult to clean up: When a program cancels a request, all layers must realize that they have to remove the corresponding state they hold.

Memory usage: Each entry needs a very small amount of memory. Under heavy load with many exchanges, the memory nevertheless becomes a problem and using more hash maps than necessary is wasteful.

Efficiency: Finding a mapping of a token to its state is fast. However, doing it more often than necessary is wasteful.

Scalability: A hash map that maps tokens to state is a source for synchronization overhead when multiple threads write to it.

Atomicity: There exist concurrent hash maps that provide powerful APIs that can bundle multiple operations to one atomic operation. However, multiple atomic operations on different hash maps are not atomic. In case multiple threads operate on the hash maps additional synchronization is required.

Note that the last two reasons were not really drawbacks for OCf because it was meant as single-threaded system.

We propose to collect all state an endpoint knows of an exchange within a single object. We call this object an Exchange (with capital E). Whenever a client intents to send a request, it first creates an Exchange. When the request traverses downwards through the CoAP stack, each layer updates the Exchange and forwards it to the lower layer. On the other hand, when the request arrives at the server, it creates an Exchange to keep the state and each CoAP layer forwards the Exchange upwards through the CoAP stack. After the target resource has computed the response, it traverses downwards through the CoAP stack of the server together with the Exchange. When the response arrives at the client, the client finds its Exchange that was created for the request and forwards the response plus Exchange upwards through its CoAP stack. Since each layer always receives not only a single message but also the Exchange with the state that corresponds to that message, they do not need their own hash maps with mappings to the state they require. When an Exchange has completed—be it successfully or not—we can remove the Exchange and all state is automatically cleaned up.

5.3 Layer Interface

The layer interface defines the methods each layer must provide to its neighbor. The interface of OCf defines two: `send(Message)` and `receive(Message)`. Since a layer should process both incoming and outgoing messages, it needs to know the direction of a message, and therefore needs a method for sending and for receiving respectively. Both methods expect an object of type `Message` as parameter. However, some layers process requests differently from responses or empty messages. Since the type `Message` is the superclass of the three, each such layer needs to test the parameter type with `instanceof` and cast it to its actual type to proceed correctly. The underlying problem is that type information gets lost when calling a method that expects an object of type `Message` but actually passing it an object of a subtype. Many consider the usage of `instanceof` as a hint for bad design and recommend "to only us it as a last resort."[2] We propose to split up methods that process a message into multiple methods that each expects a parameter of a subtype respectively. Furthermore, each layer expects the Exchange that belongs to the specified message. This gives us a total amount of six methods:

```
send(Request, Exchange)
send(Response, Exchange)
send(EmptyMessage, Exchange)
receive(Request, Exchange)
receive(Response, Exchange)
receive(EmptyMessage, Exchange)
```

Note that we cannot omit a method for empty messages because some layers such as for blockwise transfer or a future CoAP extension might depend on it.

It might look tempting to reduce the two parameters to only one. Say the method `send(Response, Exchange)` only expects an Exchange and if necessary extracts the response that is to be sent from that Exchange. Unfortunately, this does not work in a multi-threaded environment (where multiple threads concurrently access the same Exchange). The reasons are again the usual suspects: blockwise transfer, observe relations, and multicasts. In the current draft for blockwise transfer (block-12 [8]), the first and second blocks of a large response are sent at the same time. An observed resource might send

[2] `http://www.javapractices.com/topic/TopicAction.do?Id=31`, June 07, 2013

multiple notifications in a short time interval. Multiple endpoints might respond to a multicast request. As a result, multiple responses to the same request and ultimately the same Exchange might arrive at the client. In our multi-threaded system, multiple threads can process the responses and modify the Exchange concurrently. Each thread must know which response it is supposed to process. If the method `receive(Response, Exchange)` only expected an Exchange, a thread would have no way of knowing which response it is supposed to process even if they were all somehow stored in the Exchange. One possible solution is to somehow map a thread id to the response it is supposed to process but this is just another source for synchronization overhead and an arguably ugly way to do it. We find it more natural and reasonable to add the message and the Exchange to the expected parameters.

5.4 Synchronization

An endpoint must be prepared to receive the same message multiple times. Two messages are equal, if their sources, destinations, and MIDs are the same within the same lifetime. An endpoint has to remember confirmable messages for a `EXCHANGE_LIFETIME` (247 seconds) and non-confirmable messages for an `NON_LIFETIME` (147 seconds). If a message arrives a second time, the server has to respond with the exact same ACK, RST or response as before without executing the request a second time (at-most-once semantic). Therefore, a duplicate must be identified as such. It is essential that duplicate detection be an atomic operation so that the server can guarantee that it only processes one of two simultaneously arriving twin messages. Californium stores all exchanges in a `ConcurrentHashMap`, which is a powerful, highly optimized data structure that supports concurrents concurrency. We use the method `putIfAbsent()` to atomically check whether a message is already present and insert it into the map if not. The message source and MID give a good hash key and allow many threads to concurrently access the hash map without causing too much synchronization overhead. Periodically, a thread iterates through all entries and checks their age. If the lifetime of a message has expired, the deduplicator removes it from the hash map. This mechanism proved to be far superior to scheduling a task on a timer that removes a message from the hash map after its lifetime. Under a load of thousands of messages per second, the overhead for scheduling all the tasks grows too high.

Due to the deduplication mechanism the layers process each message only once. Besides duplicates, an endpoint has to be prepared for acknowledgements and separate responses that might arrive simultaneously and for multiple responses due to the reasons described in section 5.2. Fortunately, it is possible to implement them with negligible synchronization. However, the current draft for blockwise transfer (block-12 [8]) makes the server, in case of a POST request, send the first and second blocks of a large response at the same time. [3] Both blocks might arrive at the same time at the client which processes them concurrently.

[3]A change in this regard is currently in discussion.

Up to that point, the client does not yet know that the response will be sent blockwise. It is crucial that exactly one of the arriving blocks—be it the first or the second block—sets up the required state for blockwise transfer. Setting up this state is a critical section and if both processing threads entered it at the same time, they would nondeterministically interfere and potentially mess up the blockwise transfer state. Such a race condition can only be solved with synchronization within the blockwise layer. Fortunately, this synchronization is only required if a message carries a block option, and has therefore no influence at all to messages that are not sent blockwise.

When a client has established an observe relation with a resource, the resource sends a notification (response with observe option) whenever its state changes. Since UDP is unreliable, a server has to add a sequence number carried in an observe option to each notification. A higher sequence-number indicates that the notification is newer than a notification with a lower number and a client must drop obsolete notifications. Care must be taken when multiple threads issue sequence-numbers on the server side or decide whether or not to drop a notification on the client side. Assume a resource wants to send two notifications within a short time and two (endpoint-stage) threads process one notification each. Without further information, it is not possible for the two threads to know which notification was produced first by the resource. Even if they use an atomic counter and assign different sequence-numbers to the notification they process, they might have mixed up the order. Therefore, it is essential that the resource itself sets the observe-option and only then forwards them to a next thread. On the client side, multiple notifications might arrive at the same time and be processed by two different threads. At some point, each thread has to decide, whether its notification is new or obsolete. Assume the first thread realizes that its notification is new and shortly afterwards, the second realizes that its notification is even newer. Due to the nondeterministic scheduling behavior, the second thread forwards its notification first and executes the code that updates some state. Nonetheless, the first thread has already decided that its notification is not obsolete and now also invokes the code that updates the state but with the wrong value. Therefore, the update-code that reacts to notifications must be a (synchronized) critical section and do the check for sequence-numbers within that section.

5.5 Separate Stacks

We considered vertically dividing the stack to have separate stacks for different responsibilities. As a result, we could independently configure the two stacks, add different layers to them, or assign more threads to one stack than the other.

Server and client stack: The client stack sends requests and receives responses and the server stack receives requests and sends responses. A client-only application could even omit its server stack and vice versa. However, we would have a problem, splitting up the conceptual components (layers) of CoAP. For instance, non-confirmable requests and responses share the set of *MIDs*. Therefore, the component for sending requests and the component for sending responses must be able to exchange information and to synchronize how they distribute the available *MIDs*. Therefore, client stack and server stack have to exchange information.

Request and response stack: The stack for receiving and sending requests could be separated from the stack for responses. However, we have the same problem with the available *MIDs* from above. Furthermore, we only know whether we receive a request, response or empty message, when we actually decode the received bit string to its internal representation and can only then multiplex over the different stacks. It is also not clear, where to process empty messages.

Receiver and sender stack: The stack for sending requests and responses could be separated from the stack for receiving them. This would also split up conceptual components such as matching. When the sender stack sends a request, it has to pass the token to the receiver stack so that it recognizes the response later. Furthermore, multiple components (responsible for matching, reliability or blockwise transfer) need to send messages when receiving one. Thus, the send and receive stack need to be tightly coupled anyway.

In general, splitting up the stack requires the definition of two interfaces instead of one and ultimately leads to more complexity for negligible advantages. Therefore, we decided to not split up the stack in any way.

6 Performance Evaluation

One goal of Californium is to host services that communicate with other endpoints. In the IoT, we expect endpoints to exchange small messages. We consider each request and response a single unit of information. For the endpoints that communicate with a server, the relevant factors are the number of information that can be exchanged per time and the latency. Whether it is a CoAP or an HTTP message that carries the information from one endpoint to another is of no importance for the application. We measure the throughput of a server as the average number of requests that it is able to handle per second. This chapter presents the results of our experiments with Californium, Old Californium, and five state-of-the-art HTTP servers. Therefore this evaluation also serves as a comparison between CoAP and HTTP as protocols in the service backend. We particularly evaluate the scalability of the seven servers with respect to the number of available cores and with respect to the number of endpoints that concurrently communicate with the server. A node In the World Wide Web usually is either in the role of a server or a client. Therefore, HTTP servers truly are servers only and are optimized for that purpose. For the sake of comparison, we concentrate our evaluation of Californium and Old Californium only on their role as server even though both can appear as clients as well.

We distinguish between two use-cases in which a server reaches its maximum performance. First, there might be a few clients that send many requests to the server. Such a client might be a proxy, for example, that in fact forwards requests from many clients but represents itself to the server as one highly demanding client. If server and clients used HTTP, they could keep a TCP connection alive and exchange many messages over it. In the second case, there are a large number of clients, each sending only a single request to the server. For an HTTP server, this means that each client establishes a new TCP connection only to exchange one request and immediately terminate it again. This is the scenario for a resource directory (RD). An RD can be thought of as an address book for resources. A endpoint can for instance register its resources of a specific type at the RD and another endpoint that looks for such a resource might later retrieve its URI form the RD. To simulate this scenario, HTTP clients do not keep their TCP connections alive but reestablish it for each request. The distinction between these two use-cases is much more prominent for HTTP than for CoAP as there is no such thing as establishing a connection in CoAP and it makes no difference whether requests come from the same or many different clients.

6.1 Experimental Environment

The experimental environment consists of three client machines and a single server. The server is a Lenovo ThinkPad W530 with an Intel Core i7-3720QM (Quad-Core, 2.6 GHz), 24 GB of RAM and an Intel 82579 LM Gigabit network card. We use a 64-bit Windows 7 and Java 1.7.0_09 with Java HotSpot™ 64-bit Server VM and allocate 4 GB of RAM for the *Java Virtual Machine (JVM)*. We disable hyper-threading. The server is connected to the client machines over Gigabit Ethernet. Figure 6.1 illustrates the setup in detail.

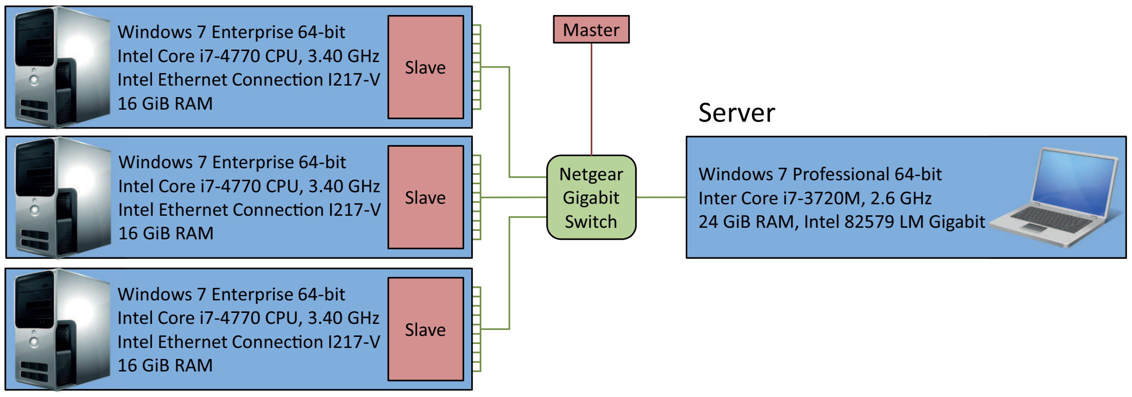

Windows 7 Enterprise 64-bit
Intel Core i7-4770 CPU, 3.40 GHz
Intel Ethernet Connection I217-V
16 GiB RAM
Slave
Windows 7 Enterprise 64-bit
Intel Core i7-4770 CPU, 3.40 GHz
Intel Ethernet Connection I217-V
16 GiB RAM
Slave
Windows 7 Enterprise 64-bit
Intel Core i7-4770 CPU, 3.40 GHz
Intel Ethernet Connection I217-V
16 GiB RAM
Slave
Master
Netgear Gigabit Switch
Server
Windows 7 Professional 64-bit
Inter Core i7-3720M, 2.6 GHz
24 GiB RAM, Intel 82579 LM Gigabit

Figure 6.1: The quad-core server is connected to three client machines over Gigabit Ethernet. Each client runs a slave instance of CoAPBench and simulates multiple virtual clients that all bind to a port and send requests to the server. The master sends a signal to the slaves to start the benchmark.

We use ApacheBench[1] to measure the performance of HTTP servers. ApacheBench creates a configurable number (called "concurrency level") of virtual clients. Each client sends requests to the server as fast as it can handle them, i.e., it always waits for a response before sending another request. We further can decide, whether virtual clients are allowed keeping their connection alive or not. Unfortunately, there is no such benchmark tool publicly available for CoAP. Therefore, we developed CoAPBench, a benchmark tool for CoAP servers. As it turns out for both tools, running only on one machine may not be enough to fully saturate the capacity of the server. Therefore, CoAPBench can be distributed over multiple machines. A centralized master is connected over TCP to one slave-instance of CoAPBench per physical machine. The master simultaneously sends a starting signal to all slaves, containing the number of virtual clients that are supposed to concurrently send requests. Each virtual client sends requests for 60 seconds and counts the responses. The sum of responses of all virtual clients divided by the time period of 60 seconds gives the average throughput that the server is able to achieve. It is possible that a UDP packet goes lost and a virtual client does not receive a response. After 10 seconds with no response, a virtual client considers the request as failed and sends a

[1] `http://httpd.apache.org/docs/2.2/programs/ab.html` August 22, 2013

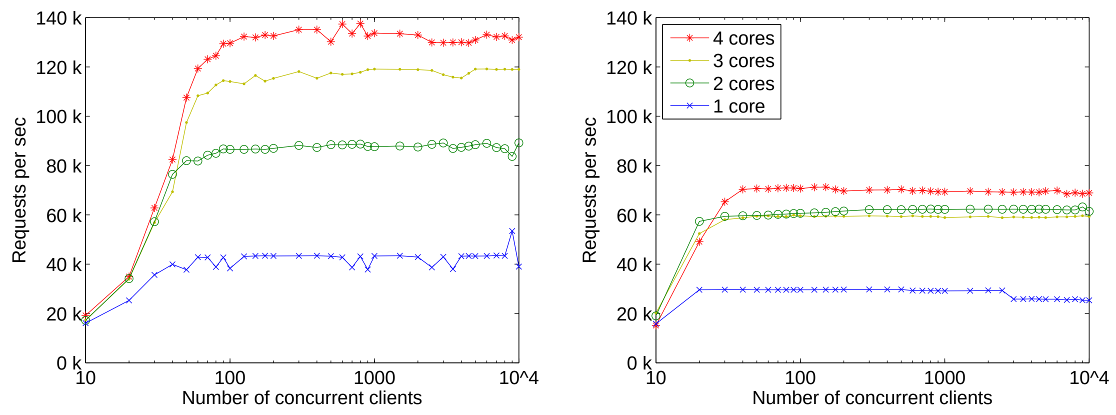
new one. In most measurements, no timeouts occurred and if, then the number was far below a thousandth of the amount of successful exchanges. CoAPBench is extensible with third-party software, which allowed us to also run multiple instances of ApacheBench on multiple machines.

6.2 Californium vs. Old Californium

We have built two simple, equivalent servers with both versions of the CoAP frameworks. The servers listen on port 5683 and have a simple resource at the top of the resource tree that responds to GET requests with the payload “hello world”. All GET requests are confirmable and the responses piggy-backed ACKs. The simplicity of the benchmark resource reduces the effort both servers must carry out to a minimum so that we can compare the bare protocol processing of the two frameworks. Since the server has to remember all requests that it has received to detect duplicates, the memory consumption becomes very high under the load we achieve. CoAP allows to relax duplicate detection for idempotent requests such as GET requests. Since the 24 GB RAM of our server machine is not enough memory, we have disabled duplicate detection. We have configured Cf to use 4 threads in the connector-stage for sending, 4 threads in the connector-stage for receiving, 4 threads in the endpoint-stage for executing the protocol and the benchmark resource does not create its own thread-pool, and therefore the endpoint-stage threads also execute the code in the resource-stage. The most recent OCf version uses a single thread for receiving requests and a thread-pool with 10 threads for executing the code in resources and then sending the response.

We measure the servers with different numbers of virtual clients. We start with a concurrency level of 10 and increase it exponentially up to 10,000. To investigate scalability of the servers in the presence of multiple cores, we bind them alternately to 1, 2, 3, or 4 cores. Figure 6.2 a) and b) show the average number of requests that the Cf and OCf servers are able to process per second. When we reach 150 concurrent virtual clients, the performance curve of both Cf and OCf stabilizes. The results of Cf with 1 or 2 cores are slightly better than those of OCf. The performance of OCf is bound by the performance of the single receiver thread which has to copy the packets from the socket and move them upwards through the stack until it can pass them to the thread-pool. This explains why OCf does not achieve the performance of Cf where multiple threads constantly work on receiving requests and others process the message in the endpoint. With 3 or 4 cores, Cf clearly outperforms OCf. Cf performs 3.5 times better on 4 cores than on 1 core. Real-world resources certainly need to do more work than our dummy resource for the benchmark. If these resources do not introduce synchronization, Cf is expected to scale even better as the fraction of concurrent computation (CoAP protocol + computation in resource) grows compared to bottlenecks such as synchronization between stages, the resource tree or hardware such as the network card.

The latency was always very low. Up to 1000 concurrent clients, more than 99% of all requests were responded within less than one millisecond. With more clients, the latency linearly increased up to 10 milliseconds which is still very low, considering the high concurrency level.



a) Califonrium's throughput.

b) Old Californium's throughput

Figure 6.2: Californium achieves a maximum throughput of 140,000 requests per second. Four cores are about 3.5 times better than one core. Old Californium achieves a comparable throughput with one and two cores but scales bad for three and four cores.

6.3 HTTP Servers

There exist numerous HTTP servers that we could compare Californium with. Since Californium is designed to host services that compute the content of a response dynamically, we have ruled out HTTP servers that only serve static content (files). We have finally chosen five servers according to their prevalence in the industry and novelty in architectural design.

Vert.x (version 1.3.1) is a novel event-driven HTTP server that describes itself as "a lightweight, high performance application platform for the JVM [. . .]."[2] A server exports its content within so called Verticles, which are independent from each other as each runs in its own 'virtual' *JVM*. Vert.x's strength is the ability to seamlessly integrate application logic written in one of several supported languages wrapped in a Verticle. Although Vert.x's backbone uses multiple threads, each Verticle experiences a strictly single-threaded event-driven environment. This makes it hard for a service wrapped in a single Verticle to be scalable. Instead, one can start Vert.x with a parameter that configures how many parallel instances of a specific

[2]Vert.x: `http://vertx.io`, October 10, 2013

Verticle will be created. However, these instances can only exchange information with each other over Vert.x's event-bus—they do not even share static variables. The consequence for a real-world application is a significant overhead in programming code and computation to synchronize Verticle instances. This is less of a problem for our benchmark since the server is only supposed to respond with a small payload as described for the Cf server. Therefore, we always configure Vert.x to create 4 instances so that it is able to fully exploit the four cores of the machine.

Apache Tomcat (version 7.0.34) is the most prevalent Web server for Java Servlets.[3] Tomcat is open source, written in Java and has been first released in 1999. It implements a multi-threaded architecture where one thread is started per TCP connection. For our benchmarks, we have disabled Tomcat's logging, set the thread limit to 10,000 and start it with 4 GB RAM.

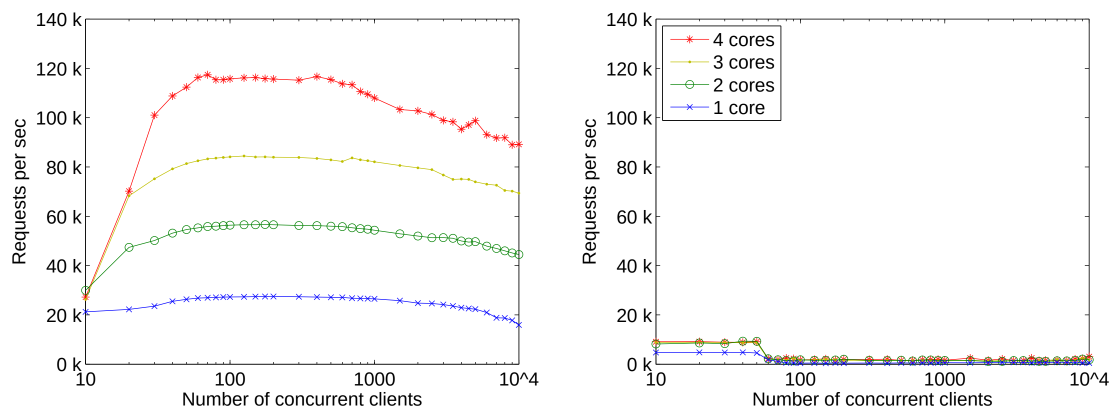
Apache HTTP Server (version 2.4.4) "has been the most popular Web server on the Internet since April 1996, [. . .]."[4] Apache HTTP server has traditionally been paired with PHP (version 5.5.1) to serve dynamic content. By default, the server uses a multi-threaded architecture on Windows. We have disabled all logs and set the thread limit to 10,000.

Project Grizzly (version 2.3.6) builds upon Java's New I/O (NIO) package. Grizzly has been developed by Sun Microsystems and Oracle and is used as servlet container in Glassfish, a server for Java EE. [5] Behind the event-driven network I/O, the server uses a multi-threaded architecture.[6]

Node.js (version 0.10.20) is an event-driven Web server. [7] Node.js is written in C++ and utilizes Google's V8 JavaScript engine. JavaScript network applications can be deployed on the server and are independent from each other. Node.js executes JavaScript code with only a single thread. It is possible to deploy an application multiple times like in Vert.x with a so called "cluster", however, this mechanism is still experimental and we have omitted it.

In the following, we present the results for all HTTP servers. For all following figures, we show HTTP's results when clients keep connections alive on the left side and the results when they do not on the right side. We use the same scale for all figures in this section as for the results of Californium. Figure 6.3 shows the results for Vert.x. When Vert.x keeps connections alive, it achieves a throughput almost as high as Californium and peaks at a concurrency level of 70 with a throughput of 117,000 requests per second (85% of Californium). With more than 1000 concurrent clients, Vert.x's performance

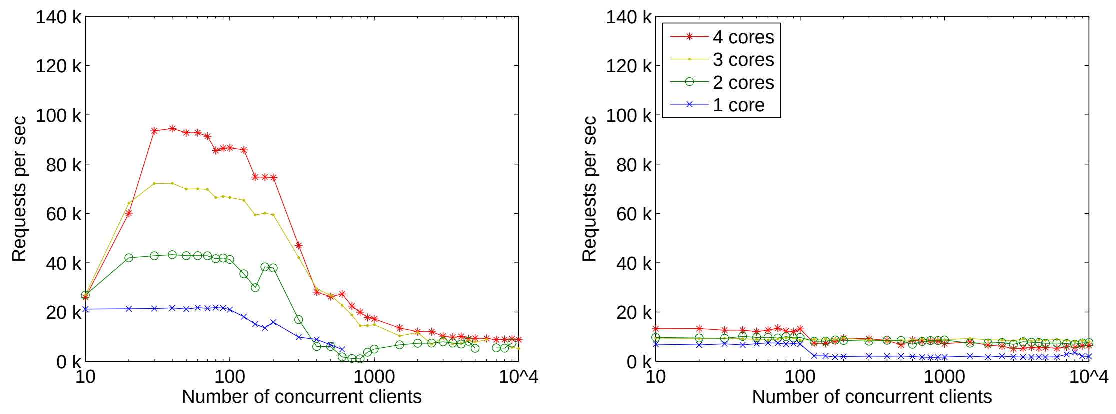
[3]Apache Tomcat: `http://tomcat.apache.org`, October 10, 2013
[4]Apache HTTP server: `http://httpd.apache.org`, October 10, 2013
[5]`https://glassfish.java.net/de/`, October 12, 2013
[6]`http://jfarcand.wordpress.com/2006/01/26/grizzly-nio-architecture-part-ii-2`, October 10, 2013
[7]Node.js: `http://nodejs.org`, October 10, 2013



a) Vert.x with keep-alive b) Vert.x without keep-alive

Figure 6.3: When keeping connections alive, Vert.x peaks at a concurrency level of 70 with a throughput of 117,000 requests per second. In contrast, Vert.x performs poorly (max. 9100) without the keep-alive mechanism.

slowly decreases. The highest values for four cores are around 4 times better than for one core, showing that Vert.x scales very well. In contrast, Vert.x performs poorly without the keep-alive mechanism. More than two cores do not help much and for a concurrency level of 60 or more, the throughput drops below 2500 requests per second. In fact, we are going to observe such a sudden drop in throughput in all HTTP servers when they do not keep connections alive.

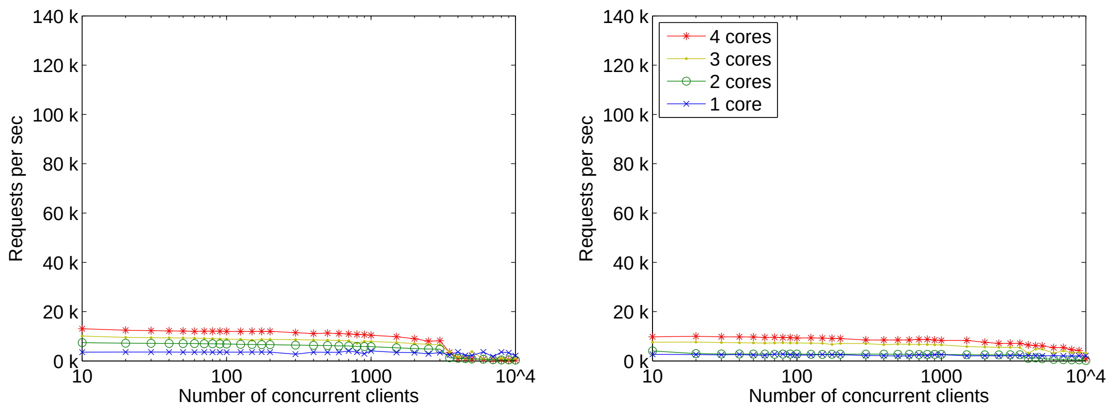


a) Tomcat with keep-alive b) Tomcat without keep-alive

Figure 6.4: Tomcat achieves maximum 70% of Cf's throughput when keeping connections alive (max. 94,000) and about a tenth otherwise (13,400).

Figure 6.4 shows the throughput of Apache Tomcat. When keeping connections alive, Tomcat at its best is able to handle 94,000 requests per second (70% of Californium). Four cores are about three times as fast as one core and the CPU utilization always reaches 100%. Tomcat peaks at a concurrency level of 30 and slowly decreases with a higher concurrency level. With more than 200 clients, Tomcat's throughput drastically drops. When not keeping connections alive, Tomcat peaks at 70 with around 13,400 requests per second. Interestingly, Tomcat's throughput with one and four cores drops at a specific point, while its throughput with two and three cores is very stable. Tomcat achieves the highest throughput among all HTTP servers without keep-alive. The CPU utilization with four cores fluctuates between 25% and 40%. It is possible that a major bottleneck in this case is the network card or the way Windows manages network I/O.

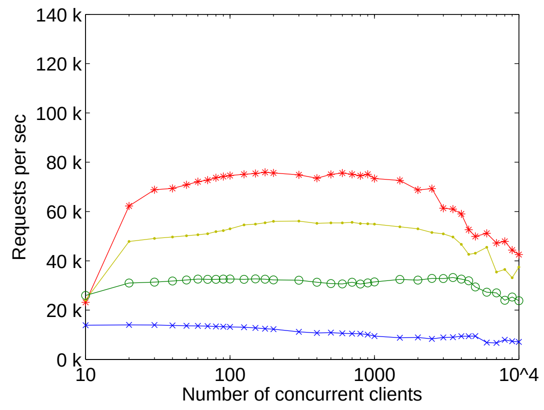
Figure 6.5 shows the results for Apache HTTP Server with PHP. Apache achieves rather low throughput even though the CPU is always fully utilized. We assume that the mechanism how Apache invokes the PHP code must be computationally intensive, even though the code consists only of a simple statement: `echo 'hello world';`. Apache scales well: four cores are 3.3 times better than one core. With keep-alive, we have about 3500 requests per second on one core, 6700 on two cores, 8800 on three cores and 11,800 on four cores. Without keep-alive, the results are a little lower with 2400 for one core, 2800 for two cores, 7100 on three cores and 9200 on four cores. Apache's throughput is very stable on its low level but the server struggles with more than 5000 concurrent clients. Surprisingly, Apache performs worse with keep-alive at this point than without. We assume the reason for this is that the number of clients at this point is larger than Apache's throughput and the effort to manage so many 'living' connections becomes heavier than just reestablishing them.



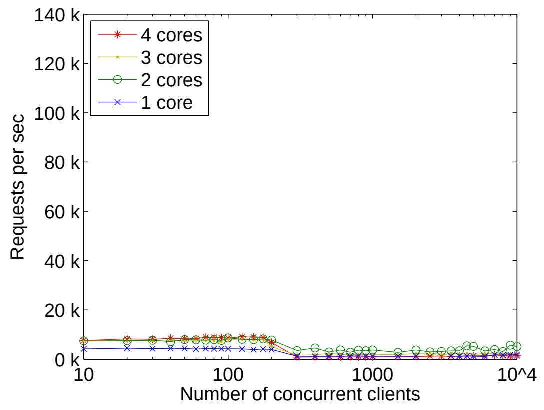
a) Apache + PHP with keep-alive   b) Apache + PHP without keep-alive

Figure 6.5: Apache plus PHP does not achieve a very high throughput with (max. 13,100) and without (max. 10,000) the keep-alive mechanism. Apache always utilizes the CPU to its fullest and scales well.

Grizzly achieves an admissible throughput of 76,000 and keeps it up to a concurrency level of 2000 as shown in Figure 6.6. Grizzly's scalability is exceptional; it scales better than it should. Grizzly's throughput with four cores is around 2.2 times higher than with two cores and even 6 times higher than with one core. There is no obvious explanation for this behavior. In contrast, Grizzly does not scale as well without keep-alive. More than two cores do not improve the throughput of the server. When using one, three or four cores, the throughput with more than 300 concurrent clients even drops below 1000. With a high concurrency level, two cores show the best performance.



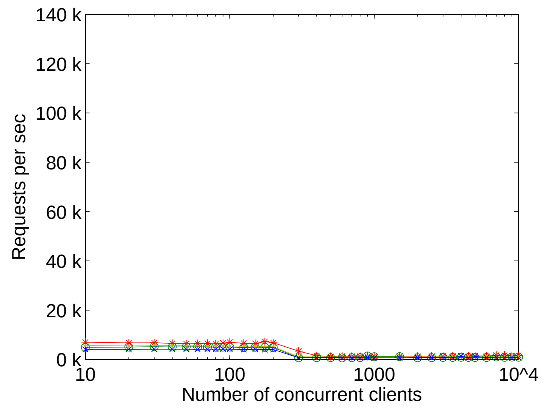
a) Grizzly with keep-alive



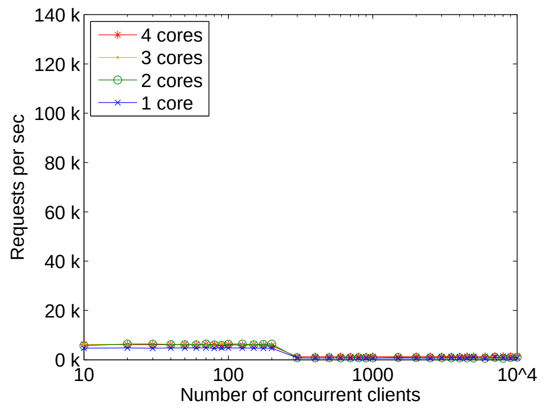
b) Grizzly without keep-alive

Figure 6.6: With keep-alive, Grizzly achieves an admissable throughput of 76,000 even for more than 1000 concurrent clients. Interestingly, it performs 6 times better with four cores than with one core. Without keep-alive, Grizzly's throughput (max. 9100) drops at 300 clients, except when running on two cores.

Node.js achieves the lowest throughput of all servers with and without keep-alive. In both cases, a sudden drop appears between a concurrency level of 200 and 300. Since Node.js uses only a single thread for JavaScript code, it has no chance against the other servers. Node.js's internal use of more than one threads explains why it scales at least a little.



a) Node.js with keep-alive



b) Node.js without keep-alive

Figure 6.7: Node.js achieves the lowest throughput of all servers (max. 7200 and 6200). Since it internally uses more than one core, it scales at least a little.

6.4 Performance on a Many-Core System

We evaluate Californium and Old Californium on a many-core *Non-Uniform Memory Access (NUMA)* system and compare them to the two HTTP servers which have achieved the highest troughput on the quad-core: the event-driven Vert.x and the multi-threaded Apache Tomcat. The system consists of two AMD Opteron(TM) Processor 6212 each having two nodes each having four cores (16 cores in total). A core runs at a frequency of 1.4 GHz and has a cache of 2048 KB. A node with four cores has 32 GB of RAM. The OS is a Red Hat Enterprise Linux Workstation release 6.4 (Santiago) with kernel version 2.6.32-358.14.1.el6.x86_64. Figure 6.8 shows the NUMA setup. We use the same Java version as for the quad-core. We have two such systems, one runs the server and another emulates the clients. They are connected over 10 Gigabit Ethernet. We run each server on 16, 8, 4, 2, and 1 core and measure its average throughput with an increasing concurrency level from 10 to 1000 (in steps of 10 up to 200 and then in steps of 25 up to 500 and then in steps of 100 up to 1000). Each measurement takes again 60 seconds. We always deploy as many instances in Vert.x as there are cores. Californium uses one thread for receiving requests and one thread for sending responses in the connector-stage and as many threads in the endpoint-stage as cores are available. We always run the *JVM* with 4 GB RAM and use the flag `-XX:+UseNUMA` to turn on Java's NUMA-aware memory allocator in conjunction with the Parallel Scavenger garbage collector.

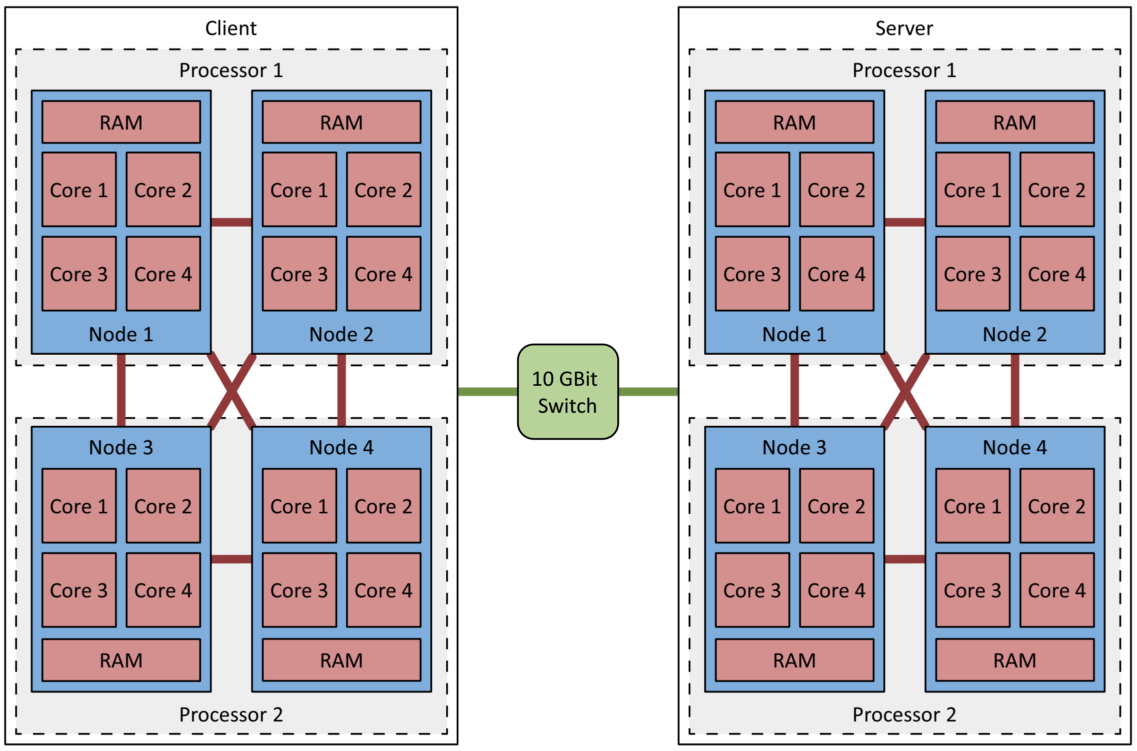
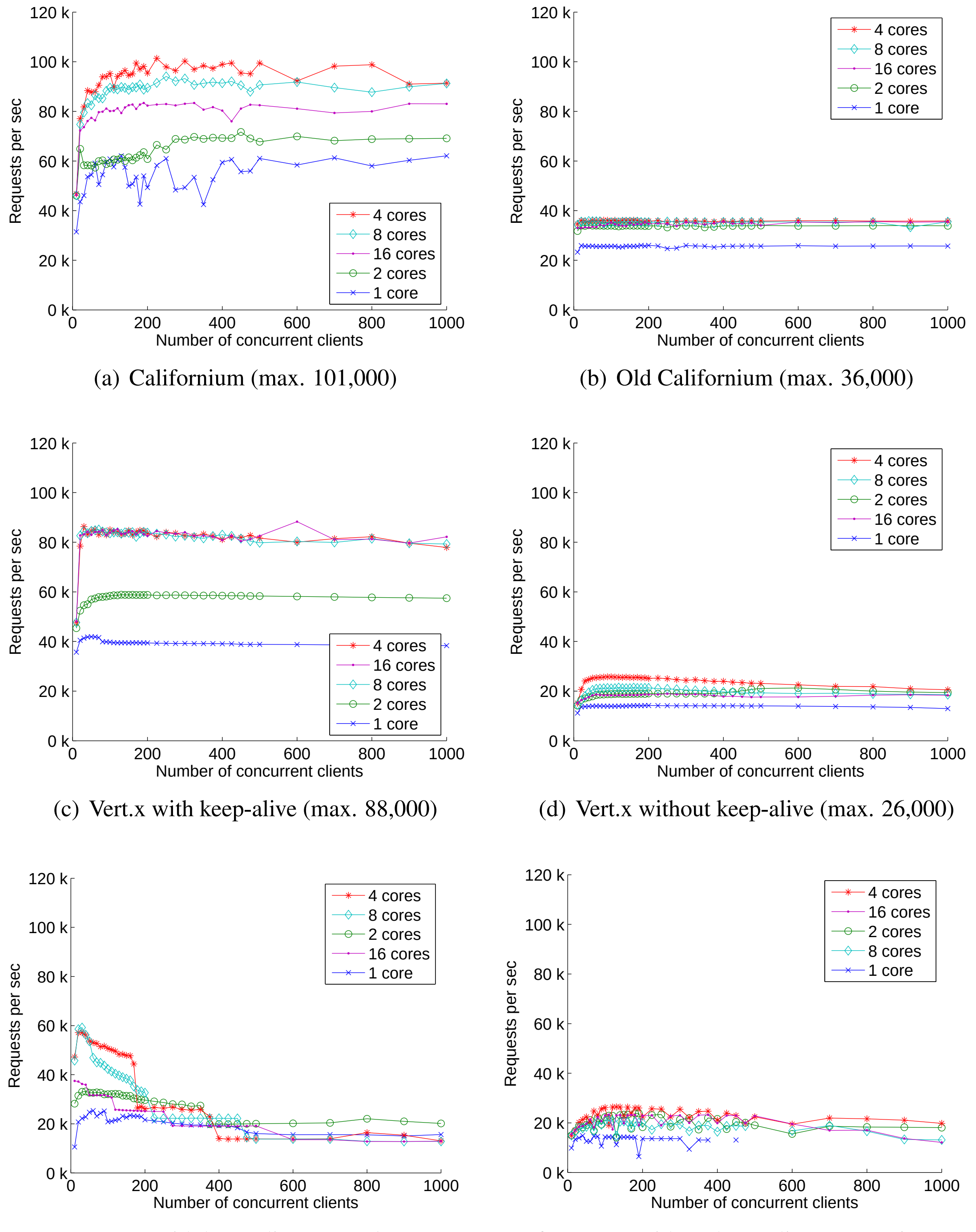


Figure 6.8: A client and server NUMA machine, both having 4 nodes with 4 cores each.



(a) Californium (max. 101,000)

(b) Old Californium (max. 36,000)

(c) Vert.x with keep-alive (max. 88,000)

(d) Vert.x without keep-alive (max. 26,000)

(e) Tomcat with keep-alive (max. 59,000)

(f) Tomcat without keep-alive (max. 26,000)

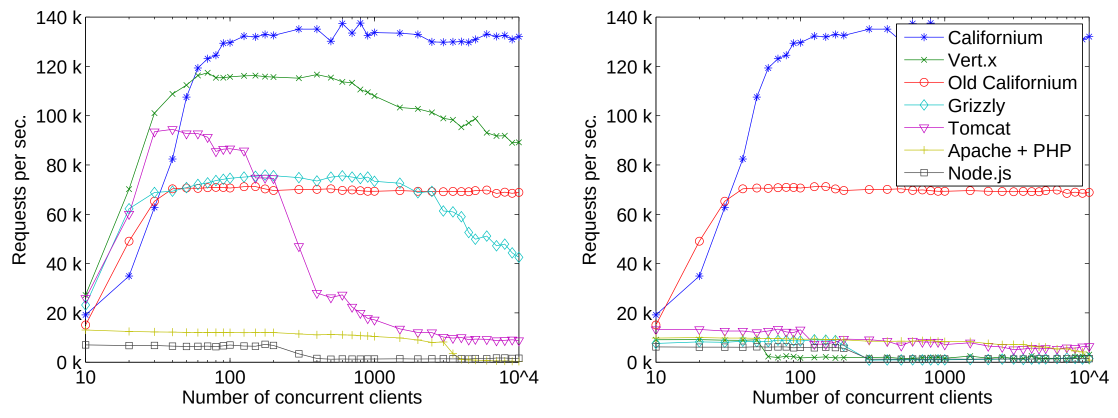
Figure 6.9: The results for Californium, Old Californium, Vert.x and Tomcat on the 16-core NUMA system. Californium perfoms best, followed by Vert.x with keep-alive.

Figure 6.9 shows the throughput for all servers on the NUMA system. We have ordered the legends of each figure according to the average throughput a line represents. Interestingly, no server performs significantly better when having 16 cores but typically even worse. Californium (a) once again achieves the highest throughput with about 100,000. However, 8 and 16 cores perform worse. Old Californium (b) cannot even benefit from more than two cores but is very stable slightly below 40,000 requests per second. Vert.x with keep-alive (c) is a close second and achieves with 4, 8 an 16 instances on an equal number of cores the same throughput as Cf with 16 cores. There is one outlier at a concurrency level of 600 that we cannot explain. Unfortunately, we no longer had access to the NUMA system to double-check this outlier, when we found it. Interestingly enough, Vert.x's performance without keep-alive (d) is an order of magnitude better on the 16-core than on the quad-core. The curve peaks with 25,000 and behaves very smooth. Vert.x may be better compatible with the network card of the NUMA system or the socket support of the OS. Tomcat with keep-alive (e) cannot keep up with Cf and Vert.x when the concurrency level surpasses 200. The curve of Tomcat's throughput stepwise decreases towards a higher concurrency level. Without keep-alive, Tomcat (f) is on the same level as Vert.x except when running on only one core. With 400 or more concurrent clients, ApacheBench often is not able to establish the required amount of TCP connections and aborts the measurement prematurely. This was not the case for Tomcat on the quad-core, and therefore might be a problem of the OS socket implementation Tomcat depends on.

6.5 Discussion

We have evaluated the performance of five HTTP servers. Apache HTTP Server and Apache Tomcat represent the fraction of multi-threaded servers, while Vert.x and Node.js are event-driven, and Grizzly is a hybrid. Figure 6.10 summarizes the results from all servers on the quad-core when all four cores are available. Just from the results, it is not possible to choose a clear winner among the architectures. Vert.x and Tomcat both reach a high throughput, Node.js and Apache HTTP Server both do not, and the hybrid Grizzly is somewhere in the middle. Vert.x, which scales best towards many concurrent clients, achieves its scalability by replicating the benchmark Verticle, which is hard to compare with. If the Verticle kept a changing state, the replicas would have to synchronize.

Californium and Old Californium both scale better with respect to a growing number of clients. On both machines, we observe that the throughput stays high even for very high concurrency levels. Since CoAP is based on UDP, there are no connections and the server does not need to keep any state or even threads for a specific client. It does not matter, whether requests stem from the same source or not. With a higher concurrency level, only the latency per request increases. If we increased the concurrency level even more, the buffer of the switch or the network card of the server machine eventually would overflow and would have to drop packets. The number of timeouts would start to grow but the throughput would remain the same. In contrast, even though HTTP in principle is a



a) Quad-core with keep-alive b) Quad-core without keep-alive

Figure 6.10: Summary of all servers on the quad-core with and without keep-alive. Californium's throughput is clearly the highest. In the use-case of a resource directory where thousands of nodes send only one request at once, the throughput with CoAP is tenfold as high as with HTTP.

RESTful protocol, TCP is not and the server has to hold and maintain the state that comes with each connection—collapsing at some point.

It is debatable whether CoAP should be rather compared to HTTP with keeping connections alive or without. In the use-case of the proxy and few other nodes that each sends many requests, it seems fair to compare CoAP to HTTP with the keep-alive mechanism enabled. Californium then performs about 15% better than the best HTTP server. In the use-case of the RD, however, nodes cannot benefit from keeping connections established. Consequently, our results suggest that the throughput with CoAP is tenfold as high as with HTTP without keep-alive.

While most servers scale very well on the quad-core, none is able to utilize more than four cores on the *NUMA* system. 16 cores have so much computing power that the socket becomes the bottleneck. The receive and send methods of the datagram socket both are synchronized, which limits the performance of the socket. Generally, the best performance comes with four cores, i.e., exactly one node whose cores share the local memory. On a *NUMA* system, the memory access time depends on its location. When using more than four cores, memory inevitably needs to travel from one node to another which slows down the system. Further phenomena like false sharing may be part of the problem as well. The *JVM* distributes the threads autonomously over the available cores giving us no possibility to assign a specific thread to a specific core with pure Java tools. The *JVM*'s abstraction hinders us from potentially optimizing Californium for *NUMA* systems. In all our tests, we never managed to utilize more than 20% of the total computing capacity. It is clear that a real-world application would do more work than just send a dummy payload as in our benchmark. If the services that were to be hosted on Cf required a lot of computation, it might be worth running it on a many-core machine.

On the 16-core *NUMA*, we have measured, whether we can achieve a higher throughput, if Californium listens on more than one port. Indeed, the combined throughput of multiple sockets on different ports can be twice as high as with one socket. However, we believe that is of no use for the majority of real-world applications, and therefore have limited our benchmarks to servers listening on one port. We have further compared the throughput that a normal (blocking) datagram socket achieves with the one of a NIO socket. On both machines, the quad-core and the 16-core *NUMA* system, the blocking datagram socket performed twice as fast as the NIO socket. We found on the quad-core with Windows, however, that we can double the throughput of a blocking socket by using four threads to receive messages. On the other hand, the throughput of a socket on the Red Hat 16-core *NUMA* drops about 60% when two threads instead of one invoke the receive-method. Luckily, Californium allows us to freely configure the number of threads in the connector.

7 Optimization

First benchmarks suggested that Cf should be able to process about 100,000 requests per second. In these benchmarks, clients and server ran on the same machine. When we moved the clients to different physical machines, we expected the performance to increase even further or at least remain the same. First results showed a disappointing throughput of 60,000 requests per second with a CPU utilization of only 60% and bad scalability for 3 and 4 cores. We located the bottleneck somewhere between the network card and the JVM. The method call that copies an incoming packet into Cf's memory is rather time expensive. As in OCf, the methods for sending and receiving UDP packets take a significant fraction of the total computation time. This is not the case when the client and server are executed on the same machine. The operating system seems to be able to transfer UDP packets much faster from process to process than from the network card to a process. It is imaginable that a designated server machine is equipped with a much faster network card and more efficient network I/O than our commodity notebook so that this problem is not apparent in the first place. We found a solution to the problem by designating more than one thread to receiving packets from the datagram socket. This is surprising because the receive-method of the socket is synchronized. The method for copying the bytes from the packet into a byte array, however, is not. With four receiver-threads, we now achieve the expected throughput of 140,000 requests per second (as described in Chapter 6) and are able to fully utilize all four cores of the CPU. Since Cf wraps the transport protocol within a connector that has its own multi-threading strategy, changing the number of threads in a connector is perfectly allowed.

When Californium runs under heavy load, it constantly allocates and frees up a significant amount of memory. In Java, the garbage collector (GC) is in charge of finding obsolete objects and reclaim their memory. An object is obsolete if it is no longer reachable over references from the program call stack. Periodically, the GC must be invoked and clean up the memory, which is called a GC-iteration. The majority of objects that GC cleans up are young objects. Therefore, Java manages its memory in 'generations.'[1] When a program constructs a new object, the object is usually placed in the 'young' generation and only after it has survived multiple iterations, the GC is going to move it into the 'tenured' generation. GC is more efficient, when it cleans the young generation more often than the tenured generation.

[1] `http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html`, August 14, 2013

In particular in a concurrent software, finding unreachable objects is difficult because race conditions must be avoided. Java comes with several garbage collectors that implement different strategies. The most important factor for choosing the correct strategy is how long it has to suspend threads. This depends on the total size of the memory, the sizes of the generations, the amount of work that GC can do in parallel, memory access time, and of course the strategy itself. Choosing the correct GC can drastically improve the performance of a memory-intensive software. Most *JVMs* even choose a GC at runtime based on the machine they are running on. On the quad-core, we achieved the best results with the default, serial GC. Under heavy load, the Windows Task Manager reported the CPU utilization as shown in Figure 7.1. The red line indicates the fraction of time spent in kernel space and the green line the total utilization of each core.

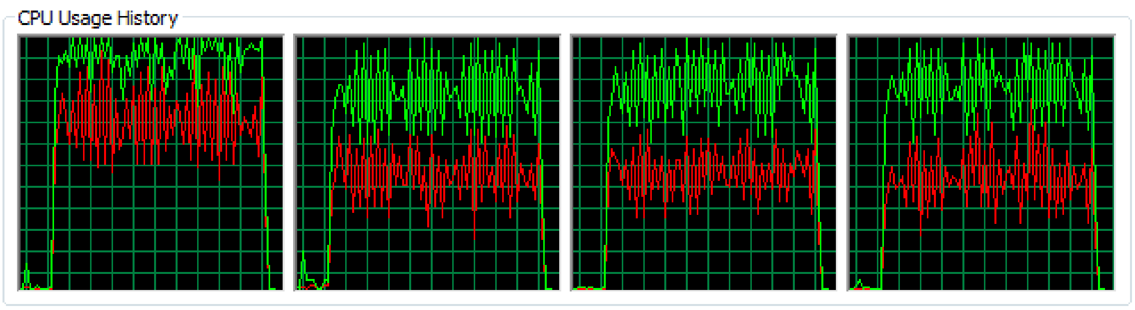
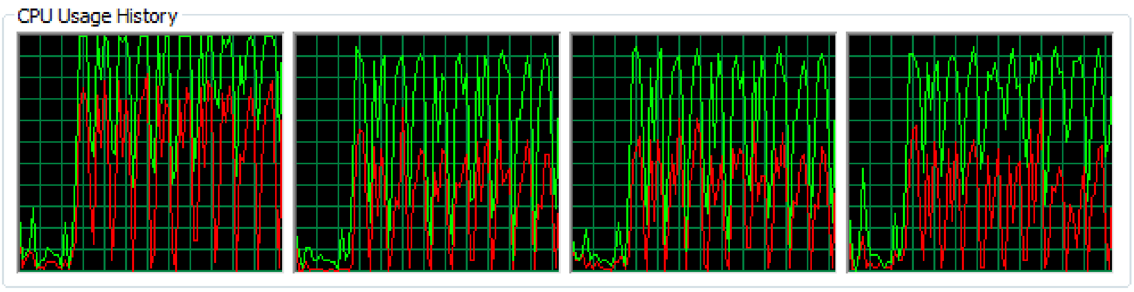


Figure 7.1: CPU utilization with the serial garbage collector on the quad-core. From left to right the utilization of core 0, 1, 2 and 3 are shown. The red line depicts the fraction of time spent in kernel space and the green line the total utilization of a core.

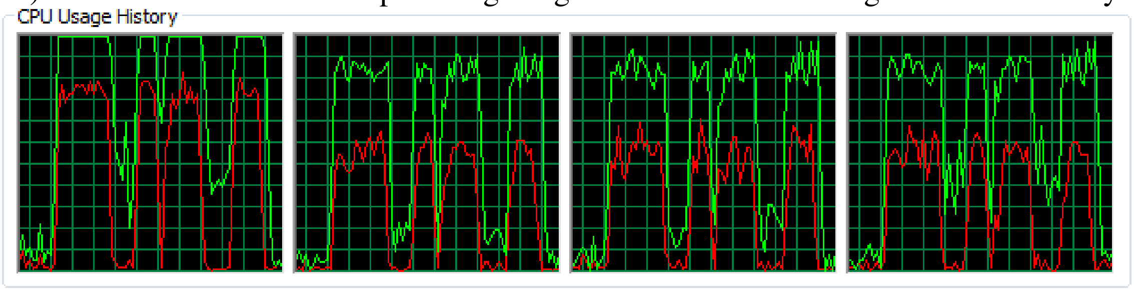
The 16-core NUMA system has a much different hardware layout. On a NUMA machine, Java can benefit from a NUMA-aware memory allocation algorithm and GC. We can explicitly enable both with the JVM parameter `-XX:+UseNUMA`. As a result, the JVM uses the Parallel Scavenger garbage collector, which we can also explicitly turn on with the parameter `-XX:+UseParallelGC`.[2] We have observed that the CPU utilization on the quad-core with the parallel GC looks much different from the serial GC.

We compare Cf with 3 GB of memory to Cf with 12 GB of memory. Figure 7.2 shows the CPU utilization of Cf over 60 seconds with 3 and 12 GB of memory. Whenever the kernel time (red line) drops, a GC iteration was performed. The overall throughput in terms of processed requests per second is roughly the same in both setups. However, from the perspective of the clients, the server behaves differently. When Cf has a larger memory space, the server can longer process requests before the next GC iteration is required. A single iteration takes more time, though. At some point, GC needs to suspend all Java Threads to have exclusive access to the memory. This suspension directly reflects in higher

[2]`http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html`, August 14, 2013



a) CPU utilization when the parallel garbage collector has to manage 3 GB of memory



b) CPU utilization when the parallel garbage collector has to manage 12 GB of memory

Figure 7.2: The more memory a server has, the less often garbage collection is required but the more time each iteration takes. This increases the latency a client sometimes experiences.

latency for request currently in Cf's pipeline. A GC iteration over 12 GB of memory takes about 3.5 seconds on our system. As a result, increasing Cf's memory increases the potential for timeouts on the client side. Fortunately, there are several JVM parameters that allow tuning the GC behavior of the JVM. In particular, one can chose an algorithm that operates more concurrently to other threads, thus, reducing the time threads must be suspended (`-XX:+UseParallelGC` or `-XX:+UseConcMarkSweepGC`). Some parameters (`-XX:MaxGCPauseMillis=<N>` and `-XX:GCTimeRatio=<N>`) allow to further control the suspension time of threads if the latency is a concern.

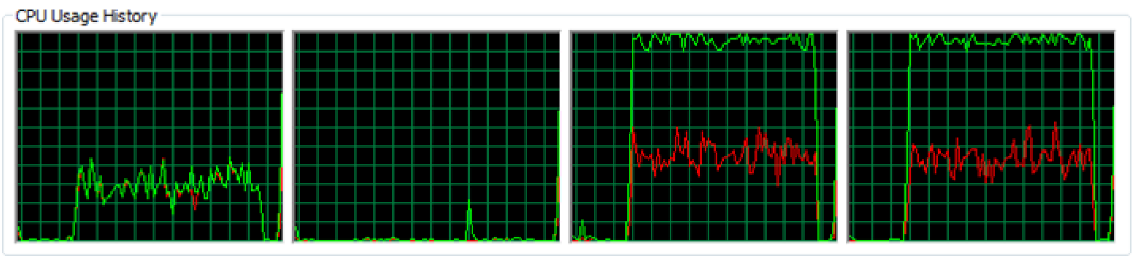


Figure 7.3: Cf bound to cores 2 and 3: both cores spend the same time in kernel space. We assume that Windows internally strictly uses core 0 for computations for network I/O.

An interesting fact is that the average fraction of kernel space computation is in all figures larger on core 0 than on the others. When we bind Cf to only core 2 and 3, we see a CPU utilization as shown in Figure 7.3. We see that both cores 2 and 3 spend the same amount of computation in kernel space. Interestingly, we see some activity on core 0 and it is all in kernel space (the red line is shadowed by the equally shaped green line). Needless to say, no other process was responsible for this activity. We assume that Windows internally strictly uses core 0 for computations for network I/O. We did not observe this phenomenon on the NUMA system with Red Hat.

8 Application Programming Interface

Californium's ultimate goal is to allow developers creating services that communicate over the CoAP protocol. This can be thought of as a two-layer application where Cf forms the bottom and the service logic builds on top of it. Californium's *application programming interface (API)* specifies how the service can interact with Cf, i.e., send and receive CoAP messages and export content in form of CoAP resources. The definition of the term *API* is somewhat fuzzy. In case of Cf, the *API* consists of several classes with public methods which the developer is encouraged to use. Quality aspects such as usability and learnability mostly depend on a good *API* and directly affect how easy a developer can familiarize himself with Cf.

There exist two books written by experts—"Effective Java" by J. Bloch [6] and "Framework Design Guidelines" by K. Cwalina (Microsoft) [15]—that give guidelines for *API* design and outline tradeoffs of different design decisions. These guidelines are not about decisions among design patterns but rather general advice, e.g., preferring interfaces to abstract classes or as preferring lists to arrays. Additionally, Stylos at al. have published several research papers [16, 43–45] about *API* design. Stylos et al. identify two groups of stakeholders of an *API*: designers and users. Our focus lies on the needs for the group of users which we call the developers (They develop the application that uses Cf's *API*). Developers expect powerful features and want to quickly write efficient, error-free programs. Stylos suggests making multiple sets of *APIs*, for instance one for novice developers and another for experts. We have adopted this proposal and provide not only the same classes that Cf internally uses to experts who ask for powerful features (remember that CoAP is not final yet but might still be extended) but also have designed a novice-friendly *API*. The latter focuses on simplicity and understandability while the expert *API* focuses on being powerful, flexible, and extensible.

8.1 Design Guidelines

We first give a list of *API* design decisions that we find most useful for making Cf's *API* (the one, experts would look at) more understandable and secondly describe our novice-friendly *API*. The guidelines from "Effective Java" are organized in 'items,' which the interested reader is invited to consult for more detailed information.

Method placement: Stylos et al. have evaluated how developers explore an *API* [45]. Developers often start from one "main" object and have trouble finding other objects that are not referenced in the methods of the main object. A potential main object in Cf certainly is the class `Request` and requests must be sent over an endpoint. This might not be obvious to a novice developer and since the class `Endpoint` is in another Java package, it might take a long time until the developer has figured out how to do it. Instead, Stylos et al. propose to add a send method directly to the request object. This method automatically finds the default endpoint of the application and sends itself over it.

Static construction methods: Constructors of a class all must have the same name, which brings some limitations. For instance, each sequence of argument types can only occur once. This is not the case for static methods that construct an object. Furthermore, the method names can indicate in which case it should be used. For this purpose, the factory pattern is often used. Stylos et al. have evaluated the usability of constructors and factories and propose to place the static methods directly in the class of the objects they construct [16]. For example, the class for empty messages provides the methods `newRST(Message)` and `newACK(Message)` to create a new ACK or RST for a specified message. This is better understandable than a constructor `new EmptyMessage(Message)` where it is not clear, whether it constructs an ACK, RST or something else.

Enforce noninstantiability (item 4): Some classes only contain static constants, enums, and methods, e.g., our class `CoAP`. With a private constructor, it is not possible to create an instance of that class which would be useless anyway.

Favor composition over inheritance (item 16): Inheritance explicitly unites the interface of a subclass with its superclass. The interface of a class not only consists of its method declarations but also the documented guarantees they promise. If a subclass is not able to provide the guarantees of a potential superclass, it should not inherit it but use the composition design pattern, i.e., using an instance of the superclass as private variable. Java's class `Properties` inherits from `HashMap<Object, Object>` which violates this concept as it is not possible to use `Integers` as keys, for example, even though an `Integer` is an `Object`. Californium's class for properties therefore uses the composition pattern instead of extending Java's `Properties`. Another example is the response of the novice-friendly *API* which uses but does not extend the Cf-internal class for responses.

Prefer interfaces to abstract classes (item 18): Java knows no multi inheritance. Therefore, an interface is typically preferable so that a subclass is not limited to one superclass only. A skeleton for an implementation of an interface still can be provided with an additional abstract class. Californium's interfaces for Resources and Layers are good examples of this.

Use enums instead of int constants (item 30): Often, integers or strings are used as constants. However, when a developer uses an invalid value, the program still compiles and later fails at runtime. Enums provide type-safe constants and even provide an explicit list of possible values to the developer. In Californium, request and response codes are implemented as enums, for instance.

Use instance fields instead of ordinals (item 31): Enums provide a method ordinal() that returns their index in the enumeration. Since the order of enums easily could be changed, the ordinal might change and should therefore not be used in the code. Instead, an enum should expect an integer parameter in its constructor that explicitly specifies its value.

Check parameters for validity (item 38): Often, the range of values that is allowed to be stored to a field is restricted. For instance, CoAP dictates non-negativity and a maximum on many numerical values for options. When a developer tries to use invalid options, the mistake should be detected and reported as fast as possible.

Prefer two-element enum types to boolean parameters (item 40): A CoAP request can be either confirmable or not. Imagine a constructor that expects a boolean to specify this parameter: `new Request(GET, true)`. From this code, it is not clear what the `true` stands for. With an enum instead, we can make the code look much clearer: `new Request(GET, CONFIRMABLE)`.

Return empty arrays or collections, not nulls (item 43): Developers often write code that iterates over a list like this: `for (Foo foo:getFooList()) {...}`. If the method returns null, however, the program throws a `NullPointerException`. This pitfall is easily avoided by returning an empty list instead.

Prefer executors and tasks to threads (item 68): Executors provide a powerful interface to execute tasks. Executors can easily be exchanged and can implement sophisticated strategies to dynamically adjust the number of threads. They can integrate the functionality of a timer and efficiently manage scheduled tasks and normal tasks with the same task queue and the same threads. Using threads directly in the code has none of these benefits.

ConcurrentHashMap (item 69): Data structures such as `ConcurrentHashMap` are highly optimized and thread-safe. They support high concurrency without causing significant amount of synchronization.

8.2 Novice-Friendly Client API

The goal for our novice-friendly *API* is to abstract more from the underlying CoAP protocol and to provide the developer with a simple and easily understandable *API* that reminds more of a connection-oriented protocol. We also use a fluent interface which allows chaining methods and with which better readable code can be written. We have analyzed several CoAP and HTTP clients to adopt strengths and avoid weaknesses in client *APIs*. The most profound difference among client *APIs* is whether they provide asynchronous or synchronous methods or both. In case of a synchronous method, the return value is usually the response. Asynchronous methods typically return `void` but expect a function handle as parameter that will be invoked when a response has arrived. We decided to provide both: synchronous and asynchronous methods.

The following classes compose our *API* for CoAP clients:

- `CoapClient`: to send requests to a server
- `CoapHandler`: function handle to react to a response or failure
- `CoapResponse`: to access the response of a request
- `CoapObserveRelation`: represents a CoAP observe relation with a resource

With an instance of class `CoapClient`, the developer sends requests to a server. A client can be used to send multiple requests sequentially or concurrently. The `CoapClient` can be thought of as a connection to a specific URI. `CoapClient` provides the methods `get()`, `post(payload)`, `put(payload)`, `delete()`, and `observe()` which can be used to send a `GET`, `POST`, `PUT`, `DELETE` request to the specified URI or to establish an observe relation to the specified resource. All can be called asynchronously or synchronously. The following example shows how a `GET` and `POST` request can be sent synchronously to a server and how the content of the response can be obtained.

```java
CoapClient client = new CoapClient("coap://example.com:5683/example");
String content = client.get().getResponseText();
String content = client.post("some payload").getResponseText();
```

The following example shows how we can asynchronously send a `GET` request and define a `CoapHandler` which will be invoked when either a response arrives or if the exchange failes. A request fails if the server is not reachable and the request timeouts or if the server rejects the request. It is not a failure, if the server responds with a valid response that has an error code in it.

```java
client.get(new CoapHandler() {
  @Override public void responded(CoapResponse response) {
    String content = response.getResponseText();
    // do something
  }

  @Override public void failed() {
    System.err.println("Failed");
    // do something
  }
});
```

The `CoapObserveRelation` represents an observe relation to a CoAP resource. It can be used to cancel or refresh the relation and to obtain the last notification. The observe method expects a `CoapHandler` that will be invoked whenever a notification arrives. The following example shows how an observe relation is established and canceled again.

```
CoapObserveRelation relation = client.observe(
  new CoapHandler() {
    @Override public void responded(CoapResponse response) {
      String content = response.getResponseText();
      // do something
    }

    @Override public void failed() {
      System.err.println("Failed");
      // do something
    }
  });
// do something
relation.cancel();
```

8.3 Novice-Friendly Server API

We have also designed a simple *API* to construct the server and its resource tree. The following example shows how a server can be constructed.

```
Server server = new Server(port);
server
  .add(new ResourceBase("A")
    .add(new ResourceBase("A1")
      .add(new ResourceBase("A1_a"))
      .add(new ResourceBase("A1_b"))
      .add(new ResourceBase("A1_c"))
```

```
            .add(new ResourceBase("A1_d"))
        )
        .add(new ResourceBase("A2")
            .add(new ResourceBase("A2_a"))
            .add(new ResourceBase("A2_a"))
            .add(new ResourceBase("A2_a"))
            .add(new ResourceBase("A2_a"))
        )
    )
    .add(new ResourceBase("B")
        .add(new ResourceBase("B1")
            .add(new ResourceBase("B1_a"))
            .add(new ResourceBase("B1_b"))
        )
    )
    .add(new ResourceBase("C"))
    .add(new ResourceBase("D"));
server.start();
```

A service that wants to export its content as CoAP resource can either implement the resource from scratch and implement the interface `Resource` or subclass `ResourceBase`. A subclass of the latter can override the handle methods for `GET`, `POST`, `PUT`, or `DELETE` requests. The handle method has an object of type `CoapExchange` as parameter which represents the current exchange. It can be used to obtain the request payload and options, and to respond. The following is a simple example of a CoAP resource. The GET handle method responds with the text "hello world". The POST handle method invokes the method accept() which acknowledges the request to use a separate response. After some computation, it sends the response. The PUT handle method invokes the method changed() which triggers the resource to reprocess all requests from observing clients and to send a new notification. The DELETE handle method deletes the resource from the resource tree and sends a 4.04 (Not found) response to all observing clients.

```
public class ResourceExample extends ResourceBase {

    public ResourceExample(String name) {
        super(name);
    }

    @Override
    public void handleGET(CoapExchange exchange) {
        exchange.respond("hello world");
    }
```

```
  @Override
  public void handlePOST(CoapExchange exchange) {
    exchange.accept();

    if (exchange.getRequestOptions()
                .hasContentFormat(MediaTypeRegistry.TEXT_XML)) {
      String xml = exchange.getRequestText();
      // do something
      exchange.respond(ResponseCode.CREATED);
    } else {
      // do something
      exchange.respond(ResponseCode.CREATED);
    }
  }

  @Override
  public void handlePUT(CoapExchange exchange) {
    // do something
    exchange.respond(ResponseCode.CHANGED);
    changed(); // notify all observers
  }

  @Override
  public void handleDELETE(CoapExchange exchange) {
    delete();
    exchange.respond(ResponseCode.DELETED);
  }
}
```

Finally, we show an example of a resource that has a critical section in its POST handle method. There are two ways to guarantee mutual exclusion. First, we can use the keyword `synchronized`. Second, we can extend the class `ConcurrentResourceBase` and configure it in the constructor to use only a single thread to handle all requests. Obviously, the developer needs to use only one of these techniques but for the sake of example we use both. Note that in the POST handle function that we first accept the request and only then enter the critical section.

The GET handle method needs to send a request itself to be able to respond to a GET request. Therefore, it uses a `CoAPClient` which is created by the superclass in `createClient()`. This client uses the very same thread as the resource to execute its code. The GET handle method asynchronously sends a second request to another resource and returns. Notice that the origin request has not yet been responded at this point, which is fine. When the response for the second request arrives, the client invokes the method `responded()` which finally sends the response to the origin request. All executed by the single thread that the resource uses.

```
public class SingleThreadedResource extends ConcurrentResourceBase {

  public SingleThreadedResource(String name) {
    super(name, SINGLE_THREADED);
  }

  @Override
  public void handleGET(final CoapExchange exchange) {
    exchange.accept();

    CoapClient client = createClient("coap://example.com:5683/target");
    client.get(new CoapHandler() {

      @Override
      public void responded(CoapResponse response) {
        exchange.respond(response.getCode(), response.getPayload());
      }

      @Override
      public void failed() {
        exchange.respond(ResponseCode.BAD_GATEWAY);
      }
    });
  }

  @Override
  public void handlePOST(CoapExchange exchange) {
    exchange.accept();

    ResponseCode response;
    synchronized (this) {
      // critical section
      response = ResponseCode.CHANGED;
    }

    exchange.respond(response);
  }
}
```

9 Conclusion

We have presented an efficient and scalable server architecture to host CoAP-based IoT Cloud services. With the gained knowledge from an extensive survey in the literature for HTTP Web server designs, we have developed a three-stage architecture. The three stages connectors, endpoints, and resources can be independently tuned to achieve the optimal performance or desired concurrent environment. A novice-friendly API lets the developer productively create services that export their functionality over a RESTful interface to the outside world. The API allows the developer to asynchronously or synchronously send requests to CoAP endpoints and establish observe relations. Experts can access Cf's internal API to have full control over CoAP messages and the protocol processing chain. The layered stack structure and the separation of processing logic and state make Cf understandable, maintainable, and extensible.

To benchmark Californium, we have built CoAPBench, a generic and distributed benchmark tool for CoAP servers. We have shown that Cf scales smoothly up to four cores for SMP and achieves a throughput as high as 140,000 requests per second on our quad-core system. The throughput stays stable and the latency low, even when 10,000 clients concurrently communicate with the server. The old version of Californium was originally designed as single-threaded prototype and, despite belated integration of multi-threading, can hardly exploit more than two cores. We have compared the old and new version of Californium to five state-of-the-art HTTP servers. Apache Tomcat and Apache HTTP Server are from the fraction of multi-threaded architectures, Vert.x and Node.js are event-driven, and Grizzly is a hybrid. We distinguished between two use-cases. In the first scenario, few clients send many requests and, when using HTTP, can keep established connections alive. This might be the case, when clients reach the server over a proxy. The second is the scenario of a resource directory, where many clients send only a single request and do not want to keep a connection alive. With connections kept alive, Vert.x achieves the best performance about 15% below Cf, which is the highest of all the HTTP servers. With an increasing amount of concurrent clients, the management of all the connections becomes harder for all HTTP servers and their throughput declines. Without keeping connections alive, Tomcat appears to be the best HTTP solution. Due to the excessive number of messages that TCP requires to establish and terminate connections, HTTP performs about ten times lower than CoAP. Neither CoAP nor HTTP servers manage to utilize more than four cores on a many-core NUMA system, though. Californium still achieves the highest performance but does not scale well if 8 or 16 cores with high cross-node memory access times are available. The computing power of a NUMA system would only be beneficial if a service itself was able to utilize it.

Our quad-core machine turns out to have an almost perfect balanced hardware for Californium. Network card and memory both are fast enough to keep the processor busy. The same is true for HTTP with the keep-alive mechanism but not without it. In the latter case, no server, except for Apache HTTP Server, is able to significantly utilize the processor.

Future work

Duplicate detection is a bottleneck in Californium. Cf is able to receive an immense number of requests per second, which demands a huge amount of memory to store all the exchanges. In the future, we might consider relaxing the principle of remembering the whole exchange for each request and find a more economical solution. A radically different approach is the thin server architecture [25] in which backend services are merely in client role and devices in server role. This way, the bulk of message deduplication is distributed and Californium unburdened.

Currently, Californium implements IETF's Proposed Standard for CoAP (coap-18) and the drafts for blockwise transfer (block-12) and resource observation (observe-08). Cf integrates resource discovery, HTTP and CoAP proxy support and DTLS. Currently, we are working on group communication and OSGi integration.

Bibliography

[1] A. Abhari, A. Serbinski, and M. Gusic. Improving the Performance of Apache Web Server. In *Proceedings of the 2007 Spring Simulaiton Multiconference (SpringSim 2007)*, Norfolk, Virginia, USA, 2007.

[2] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805, 2010.

[3] V. Beltran, D. Carrera, J. Torres, and E. Ayguade. Evaluating the Scalability of Java Event-Driven Web Servers. In *Proceedings of the 33rd International Conference on Parallel Processing (ICPP 2004)*, Montreal, Quebec, Canada, 2004.

[4] V. Beltran, J. Torres, and E. Ayguade. Understanding Tuning Complexity in Multithreaded and Hybrid Web Servers. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, Miami, Florida, USA, 2008.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), 1996.

[6] J. Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[7] C. Bormann, A. Castellani, and Z. Shelby. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.

[8] C. Bormann and Z. Shelby. Blockwise Transfers in CoAP. I-D: draft-ietf-core-block-12, 2013.

[9] C. Buckl, S. Sommer, A. Scholz, A. Knoll, A. Kemper, J. Heuer, and A. Schmitt. Services to the Field: An Approach for Resource Constrained Sensor/Actor Networks. In *Proceedings of the 23rd International Conference on Advanced Information Networking and Applications Workshops (WAINA 2009)*, Bradford, England, 2009.

[10] D. Carrera, V. Beltran, J. Torres, and E. Ayguade. A Hybrid Web Server Architecture for E-Commerce Applications. In *Proceedings of the 11th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2005)*, Fuduoka, Japan, 2005.

[11] J. Charzinski. Http/tcp connection and flow characteristics. *Performance Evaluation*, 42(2):149–162, 2000.

[12] G. S. Choi, J.-H. Kim, D. Ersoz, and C. R. Das. A Multi-Threaded PIPELINED Web Server Architecture for SMP/SoC Machines. In *Proceedings of the 14th International World Wide Web Conference (WWW 2005)*, Chiba, Japan, 2005.

[13] M.-J. Choi, H.-T. Ju, H.-J. Cha, S.-H. Kim, and J.-K. Hong. An Efficient Embedded Web Server for Web-Based Network Element Management. In *Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS 2000)*, Nara, Japan, 2000.

[14] F. Corazza. *A Smart City Infrastructure*. Master's thesis, Department of Computer Science, ETH Zurich, Switzerland, 2012.

[15] K. Cwalina and B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley Professional, 2nd edition, 2008.

[16] B. Ellis, J. Stylos, and B. Myers. The Factory Pattern in API Design: A Usability Evaluation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, Minnesota, USA, 2007.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[18] R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *Transaction on Internet Technology*, 2(2):115–150, 2002.

[19] S. Genevès. An Analysis of Web Servers Architectures Performances on Commodity Multicores. Research report, LIG, Grenoble University, France, 2012.

[20] A. S. Harji, P. A. Buhr, and T. Brecht. Comparing High-Performance Multi-Core Web-Server Architectures. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR 2012)*, Haifa, Israel, 2012.

[21] K. Hartke. Observing Resources in CoAP. I-D: draft-ietf-core-observe-10, 2013.

[22] S. Jucker. *Securing the Constrained Application Protocol*. Master's thesis, Department of Computer Science, ETH Zurich, Switzerland, 2012.

[23] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proceedings of the 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2011)*, Valencia, Spain, 2011.

[24] M. Kovatsch, M. Lanter, and S. Duquennoy. Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications. In *Proceedings of the 3rd International Conference on the Internet of Things (IoT 2012)*, Wuxi, China, 2012.

[25] 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 *Proceedings of the 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS 2012)*, Palermo, Italy, 2012.

[26] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg. Implementation of CoAP and its Application in Transport Logistics. In *Proceesings of the 10th International Conference on Information Processing in Sensor Networks (IPSN 2011)*, Chicago, Illinois, USA, 2011.

[27] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES 2001)*, Snow Bird, Utah, USA, 2001.

[28] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC 6202 (Informational), 2011.

[29] A. Mourad and H. Liu. Scalable Web Server Architectures. In *Proceedings of the 2nd IEEE Symposium on Computers and Communications (ISCC 1997)*, Los Alamitos, California, USA, 1997.

[30] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: an Efficient and Portable Web Server. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC 1999)*, Monterey, California, USA, 1999.

[31] M. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. Grieco, G. Boggia, and M. Dohler. Standardized Protocol Stack for the Internet of (Important) Things. *IEEE Communications Surveys & Tutorials*, 15(3):1389–1406, 2013.

[32] 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.

[33] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. Comparing the Performance of Web Server Architectures. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys 2007)*, Lisbon, Portugal, 2007.

[34] J. Postel. User Datagram Protocol. RFC 768 (Internet Standard), 1980.

[35] J. Postel. Transmission Control Protocol. RFC 793 (Internet Standard), 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[36] T. Potsch, K. Kuladinithi, M. Becker, P. Trenkamp, and C. Goerg. Performance Evaluation of CoAP Using RPL and LPL in TinyOS. In *Proceedings of the 5th International Conference on New Technologies, Mobility and Security (NTMS 2012)*, Istanbul, Turkey, 2012.

[37] A. Rahman and E. Dijk. Group Communication for CoAP. I-D: draft-rahman-core-groupcomm-07, 2011.

[38] A. Rahman and E. Dijk. Group Communication for CoAP. I-D: draft-ietf-core-groupcomm-16, 2013.

[39] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), 2012.

[40] M. Serrano. Hop, a Fast Server for the Diffuse Web. In *Proceedings of the 11th International Conference on Coordination Models and Languages (COORDINATION 2009)*, Lisboa, Portugal, 2009.

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

[42] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). I-D: draft-ietf-core-coap-18, 2013.

[43] J. Stylos and S. Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, Minnesota, USA, 2007.

[44] J. Stylos and B. Myers. Mapping the Space of API Design Decisions. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2007)*, Coeur d'Alene, Idaho, USA, 2007.

[45] J. Stylos and B. A. Myers. The Implications of Method Placement on API Learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2008/FSE-16)*, Atlanta, Georgia, 2008.

[46] R. von Behren, J. Condit, and E. Brewer. Why Events are a Bad Idea (for High-Concurrency Servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HOTOS 2003)*, Lihue, Hawaii, 2003.

[47] M. Welsh, D. Culler, and E. Brewer. SEDA: an Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, Banff, Alberta, Canada, 2001.