CoAP infrastructure for IoT

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Heng Shi

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

The Internet of Things (IoT) can be seen as a large-scale network of billions of smart devices. Often IoT devices exchange data in small but numerous messages, which requires IoT services to be more scalable and reliable than ever. Traditional protocols that are known in the Web world does not fit well in the constrained environment that these devices operate in. Therefore many lightweight protocols specialized for the IoT have been studied, among which the Constrained Application Protocol (CoAP) stands out for its well-known REST paradigm and easy integration with existing Web. On the other hand, new paradigms such as Fog Computing emerges, attempting to avoid the centralized bottleneck in IoT services by moving computations to the edge of the network. Since a node of the Fog essentially belongs to relatively constrained environment, CoAP fits in well. Among the many attempts of building scalable and reliable systems, Erlang as a typical concurrency-oriented programming (COP) has been battle tested in the telecom industry, which has similar requirements as the IoT. In order to explore the possibility of applying Erlang and COP in general to the IoT, this thesis presents an Erlang based CoAP server/client prototype ecoap with a flexible concurrency model that can scale up to an unconstrained environment like the Cloud and scale down to a constrained environment like an embedded platform. The flexibility of the presented server renders the same architecture applicable from Fog to Cloud. To evaluate its performance, the proposed server is compared with the mainstream CoAP implementation on an Amazon Web Service (AWS) Cloud instance and a Raspberry Pi 3, representing the unconstrained and constrained environment respectively. The ecoap server achieves comparable throughput, lower latency, and in general scales better than the other implementation in the Cloud and on the Raspberry Pi. The thesis yields positive results and demonstrates the value of the philosophy of Erlang in the IoT space.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
ACK	Acknowledgement
ATM	Asynchronous Transfer Mode
AWS	Amazon Web Sevice
AMPED	Asynchronous Multi- Process Event-Driven
API	Application Programming Interface
BLE	Bluetooth Low Energy
CoAP	Constrained Application Protocol
CoRE	Constrained RESTful environments
CON	Confirmable
CPU	Central Processing Unit
DDS	Data Distribution Service
DCPS	Data-Centric Publish-Subscribe
DLRL	Data Local Reconstruction Layer
DICE	DTLS In Constrained Environments
DTLS	Datagram Transport Layer Security
ELIoT	ErLang for the Internet of Things
GDS	Global Data Space
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IoT	Internet of Things
JVM	Java Virtual Machine
LWM2M	OMA Lightweight M2M
M2M	Machine-to-Machine
MID	Message Identifier
MT	Multi-Threaded
MTU	Maximum Transmission Unit
MP	Multi-Process
MQTT	Message Queue Telemetry Transport
NAT	Network Address Translation
NIC	National Intelligence Council
NON	Non-confirmable
OTP	Open Telecom Platform
OMA	Open Mobile Alliance
PID	Process Identifier
PIPELINED	Multi-Threaded Pipelined
QoS	Quality of Service
RD	Resource Directory
REST	Representative State Transfer
RFID	Radio-Frequency Identification
RST	Reset
RTPS	Real-Time Publish-Subscribe
SEDA	Stage Event-Driven Architecture
SMS	Short Message Service
SoC	System on Chip
SPED	Single-Process Event-Driven
TCP	Transmission Control Protocol

TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
vCPU	virtual CPU
VM	Virtual Machine

Chapter 1

Introduction

With the rise of the Internet of Things (IoT), more smart devices (sensors, actuators, etc.) will be deployed in people's lives everywhere. A large number of these devices brings up new challenges to applications.

Traditional Cloud-based solution may not be enough for IoT applications with low-latency requirements. Thus new concepts and paradigms such as Fog Computing have been proposed, which moves computation gradually to the edge of the network and enables even larger scale network of devices to be involved smoothly. The Fog usually consists of many distributed nodes that sit between devices with very constrained resources like sensors and actuators and platform which is more unconstrained and resource-rich, such as the Cloud. The fact that a Fog is not as powerful as a Cloud but often executes some complex tasks on behalf of many low-end devices puts some performance demand on it, similar to traditional backend services. Fog and Cloud can be combined together to satisfy the complex requirements of the IoT.

Another challenge is that existing communication protocols do not fit well in the IoT space. Typical IoT applications are intended to consist of many resource-constrained devices such as sensors, data collectors, actuators, controllers or other embedded devices. Many of them will not be able to handle complex protocols solely relying on their own limited resources. Network bandwidth is also limited in the constrained environment. Therefore popular Web protocols like the Hypertext Transfer Protocol (HTTP) should be reconsidered when it comes to IoT applications due to the overhead. To smooth this problem, many new protocols emerged, including but not limited to the Constrained Application Protocol (CoAP) [89], Message Queue Telemetry Transport (MQTT) [10] and the Data Distribution Service for real-time systems (DDS) [34]. Among them, CoAP, as a lightweight and efficient application layer protocol, targets the problem by using low-overhead UDP as its transport layer to ease the stress in the constrained environment. It also embraces the Representational State Transfer (REST) [45] architectural style so that interactions with existing Web becomes easier. Beyond that, CoAP supports asynchronous message notification, a.k.a. resource observation, to fit in the subscriber model that is widely used in many IoT scenarios. The above features enable CoAP as a competitive candidate for the wire protocol for both Cloud and Fog based IoT applications.

Regardless of paradigms and protocols, IoT applications in general place high requirements on infrastructures. Scalability and reliability become more important. One constrained device may not be able to send large volume data at once or send data very frequently, but millions of them could result in huge amount of ongoing traffic, which requires more scalable backends. It is desired that the software could scale on demand.

On the other hand, as more cyber-physical or mission-critical systems, such as industrial control systems, smart cities, and connected cars, are connected to the IoT, the uninterrupted and safe operation is often the top priority [25]. In another word, downtime due to the failure of subsystems should be minimized. This is where reliability must be emphasized more than ever.

An industry with totally different targets but similar requirements is telecommunication since a large amount of in and out phone calls must be handled concurrently in a system with very low downtime. The high concurrency implies a system which could scale up and down while the low downtime implies a fault-tolerant system with high availability, hence reliability. Back to 1980s, one of the top telecom equipment manufacturers, Ericsson, attempted to solve the problem by introducing a new programming environment called Erlang [38]. Erlang approached the problem by following the famous actor model [3], which models the desired system as a combination of many independent, isolated, concurrent actors communicating only through messages. Erlang leads to a new programming paradigm called concurrency-oriented programming (COP) [8] and has influenced many subsequent languages such as Go [50] and Scala [87].

Erlang is proved to be suitable for building massively scalable soft real-time systems with requirements on high availability. As an instance of Erlang's application in the real world, the AXD301 is a fault-tolerant carrier-class ATM switch manufactured by Ericsson Telecom AB, which has the measured reliability quoted as being 99.9999999% (9 nines) corresponding to a downtime of 31 ms per year [8]. Nevertheless, very few research of applying Erlang or COP in general in the IoT area have been presented. Sivieri et al. [91] proposed an Erlang-based development framework called ELIoT which aims at the coordination of wireless sensor network. Hiesgen et al. [55] introduced a modified actor model and corresponding runtime environment built with C++ to IoT application programming. The authors listed Erlang as a design reference and gave an example implementation utilizing CoAP as the communication channel among actors.

The similarity between the requirements of IoT applications and the design goal of Erlang leads us to this research. In order to explore the use space of COP language in the IoT world, this work attempted to use the typical COP language Erlang to model a CoAP infrastructure (server and client) and demonstrate a prototype implementation called ecoap, aiming at scalability and reliability in both constrained (Fog) and unconstrained (Cloud) environment.

Subsequent chapters are organized as follows. chapter 2 gives a more precise definition of the research problem. chapter 3 presents the background and state-of-the-art work, including a comparison of popular IoT application protocols, an introduction of important features of CoAP and summary of existing CoAP implementations. It is followed by chapter 4, which firstly argues the difference between the popular paradigms of designing concurrent server-side software and typical Erlang applications, and then presents the proposed architecture and implementation details. After that chapter 5 demonstrates a scalability benchmark conducted on both constrained and unconstrained platform (in comparison to the Californium (Cf) CoAP framework [18], a popular Java CoAP implementation) and a fault-tolerance benchmark against ecoap itself. In the end, chapter 6 shows the limitations of the research, the conclusion, and contribution as well as future work.

Chapter 2

Problem Statement

The main objective of this work is to answer the following question:

How could we use a typical concurrency-oriented programming language like Erlang to model scalable and reliable IoT infrastructure utilizing CoAP?

Moreover, with Fog Computing paradigm in mind, exploring how the same Erlang-based solution could *scale down* to a constrained environment such as single-board platform and *scale up* to an unconstrained environment such as cloud is another research interest here.

The objective of the research can be further divided to following sub-questions:

- How can the Constrained Application Protocol be implemented in the context of Erlang?
- Following idiomatic Erlang design patterns, what architecture should the CoAP implementation use to provide both scalability and reliability?
- What interface should be provided to application developers for easy integration?
- How can a benchmark tool be constructed to evaluate the implementation?

Since scalability and reliability are the desired goals. They are specified in the scope of this work as follows:

- Scalability is the ability of a system, network, or process to adapt itself to handle a growing amount of work [12]. A system is claimed to be scalable if it gains improvements in its performance under an increased load when more resources are added (typically hardware). When it comes to multiprocessor computing, measurements of scalability can be generally classified into two groups, horizontal scalability (or *scaling out/in*) and vertical scalability (or *scaling up/down*) [71]. Horizontal scalability refers to adding more nodes to (or removing nodes from) a system, such as adding a new computer in a distributed application. While vertical scalability means adding resources to (or removing resources from) a single node in a system, typically involving the addition of CPUs or memory capacity to a single computer.

  It is particularly interesting to evaluate horizontal scalability in IoT scenarios, especially when Erlang comes into play since it is famous for its transparent distribution support. However, horizontal scalability heavily depends on application-specific requirements, which makes it difficult to model and

test generally. Therefore, for simplicity vertical scalability is primarily considered in this work. It is measured by observing how many concurrent requests a system can handle on average under increasing load, and how this behaviour changes when more resources are added to the underlying platform.

- Reliability can refer to many aspects. In this work, availability is the principal concern. In general, availability is the proportion of time a system is in a functioning condition. It is measured qualitatively by observing whether a system remains available and behaves as expected when random faults occur within any of its sub-systems. This is sometimes also referred to as fault-tolerance. Fault-tolerance and reliability are used interchangeably in this work.

As a summary, in a client-server model, a scalable and reliable server means it should fully utilize modern multi-core systems, deliver stable performance facing a large number of concurrent clients/requests and behave as expected when faults occur.

Chapter 3

Related Work

3.1 Background

3.1.1 IoT and Fog Computing

Recently, researchers have shown an increased interest in the Internet of Things (IoT). The IoT aims at providing an intelligent environment where ubiquitous goods in the the physical world can be connected to the Internet through unique addressing schemes so that they can interact with each other and cooperate with their neighbours to reach common goals without human involvement [9]. It is enabled by a variety of technologies including Radio-Frequency Identification (RFID), smart sensors and actuators, mobile computing and Internet protocols [47]. Potential application areas in IoT range from transportation and logistics, healthcare, social networking, smart environment (home, office, plant), industrial automation, to emergency response to disasters and more other areas [9, 47]. The US National Intelligence Council (NIC) considers the IoT as one of the six "Disruptive Civil Technologies" that will have an impact on US National Power and indicates that "by 2025 Internet nodes may reside in everyday things – food packages, furniture, paper documents, and more" [1]. It is estimated that 28.1 billion IoT devices will be connected which brings a 7.1 trillion dollars worldwide market by the end of 2020 [70].

The vision of IoT has foreseen many common challenges, among which, scalability and reliability are more critical than ever [9, 47, 101]. Over the past decade, the industry has seen a trend of moving computing and storage resource behaviourinto the Cloud. It is straightforward to integrate the IoT with the Cloud to address the scalability and reliability issue. However, the emerging IoT introduces many new challenges that can not be adequately addressed by today's Cloud Computing models alone. These challenges [25, 103] include but are not limited to, stringent latency requirements under certain environment (such as industrial control systems), inflexibility of the Cloud supporting rapid mobility patterns, network bandwidth limitation due to rapid growing number of connected things, difficulty for resource-constrained devices to interact with the Cloud using complex protocols, applications which require uninterrupted services but with intermittent connectivity to the Cloud, and security concerns caused by the combinations of one or more issues mentioned above.

In order to fill the technology gaps in supporting the IoT, a new architecture - Fog Computing - was first

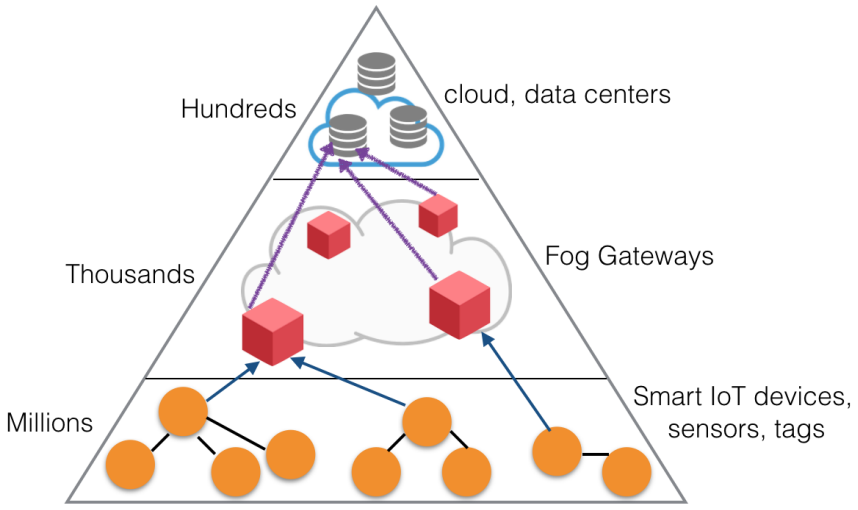


Figure 3.1: The role of the Cloud and Fog play in the delivery of IoT services. [47]

introduced. Fog is an architecture where computation, networking and storage are arranged closer to end devices instead of inside Cloud data centres [14]. Fog consists of heterogeneous, potentially wide-spread and geographically distributed networks of nodes, where a node can be any smart edge device or system that acts as a local intelligence for all related sensors and actuators. A Fog in each location can be as small as a single node or as large as required to meet customer demands and many small Fog nodes may constitute a large Fog system [25].

The primary idea behind Fog Computing is to put the force of analyzing and processing data to applications hosted by devices within the network instead of to a centralized Cloud [103]. As a result, it enables IoT applications which require low-latency, location-awareness, real-time interactions and analytics and mobility support. Fog also enhances the scalability and resilience of IoT applications by its nature [47]. Various services and computing tasks can run on a Fog node, depending on how much resource a node owns and application-specific requirements. Applications could on the one hand let local Fog carry out resource-intensive tasks (M2M interaction, data collection and processing, actuator control, etc.) on behalf of resource-constrained devices when such tasks cannot be moved to Cloud due to latency constraints or any other reasons, on the other hand expect the preprocessed data to be consumed by higher tiers, which can be other Fog or the global Cloud, for long-term analysis and storage [25]. Figure 3.1 illustrates the roles that the Cloud data centres and the Fog play to deliver IoT services to end-users.

Fog Computing has the potential to increase the overall performance of IoT applications. Scenarios where Fog could help include low-latency applications such as gaming and video conferencing, geo-distributed applications such as pipeline monitoring and wireless sensor networks, fast mobile applications such as connected vehicle and connected rail, and large-scale distributed control systems such as smart grid and smart traffic light systems [13]. A real-world success of Fog has been discussed in [25], where Barcelona utilized Fog as

a uniform platform for all services in their smart city management applications and reduced overall system costs.

Fog Computing provides a new application scenario for this work apart from the Cloud. Scalability and reliability are still required on a Fog node especially when complex tasks and services are deployed. The Fog node is highly possible a less-constrained embedded device such as the Raspberry Pi [83], which has more resources than ordinary sensors and actuators yet is much less powerful than the Cloud. Moreover, since the Constrained Application Protocol (CoAP) is a standard machine-to-machine (M2M) protocol, it surely has its use space under a Fog environment. It is therefore considered the combination is not only a valid use case in terms of Fog Computing, but also falls within the scope of this thesis.

3.1.2 Common Patterns of IoT Applications

Some of the architectural patterns that are common in other information systems can be applied to the design of IoT applications as well. This section introduces three most widely used ones: REST, publish/subscribe and observer pattern.

REST

The Representational State Transfer (REST) was first defined by Fielding [45] in his doctoral dissertation in 2000. It is an architectural style that defines constraints for creating Web services, and its principles were used in designing the HTTP 1.1 [46] and Uniform Resource Identifiers (URI) [11] standards. Web services that follow the REST style are claimed as RESTful, which provide interoperability to a majority of computers connected to the Internet nowadays. Though REST is not tied to any particular technology, HTTP, which has four basic operations (HTTP verbs): GET, POST, PUT and DELETE, is commonly used as its pattern matches REST well. In general, REST defines the behaviour of a well-designed Web service, which should consist of a network of Web resources that the user could go through via corresponding links and applying operations such as the HTTP verbs to0. The operations trigger state transitions of the resource and as a result, the next resource is transferred to the user.

Some of the important characteristics of REST are:

- REST relies on the client-server hence request-response model.
- REST identifies resources using URIs.
- REST uses stateless operations upon resources.
- REST separates resources from their representations.

Modern Web applications extensively use REST since its semantics naturally allows for caching, proxying and redirecting requests/responses between Web resources distributed over loosely coupled Internet endpoints, resulting in a highly flexible, scalable, reliable and interoperable architecture.

Publish/Subscribe

Publish/subscribe is a messaging pattern that provides loosely coupled interactions. The receivers of messages, called subscribers, could specify their interest in messages of certain patterns and are subsequently notified asynchronously of any message that is produced by the senders of messages, called publishers when the message matches the registered interest [104]. Since subscribers only receive a subset of all published messages, publish/subscribe systems differ in the way of selecting the messages of interest and can be primarily classified to following types: topic-based, content-based and type-based [44]. In a topic-based system, messages are classified by topics and subscribers only receive messages published to the topics that they subscribe. In a content-based system, messages are delivered to subscribers if they match certain constraints the subscribers to define over the properties of the message and are not pre-classified. In a type-based system, messages are also filtered based on the type or class, which can be defined, for example, using object-oriented languages.

Publish/subscribe pattern has the following advantages:

- Space decoupling: Publisher and subscriber do not need to know each other.
- Time decoupling: Publisher and subscriber do not need to be active at the same time.
- Synchronization decoupling: Publisher and subscriber do not block each other when publishing and receiving messages.

A publish/subscribe service can be implemented with a centralized broker, which stores and manages subscriptions. It can also be realized in a distributed manner, where the management work is implemented both in publishers and subscribers. The former brings stronger reliability while the latter is more scalable [44].

Observer Pattern

Observer pattern [48] is a term used more commonly in object-oriented software designs.

It defines the relationship of a "subject" and one or more "observers", where a subject has the state of interest and maintains a list of all dependencies, called observers, who register themselves at the subject and are interested in being notified whenever the state undergoes a change. The observer pattern itself shares similarities with publish/subscribe pattern. The main difference is that in observer pattern the subject is aware of the observers and maintains states about them, while in publish/subscribe pattern both parties of the interaction are decoupled.

In summary, all the above patterns can be useful when designing IoT applications. the REST architectural style essentially brings what makes the current Web robust to the IoT field, so that an IoT application exposing as a RESTful service can easily reuse existing Web capabilities. The publish/subscribe pattern is extremely useful when the producer and consumer of IoT services are not synchronous by their nature. The

observer pattern, being a variation of the publish/subscribe pattern, is used by CoAP to realize resource state observation, which does not fit naturally with the stateless REST style.

3.1.3 A Brief Introduction to Erlang/OTP

This section contains a short summary of concepts of the Erlang language that are important to this work.

Erlang is a single assignment, dynamically typed functional language. Single assignment means that a variable gets bound to a particular value and can not be reassigned to a different value. In Erlang, a variable starts with a capital letter. Some of the basic data types of Erlang are:

- Term: A term is of any data type.
- Number: A number can an integer, e.g. `5`, or a float, e.g. `1.5`.
- Atom: An atom is a literal constant with a name. It is to be enclosed in single quotes if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscores, or at signs. e.g. `red`, `a_long_name`. Note there is no boolean type and `true`, `false` are of atom type.
- Tuple: A tuple is a compound data type with a fixed number of terms. e.g. `{john, {july, 24}}`.
- List: A list of a compound data type with a variable number of terms. e.g. `[a,2, {c, 4}]`.
- String: A string is enclosed in double quotes, and is actually a list of integers where each integer represents a Unicode codepoint. e.g. `"Hello"`.
- Bit string and binary: A bit string is used to store an area of untyped memory. Bit strings that consist of a number of bits that are evenly divisible by eight, are called binaries. e.g. `<<"ABC">>`, `<<1:1,0:1>>`.
- Record: A record is a data structure for storing a fixed number of elements and is to be transferred to tuple during compilation. Example of defining a record: `-record(person, {name, age})`. Example of initializing a record: `#person{name=Name, age=Age}`.
- Map: A map is a compound data type with a variable number of key-value associations. Key can be any valid Erlang term. e.g. `#{name=>adam, age=>24, date=>{july, 29}}`
- Pid: A pid is a unique process identifier. e.g. `<0.100.0>`.
- Reference: A reference is a globally unique Erlang term created using the built-in functions.

Listing 1 shows a sequential program which calculates the factorial of `N`. It starts with a module definition and a list of exported functions. Erlang programs are organized through files called modules each containing a sequence of function declarations. Each function declaration is a sequence of function clauses separated by semicolons, and terminated by a period `(.)`. A function clause consists of a clause head and a clause body, separated by `->`, and an optional guard sequence beginning with the keyword `when`. A clause head consists

```erlang
-module(math).
-export([fac/1]).

fac(N) when N > 0 -> N * fac(N-1);
fac(0) -> 1.
```

Listing 1: Erlang factorial example

of the function name and an argument list. The function name is an atom. Each argument is a pattern. The number of arguments N is the arity of the function. A function is uniquely defined by the module name, function name, and arity. A clause body consists of a sequence of expressions separated by a comma `(,)`. A function named `f` in the module `m` and with arity `N` is often denoted as `m:f/N`, while an internal function can be referred to without specifying the module name. The example defines a `math` module which exports a function named `fac` that has one argument. The `fac/1` function consists of two clauses. The first clause is a pattern that matches `N>0` and executes recursively until hitting the base case, that is, `fac(0)`. When another pattern occurs, the function clauses are scanned sequentially until one pattern matches otherwise the evaluation fails.

```erlang
-module(counter).
-export([start/0, stop/1, add_number/2, get_number/1]).

start() ->
    spawn(fun() -> loop(0) end).

add_number(Pid, M) ->
    Pid ! {add_number, M},
    ok.

get_number(Pid) ->
    Pid ! {get_number, self()},
    receive
        {counter, N} -> N
    end.

stop(Pid) ->
    Pid ! stop,
    ok.

loop(N) ->
    receive
        {add_number, M} -> loop(N+M);
        {get_number, From} -> From ! {counter, N}, loop(N);
        stop -> ok
    end.
```

Listing 2: Erlang messaging example

Listing 2 shows a concurrent program where an Erlang process can be spawned which holds a counter while other processes can interact with it. By calling `counter:start/0` a process is created which executes the `loop/1` function, and the process identifier (PID) is returned. `counter:add_number/1` accepts the Pid and a number `M` as the argument, then sends a message `{add_number, M}` to the counter process using the send operator `!`. The counter process uses `receive ... end.` syntax to pattern match received messages. If a message matches the corresponding clause is evaluated. If no clause matches the message will be left in the message box of the process. In the example, the above function call will make the counter process add the number `M` together with the number it gets as its initial state, `N`, and use the result as the new state for the next recursion loop. Similarly, `counter:get_number/1` will trigger the counter process send back its state to the caller and `counter:stop/1` will cause the counter process to exit the loop and hence terminate itself.

Real world Erlang applications seldom use plain Erlang extensively. Instead, the Open Telecom Platform (OTP) is preferred. It is a set of libraries and procedures for building scalable and fault-tolerant applications. The core concept of OTP is OTP behaviour. A behaviour encapsulates common behavioural patterns. It is driven by a parameterized callback module. In a word, the behaviour solves the nonfunctional parts of the problem, while the callback solves the functional parts. Most commonly used OTP behaviours include but are not limited to `gen_server` which provides generic client-server abstraction, `gen_statem` which provides a skeleton for the finite state machine, the `supervisor` that describes the supervision architecture, etc. It is also possible to create user-defined behaviour. More details about behaviours used in this work can be found in section 4.2.

3.2 Application Layer Protocols for IoT

Because of the fact that the IoT consists of a considerable amount of devices with vastly different capabilities, standards are in need to support interoperability between newer applications and services with existing hosts and IoT nodes [26]. Among the many standards introduced facilitating interoperability, the Internet Protocol (IP) and application layer protocols are of the most importance, since the former enables devices to be part of the current Internet and the latter directly drives the development of different IoT services.

In general, current IoT application protocols can be divided into 3 types: message-oriented, data-oriented, and resource-oriented [90]. Representative protocols of the 3 types are Message Queue Telemetry Transport (MQTT) [10], the Data Distribution Service for real-time systems (DDS) [34] and the Constrained Application Protocol (CoAP) [89]. In this section, these protocols are discussed and compared in terms of their architecture and use cases. Reason for choosing CoAP as the protocol used in this work is also stated.

3.2.1 Message-oriented: Message Queue Telemetry Transport (MQTT)

Message Queue Telemetry Transport (MQTT) [10] is a messaging protocol first introduced by IBM in 1999 and was later standardized by OASIS in 2013. It relies on a topic-based publish-subscribe architecture

with a central message-broker that links the publishers and subscribers, as shown in Figure 3.2. It inherits advantages of publish/subscribe pattern, that is, decoupling of time, space and synchronization of both message producers and consumers.

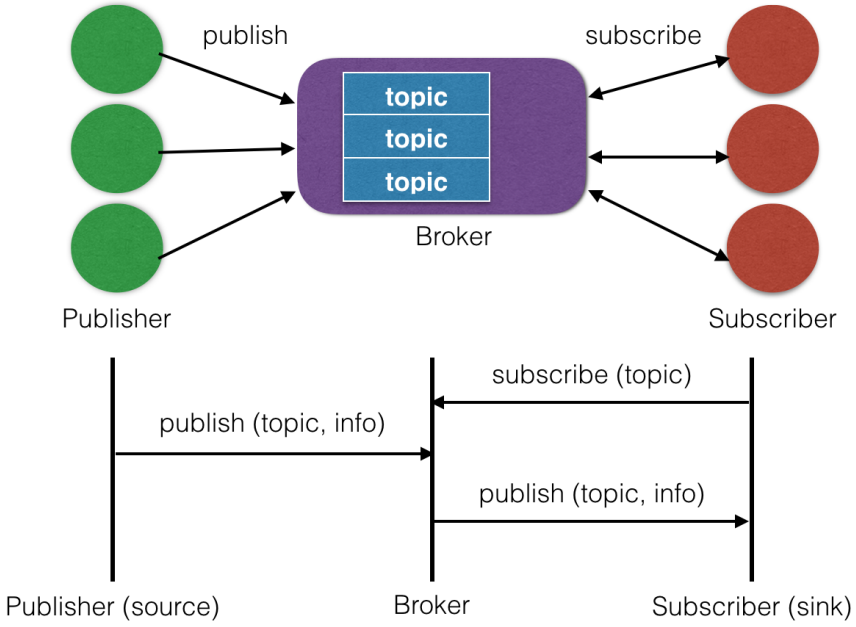


Figure 3.2: The architecture of MQTT and its publish/subscribe process

A typical workflow includes an interested device registering as a subscriber for specific topics, then when the publisher generates data, the subscriber would be notified by the broker and receives such data through the broker. In the process, the broker is also responsible for authorization checking. MQTT employs a flexible routing mechanism (one-to-one, one-to-many, many-to-many) for connecting embedded devices and networks with applications and middleware [47]. It provides 3 levels of Quality of Service (QoS) for message delivery with only a small transport overhead [10].

MQTT is a connection-oriented protocol that runs on top of TCP/IP. The fact that MQTT requires persistent connections to the message broker for both publisher and subscriber results in increasing communication cost, which further makes it unfeasible for wireless sensor networks. In order to avoid such drawbacks, MQTT-SN (formerly MQTT-S [58]) was developed, which is an extension of MQTT that is agnostic of the underlying networking service and therefore can run on non-TCP/IP networks [95].

Under the message-oriented paradigm, MQTT stands out as a reliable and flexible protocol for message distribution due to its simplicity and efficiency, ensuring small footprint and low power consumption on embedded devices [104].

3.2.2 Data-oriented: Data Distribution Service (DDS)

The Data Distribution Service for real-time systems (DDS) is a publish-subscribe protocol for real-time M2M communications developed by Object Management Group (OMG) [34], with an origin in defence and aerospace domains. It applies a data-oriented/data-centric paradigm, which facilitates finely grained data specification by allowing for the modification of numerous QoS parameters, including but not limited to how long a specific piece of data is valid as well as the rate of subscription and publication [80].

DDS primarily consists of two components, namely the Data-Centric Publish-Subscribe (DCPS), which is the lower layer API for interacting with other DDS driven applications, and the optional Data Local Reconstruction Layer (DLRL), that is the upper layer specifying in what methods an application can process DCPS data fields using self-defined object-oriented programming classes [80]. It utilizes multicasting for ensuring outstanding QoS and high reliability. Similar to MQTT, it supports abundant QoS parameters (up to 23 policies), however, unlike MQTT, DSS is broker-less, which addresses the requirement for realtime-ness of IoT and M2M communications well [47]. The advantages of DSS are discussed in [80] in details, including its support for "auto-discovery" for new or stale endpoint applications, low overhead, dynamic scalability and flexible routing and configuration.

Note that OMG's DDS has no specification for the wire protocol. To address the incident where DDS products from different vendors are difficult to interoperate, Real-Time Publish-Subscribe protocol (RTPS) [98] was promoted by OMG as a wire-protocol for DDS. RTPS is based on UDP and multicast for the transmission of data between publishers and subscribers. The transmission throughput can be promising depending on which network topology and routers are used.

As the evaluation [41] for two DDS implementations pointed out, this protocol scales well with an increasing number of nodes. While protocols that are either inspired by or compatible with DSS have been proposed for wireless and sensor environments, there seems to be a lack of successful deployments, partly due to its complexity [90].

3.2.3 Resource-oriented: the Constrained Application Protocol (CoAP)

IoT applications usually consist of myriads of devices that have minimal unit costs, which means they are constrained in power supply, available memory footprint, processing capabilities and many other aspects. However, these constrained devices can surely benefit from a connection to the Internet as they can be integrated into distributed services. The Internet Engineering Task Force (IETF) has carried out a lot of standardization work to make it happen. The IPv6 over Low-Power Wireless Area Networks (6LoWPAN) standards [67, 57] now enable IPv6 on very constrained networks, which makes integration of sensors and actuators into the Internet seamless.

Nonetheless, Devices and services must likewise achieve interoperation at the application layer for full convergence [65]. Using HTTP for accessing numerous mashup services on the World Wide Web (WWW) is

the norm for the Internet and Web-based applications. When it comes to constrained environments, HTTP over TCP is too clunky as a result of its overhead in implementation code space and the fact that constrained networks are vulnerable to high packet error rates and lossy links [15]. To fill such gaps, a new Web protocol, the Constrained Application Protocol (CoAP) is proposed by IETF, in order to leverage the REST paradigm in these environments as well.

CoAP follows the REST style similar to HTTP but is customized for constrained devices and networks. It employs a compact binary format and UDP (or Datagram Transport Layer Security (DTLS) for security) as the underlying transport layer, which largely reduces complexity brought by TCP. It in general consists of a message layer that detects duplicates and provides optional reliable delivery of messages (stop-and-wait retransmission with exponential back-off), and a request/response layer that allows RESTful interactions with the Web much the same as HTTP does, that is, using well-defined verbs and response codes. As a result, CoAP can interwork with and be mapped to HTTP (and hence external services using HTTP) in a straightforward way, allowing hybrid application architecture as shown in Figure 3.3.

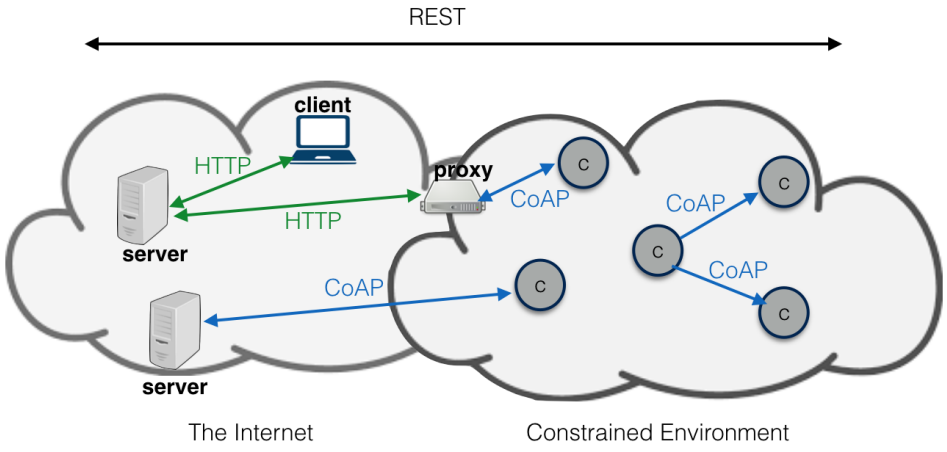


Figure 3.3: An application architecture combining CoAP and HTTP

However, CoAP is not merely a compressed version of HTTP [15]. It supports group communication through IP multicast. CoAP enabled devices can discover services exposed by each other via standardized interface and resource description based on the well-known resource path and Web Linking [76, 75, 88]. CoAP allows observable resources which pushes state updates to registered clients through continuous notifications in an asynchronous manner [53]. Resource with a large body that exceeds the maximum transmission unit (MTU) can be processed using block-wise transfer, which is essentially an application layer fragmentation mechanism [17]. Furthermore, alternative transports such as Short Message Service (SMS) is also feasible and is under active standardization development [86].

3.2.4 Comparison

MQTT is a message-oriented protocol which means that it is content agnostic and only focuses on the delivery of messages, while CoAP is all about operations against the representation of resource since it follows the REST style. Therefore, although from the perspective of implementation cost, CoAP has a wider acceptance for constrained devices due to its low overhead and better support for 6LowPAN, for less constrained devices that could afford TCP stack, MQTT is simpler to implement on the device side in terms of semantics. In addition, Zhou and Zhang [104] pointed out other disadafvantages of resource-oriented IoT protocols like CoAP, including overhead introduced to constrained devices in terms of concurrency, computing and networking, inflexibility that IP address of each device must be known, as well as Network Address Translation (NAT) issue for global accessibility is currently most common IPv4 subnet environment.

On the other hand, CoAP has a clear read/write semantics, which makes it not only easier to integrate with other Web-like services, but also possible to add infrastructure support for caching and proxying. MQTT's publish/subscribe architecture results in a non-intuitive way of communication when the subscriber also needs to send feedback of configuration command to the publisher since it is not the default message flow direction. Moreover, MQTT and CoAP differ a lot in terms of discovering services. With MQTT, it is necessary to first find the central server (broker) which all messages must go through. In contrast, CoAP can avoid the need for a central endpoint since it supports multicast to discover available services. The read/write semantics of CoAP then ensures independent communication endpoint to endpoint. As a result of the aforementioned difference, it requires less effort to scale CoAP than MQTT. Similarly, Kovatsch [63] wrote that the main drawback of MQTT is lack of extensibility, as MQTT clients have to be pre-configured with a dedicated service similar to HTTP clients for the IoT, which makes it difficult to adapt to an evolving environment.

It should be noted that resource observation in CoAP is similar to publish/subscribe pattern in MQTT at first glance, but unlike publish/subscribe, where the goal is to propagate every event, observe only guarantees that eventually, all registered observers will have a current representation of the latest resource state.

There are few research publications as well as industry deployments about DDS, which implies it may be still in exploration and optimization stage. On the other hand, there are known scalability [42] and performance [85] issues associated with RTPS, which is the wire protocol used in DDS. Also, DDS's Global Data Space (GDS) concept is of limited use when faced with unreliable, low bandwidth and high latency networks [90].

Table 3.1 provides a brief comparison between MQTT, DDS, CoAP and HTTP in terms of architecture and semantics. HTTP is also included here as a contrast to CoAP since they share many similarities. Table 3.2 shows the potential application areas of the three protocols.

The resource-oriented pattern, in general, provides a more intuitive abstraction when linking devices to the Internet. The Web is a loosely-coupled application layer architecture [15] and so is the IoT. With such a pattern, devices connected to the Internet are viewed as unique resources (identified by URIs) and accessed through well-known methods (such as GET, PUT, POST, and DELETE) with a clear semantics

	Transport	RESTful	Publish/ Subscribe	Request/ Response	Central Broker	Header Size (bytes)	Security	Reliability
MQTT	TCP	✗	✓	✗	✓	2	SSL	3 QoS levels
DDS	TCP UDP	✗	✓	✗	✗	-	SSL DTLS	23 QoS policies
CoAP	UDP	✓	✓	✓	✗	4	DTLS	Stop-and-wait retransmission with exponential back-off
HTTP	TCP	✓	✗	✓	✗	-	SSL	-

Table 3.1: A brief comparison between MQTT, DDS and CoAP

	Typical application
MQTT	MQTT is for applications which require real-time messaging, fall into the topic-based publish/subscribe pattern and accept persistent connections to message broker. The message broker is responsible for bridging end devices to the rest of the Web.
DDS	DDS is for applications and system-of-systems using publish/subscribe that have to support dynamically changing environments and configurations, be constantly available, and be instantly responsive. Integrating data across many platforms and disparate systems are also necessary.
CoAP	CoAP is for applications that consist of constrained devices but need to run RESTful Web services to achieve direct connectivity to the Web with HTTP like methods and URIs. Usually, applications require both request/response and publish/subscribe-like interactions.

Table 3.2: Potential usage of MQTT, DDS and CoAP

about read/write and representation state (controlled by Internet Media Types). An interesting aspect of the resource-oriented pattern is that it adapts to the model of Fog Computing well. It is straightforward to model Fog as a resource or collection of resources each capable of performing basic processing tasks, which further distributes the processing load and network traffic on backend services.

There is no IoT application layer protocol that fits all situations. As Al-Fuqaha et al. [47] pointed out, gateways that could interoperate with many different IoT protocols are under active research so that protocols can be deployed with much more flexibility. Among the three protocols that have been discussed, for example, MQTT could be suitable for situations where data need to be directly and reliably transferred to a server hosted on Cloud platform, while CoAP provides better support for device-to-device and device-to-Web communication. On the other hand, DDS makes more sense when deploying large-scale applications with strict real-time and QoS requirements under a reliable network environment. In the scope of this work, since the semantics of CoAP ensures its position in both Fog and Cloud, it is more straightforward to select this protocol to investigate the scalability of the IoT.

3.3 CoAP Fundamentals and Implementations

This section gives an inspection of some key features and concerns of CoAP, which further shows an insight into the ideas and trade-offs made in the proposed architecture that are discussed in the following chapters. It is followed by a brief analysis of available implementations which also shows the position of this work.

In detail, target environment of CoAP is stated in subsection 3.3.1; basic semantics of the core protocol is introduced in subsection 3.3.2; important extensions of the core protocol - resource observation and block-wise transfer are discussed in subsection 3.3.3 and subsection 3.3.4 respectively; security issue is covered in subsection 3.3.5; a summary of current implementations is in subsection 3.3.6.

3.3.1 Constrained RESTful Environments

To help the standardization work for constrained IP networks that are found in the IoT, the working group for Constrained RESTful Environments (CoRE) classifies resource-constrained devices to the following 3 classes based on their capabilities [16]:

- **Class 0** devices are the most constrained ones in terms of memory and processing capabilities. Their memory sizes are x in the order of hundreds of bytes only. It is unlikely for Class 0 devices to directly communicate with the Internet securely. Usually, they rely on larger devices that behave as application-level gateways to connect to general IP based systems.
- **Class 1** devices are the ones with lowest capacities to directly connect to the Internet in a secure manner, which in general have about 100 KB of ROM and about 10 KB of RAM installed. Class 1 devices are not powerful enough to talk to other nodes using protocol stacks like HTTP and Transport Layer Security (TLS) but are fully capable of utilizing lightweight ones such as CoAP.

- **Class 2** devices are less constrained and almost capable of supporting network stacks used by smartphones or laptops, which is enabled by about 250 KB of ROM and about 50 KB of RAM. Class 2 devices can still benefit from lightweight and energy-efficient protocols so that more resources are left for applications and overall operational costs can be reduced.

Constrained devices that are more powerful than Class 2 devices may still have some limitations, for example, they might depend on limited energy supply. Though the focus of the Constrained RESTful Environments (CoRE) working group is primarily class 1 devices, using lightweight protocols on class 2 devices and beyond enable direct interoperability between the most constrained devices and general Internet endpoints, which is essential to the IoT.

According to the above definitions, sensors and actuators are more likely to be class 0 devices. Some System on Chip (SoC) platforms fall into class 1 and may operate on behalf of the class 0 devices and expose their resources to the outside world. Other embedded platforms such as the mobile phone and the Raspberry Pi [83] can be seen as class 2 or beyond. They could thus perform more complex tasks and make real-time decisions based on information provided by potentially many class 0 and class 1 devices. Such a hierarchical structure naturally fits into the definition of a Fog. On the other hand, Erlang was initially deployed on telecom switches which are also embedded platforms. Therefore choosing the Raspberry Pi as the *scale down* option for investigating the Erlang-based solution should be a valid match.

3.3.2 Core Protocol

CoAP is designed to use minimal resources for both device and network. Instead of a complex transport stack, it gets by with UDP on IP. A 4-byte fixed header and a compact encoding of options enable small messages that cause no or little fragmentation on the link layer. Figure 3.4 shows the structure of a CoAP message, which includes the four-byte base header, the variable-length token, multiple header options, and a payload carrying a representation of the resource.

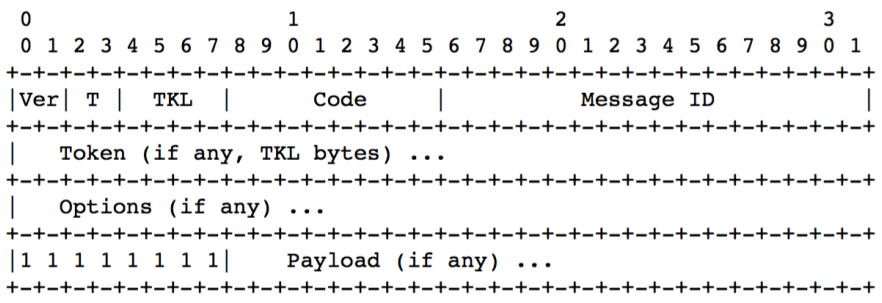


Figure 3.4: CoAP message format [89]: A 4-byte base header containing 2 bits for versioning, 2 bits for message type encoding, 4 bits for the token length, 1 byte for the message code (either a RESTful method or a response code), and 2 bytes for the message identifier (MID), followed by a token, options and a payload.

An entity participating in the CoAP protocol is called an endpoint [89]. It lives on a network node and is identified by its IP address, port, and security association. One can think CoAP of as having two sublayers: the request/response layer and the message layer, as shown in Figure 3.5. The former handles REST communications while the latter provides duplicate detection and optional reliable delivery of messages.

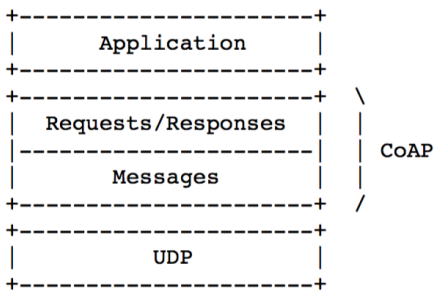


Figure 3.5: Abstract layering of CoAP [89]: Request/response layer for RESTful interactions and message layer for deduplication and optional reliable delivery.

On the message layer, messages can either be confirmable (CON), non-confirmable (NON), acknowledgements (ACK) or resets (RST). A message is identified by its message identifier (MID). CON and NON messages carry requests or responses. CON messages are used for reliable transmission. An endpoint retransmits a CON message with an exponentially increasing back-off timer until it is properly acknowledged or the maximum retransmission count is reached (which is typically 4). The response to a CON request can be sent with ACK with the same MID for message correlation (a piggybacked response), or in a separate CON response, which carries a different MID generated by the server. The separate response is useful to avoid waiting on the client side when the server can not respond instantly and can be closed by responding with an empty ACK that only contains the 4-byte base header. On the other hand, NON messages use a best-effort strategy and are not retransmitted in case of loss. A NON message can be answered with another NON message with a new MID generated by the sender. If an endpoint receives a CON or NON that it does not know how to process, it rejects it with an RST. An RST message also carries the same MID as the origin message similar to ACK, indicates failure of the message exchange and closes corresponding reliable transmission, if any. Examples of reliable and unreliable messages are shown in Figure 3.6a and Figure 3.6b respectively.

An endpoint needs to temporarily remember incoming MIDs to detect duplicates, which are usually caused by loss of ACK messages that should close corresponding CON messages, or by network jitters. The MID of each CON and NON message should be unique in the scope of the source endpoint, and should not be reused within the lifetime of the message, which is 247 seconds for CON message and 147 seconds for NON message.

On the request/response layer, requests have a method code (GET, POST, PUT, or DELETE) and responses have a response code (either of class 2.xx (success), 4.xx (client error), or 5.xx (server error)). A

(a) Reliable **(b)** Unreliable

Figure 3.6: Example of reliable and unreliable message transmission [89]

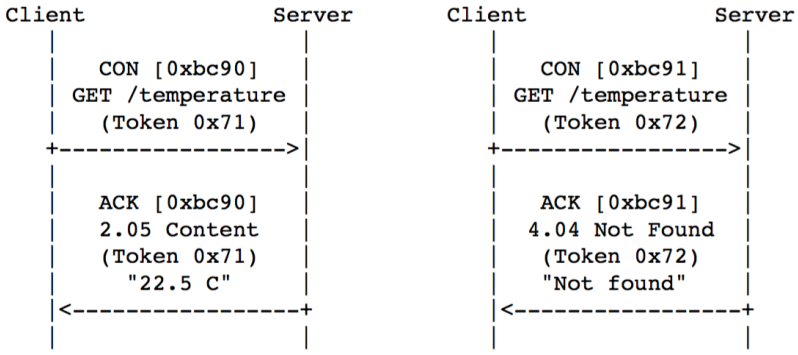


Figure 3.7: Example of two GET requests with piggybacked responses [89]

token (up to 8 bytes), chosen by the client, serves as the 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. The token must be unique at least within the scope of each destination. 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. Two examples for a basic GET request with piggybacked response are shown in Figure 3.7, one successful, one resulting in a 4.04 (Not Found) response.

3.3.3 Observing Resources

An essential feature for the IoT is the ability to track state change of particular data. Based on the observer pattern, the observe extension of CoAP [53] enables efficient server push notifications. It is designed as an optional feature on top of GET with an elective `Observe` option. The client sets the option to zero indicating its interest in observing particular resource. If a server does not support this extension, it responds as answering a normal GET request and the client will be aware of this situation and can further attempt to repeat request for polling. When the server supports this feature, it will include this option in the response, turning it into a notification. The server tries its best to keep the interested client on its observers list as long as possible. Whenever the observed resource undergoes a change, new representations will be pushed. All notifications are subsequent responses of the original request, correlated via the request token. The `Observe` option in notifications additionally provides a sequence number for reordering detection so that the client can

ensure it treats the most recent notification as the freshest one. Cache control is also enabled for notifications in terms of valid lifetime, defined by the `Max-Age` option. As a result, notifications are cacheable and the server would normally send a new representation before `Max-Age` expires. The client would assume it is removed from the server's observer list when a representation gets stale, which may be due to an expected reboot, for instance. The client has the chance to re-register itself by sending a new observe request with the same token. It is required that all options should be identical to the original observe registration, in order for the request to match all cache keys in case any intermediary is involved. See an example of observing in Figure 3.8.

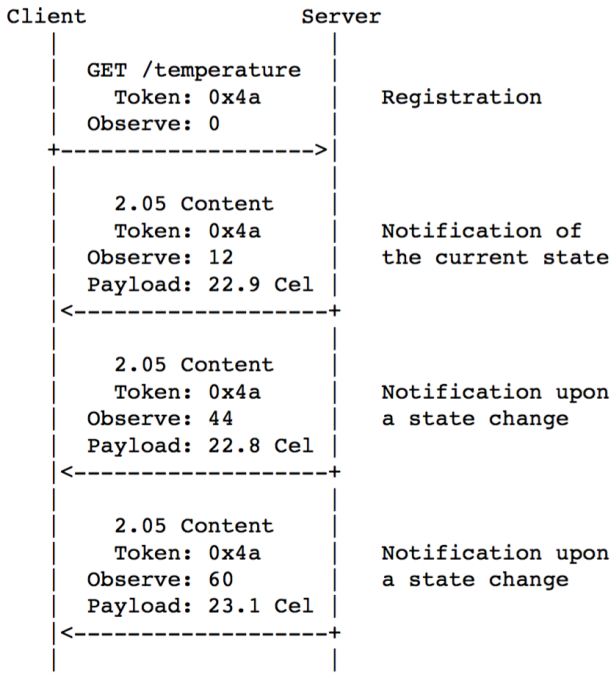


Figure 3.8: CoAP observe synchronizes the local state with the resource state at the origin server by sending push notifications. [53]

There are two methods to end an observe relationship once the client is no longer interested: [63]:

- **Re-active cancellation:** Observers simply get rid of the local relationship states. As a result, when the next notification arrives, the client does not recognize the token and will reject it. As according to the specification the server must use a CON notification occasionally in order to detect inactive observe relations, it will eventually receive an RST which triggers the server to remove the client from its observers list. The same applies to the case where a client shuts down, making a CON notification times out.
- **Pro-active cancellation:** If it is desired to cancel the observation in a more eager manner to save resources, clients can send a request indicating the cancellation by setting the `Observe` option to one and using the token associated with the observation.

3.3.4 Block-wise Transfer

CoAP is initially designed for small payloads, which is expected from most sensor-like devices. However, sometimes applications need to transfer larger payloads, for example, updating the firmware on the fly and transferring a CoRE Link Format of resource descriptions [88]. Though the underlying IP layer already provides a fragmentation mechanism, it is not always feasible. CoAP uses block-wise transfer [17] for these purposes. It is enabled by adding a pair of `Block` options so that transferring a large resource representation can be turned into multiple request-response pairs. In detail, the `Block1` option is used for uploading (PUT/POST) and the `Block2` option is used for downloading (GET). Multiple block size choices are available for different scenarios. Block-wise transfer can be implemented in transparent, atomic fashion, where the application logic will only handle the complete representation after all blocks are transferred. If not, applications have to process block logic explicitly, which is beneficial for devices with limited memory as they can do this step by step.

It is to be noted that block-wise transfer combined with observation will lead to only the first block to be sent as notification [17]. Rest blocks need to be fetched by the client using normal GET requests. As a result, complexity on the server side is largely reduced.

3.3.5 Security

Since Transport Layer Security (TLS) is not available for UDP, CoAP relies on Datagram Transport Layer Security (DTLS) for security [84, 89]. The DTLS In Constrained Environments (DICE) Working Group of the Internet Engineering Task Force (IETF) works on supporting the use of DTLS in constrained environments. Various implementations exist, for instance, tinyDTLS [37] is a C implementation targeting constrained devices, and integration of DTLS into the Java CoAP implementation Californium [18] is discussed in a master's thesis from 2012 [60]. It is considered that DTLS adds extra complexity and overhead and new version of the specification that aims at reducing latency has not been finalized yet [63]. As a result, it is not a focus of this work.

3.3.6 CoAP Implementations

Several CoAP implementations already exist and either target constrained or unconstrained platforms.

For embedded platforms, there are a variety of optimized implementations including CoAPBlip for TinyOS [81], SMCP [92], libcoap [66], Erbium (Er) for Contiki [64] and CoAPSharp [29] for the .NET Micro Framework. Cantcoap [22] is a CoAP implementation that focuses on simplicity by offering a minimal set of functions and a straightforward interface. Being a C implementation, however, it only focuses on decoding and encoding, leaving the actual protocol to the application. Although these implementations can also be deployed on an unconstrained platform, they are not designed for scalability and not suitable for performing complex services.

OpenWSN [79] is a comprehensive IoT project at UC Berkeley. Its main aspect is high reliability for low-power communication. Besides a full software stack for sensor nodes, OpenWSN offers a CoAP Python library [27] to implement backend services. It primarily targets easy interaction with OpenWSN devices and is not designed for scalability.

CoAPthon [30, 97] is another Python library for CoAP built on top of the Twisted framework [99]. It targets embedded systems or above, favouring easy development over large scalability.

The Sensinode NanoService Platform is a commercial solution that offers good support for industry-relevant features [63]. At the time of writing, it has become part of the ARM mbed platform [7] since the Sensinode start-up has been acquired by ARM at an earlier time. Java and C libraries are included for both devices and Cloud. However, these libraries are commercial and not publicly available.

mjCoAP is a lightweight and open-source CoAP implementation written in Java which targets less constrained embedded devices such as Raspberry Pi and middle-class smartphones [26]. Its design goals include interoperability, development simplicity, and code reusability rather than scalability.

There are two open-source Java frameworks jCoAP [59] and nCoAP [74] which target unconstrained platforms. The former one only implements an early draft version of CoAP at the time of writing thus is incompatible with the current CoAP standard. While the latter one, according to [68], may still be in early optimization stage and have problems that affect its performance.

Californium (Cf) [18] is a modular, open-source framework that facilitates deployment of backend services, serving as an intermediary between the logic of a service and the IoT. It originated from the research in ETH, Switzerland and has become an Eclipse Foundation project since 2014. With its core based in Java, it aims at providing scalable backend services for CoAP in the Cloud and has shown better performance than other implementations [68, 65, 63]. It also targets unconstrained platforms hence it makes more sense to run Californium on commodity server machines rather than on constrained devices, though it is still available on some embedded platforms due to Java's portability.

Muller [73] explored the possibility of using common Web application frameworks for HTTP such as Ruby on Rails in the Internet of Things utilizing CoAP and proposed a CoAP server implementation in Ruby called David [35]. It puts more emphasis on interoperability between common Web applications and CoAP and to what extent existing Web framework can be reused in IoT scenarios. The architecture of David is inspired by Californium and it has a similar or less performance as Californium.

Copper (Cu) [33] is a CoAP user-agent for Firefox implemented in JavaScript. It enables users to browse IoT devices in the same fashion they are used to explore the Web. This is done by providing a presentation layer that is originally missing in the CoAP protocol suite. Its ability to render a number of different content types such as JSON or the CoRE Link Format [88] makes it a useful testing tool for application as well as protocol development [60]. Copper is meant to run only as a client.

A brief summary and comparison between major CoAP implementations are shown in Table 3.3. Some details about certain implementations are temporarily not available at the time of writing. The column

	Language	CoAP Version	Target Platform	Scalability	Fault-tolerance
CoAPBlip	nesC/C	CoAP-13	Very Constrained	No	-
SMCP	C	RFC	Very Constrained	No	-
libcoap	C	RFC	Very Constrained	No	-
Erbium	C	RFC	Very Constrained	No	-
Cantcoap	C++/C	RFC	Very Constrained	No	-
CoAPthon	Python	RFC	Constrained	No	-
CoAP/OpenWSN	Python	CoAP-18	Unconstrained	No	-
CoAPSharp	C#	RFC	Constrained	No	-
mjCoAP	Java	RFC	Constrained	No	-
jCoAP	Java	-	Unconstrained	-	-
nCoAP	Java	RFC	Unconstrained	Yes	-
Californium	Java	RFC	Unconstrained	Yes	-
David	Ruby	RFC	Unconstrained	Yes	-
Copper	JavaScript	RFC	Web Browser	-	-

Table 3.3: Brief summary and comparison of major CoAP implementations. Target environment ranges from very constrained to unconstrained, which covers sensors, more powerful embedded systems and Cloud backends. - implies not applicable or not mentioned clearly.

scalability/fault-tolerance means whether the implementation declares itself as designed for scalability/with fault-tolerance in mind or relevant tests show these features. Current and comprehensive lists of CoAP implementations are to be found in the Wikipedia [31] and on the Website coap.technology [28].

Table 3.3 shows that major CoAP implementations either target constrained devices with poor support for scalability, or equip with full support for scalability but target unconstrained platforms such as the Cloud. Many CoAP server implementations such as jCoAP, CoAP Python library from OpenWSN and CoAPthon use Single-Process-Event-Driven (SPED) architecture which implies lacking support for scalability when it comes to multi-core environment [63]. On the other hand, there is a silence on fault-tolerance feature. Among these popular implementations, the lack of a combination of scalability and fault-tolerance within one solution that has a wider usage scenario ranging from less constrained embedded platform to resource-rich Cloud backend reduces flexibility for potential IoT applications.

The summary does not include information on implementations based on concurrency-oriented languages, though. By the time of writing, there is only one publicly available CoAP client/server implementation in Erlang called gen_coap [49], which is not under active development though. The ecoap prototype is inspired by some of its insights. The main motivation of developing another Erlang implementation is that gen_coap is more a proof of concept and only gives a rough idea on how concurrency can be modelled in Erlang. Many designs of gen_coap has to be reconsidered in the context of this thesis. For instance, the relationship between different components, the data flow of request handling and fault-tolerance policy. Moreover, it

lacks performance evaluation both under constrained and unconstrained environment. Local tests during development show that the proposed prototype has improved performance. Despite gen_coap, no other complete CoAP library in Erlang is publicly available. Some implementations utilizing other concurrency-oriented languages such as Go [50] exist, namely go-coap [32] and canopus [21]. However, both projects are incomplete as they only implemented part of the whole core specification.

Few benchmark tools for CoAP server performance testing have been presented, among which, CoAP-Bench from Californium (Cf) Tools [19] is a relatively widely adopted benchmark tool. Like Californium, it is also developed using Java. However, in order to generate high concurrent virtual clients for stress testing, the distribution mode which uses a cluster of machines to run the benchmark has to be employed. For the sake of simplicity, an Erlang benchmark tool following the same idea of CoAPBench has been developed and used for evaluation of this work. More details of the benchmark tool are discussed in subsection 5.1.1.

Chapter 4

Architecture and Implementation

Research on architectures for Web servers have been conducted for years since the Web itself scales rapidly with the ever-growing traffic. One focus in such research is scalability, which can be even defined with more restrictions as, executing concurrent tasks efficiently on modern computing hardware with multiple cores integrated. Originally handling concurrent requests in Web servers was introduced by the scenario that connections from multiple clients need to be accepted at the same time in order to better utilize the CPU during I/O operations [63], which is more a requirement for the connection-based protocol like HTTP. Though CoAP takes a different way compared with HTTP, the stage of request processing is similar and concurrency support is still necessary when facing a busy traffic. Different server architecture evolves over a decade, from which M. Lanter [68] and M. Kovatsch [63] have summarized the most significant ones as following, Multi-Process (MP), Multi-Threaded (MT), Single-Process Event-Driven (SPED), Asynchronous Multi-Process Event-Driven (AMPED), Stage Event-Driven Architecture (SEDA) and Multi-Threaded Pipelined (PIPELINED).

The first attempts of improving server performance was to employ more processes or threads in order to better utilize the CPU(s), which was easy to follow (as one process/thread to one user/request mapping was used and sequential logic could still be applied) but has been proved too expensive in terms of creation time, memory usage and context-switching. The following event-driven style architecture offers better performance especially on the single core as it is non-blocking and fewer threads are created and reused during the lifetime of an application. However, both methods have their pitfalls.

At a logical level, all server-side frameworks and applications have concurrent threads of control that transform the state space in a collaborative fashion [93]. Nevertheless, using threads directly as a programming model adds complexity in both data and control plane [93][69]. Threading model usually assumes a shared memory in which the updatable state resides and different threads of control take turns directly modifying the state space in-place. A variety of locking mechanism is used to guard and serialize the updates to the shared state in order to synchronize among threads. This can make for a highly concurrent system but is extremely error-prone and hard to reason about especially when it is combined with object-oriented programming because the later then limits the visibility that certain portions of a program have into portions of the state and effectively partitions the state space [69]. Threads are also not a practical abstraction in distributed computing as the effort to make a shared memory illusion is expensive [69]. On the other hand,

the asynchronous, non-blocking event-driven code essentially consists of a single thread with a main loop which waits and processes events accordingly. Because the thread of control is not interrupted preemptively, which in turn ensures that state updates can be made consistently without using locks, event-driven style often provides simplicity and performance over a multi-threaded architecture [93]. However, event-driven inverts the control flow and effectively turns a program into a reactive style, which makes it more complicated to block and harder to understand when the problem space increases [102]. Also, event-drive often assumes a uniprocessor context when compared with multi-threads model, and therefore would underperform under a multi-core or many-core environment. There exist attempts to combine the two paradigms where many single-threaded event-driven processes cooperate together. Such effort, however, still suffers from the synchronization issues mentioned before [102].

Concurrency in software is difficult and using threads as a concurrency abstraction makes it worse [69]. Non-trivial multi-threaded programs are difficult to comprehend. On the other hand, because threads are insufficient from a footprint and performance perspective, using threads as software unit of concurrency cannot match the scale of the domain's unit of concurrency today anymore. As an alternative, programmers have to use constructs that implement concurrency on a finer-grained level than threads and support concurrency by writing asynchronous code that does not block the thread running it, which are, however, hard to write, understand and debug as well [82]. Moreover, since today's modern hardware are equipped with multi-core or even many-core chips, which are essentially distributed systems themselves, the assumption of threading model makes it harder to be transparently portable and fully utilize the underlying hardware [93].

With all above being said, as Lee [69] pointed out, an alternative paradigm such as concurrent coordination languages with actor-oriented style of concurrency should be put more attention to. A language with actor paradigm built-in such as Erlang, though does not fully qualify a proper concurrent coordination language, still compensates many shortages of the threading model. An actor in Erlang is an Erlang process which is lightweight enough and can be started and destroyed very quickly, therefore allowing much easier design patterns that could use as many processes as needed. And code can be written in a linear, blocking, imperative style. All above greatly eases the burden of modelling the domain problem. On the other hand, actors have isolated memory and can only communicate using message passing, which avoids the problems and mistakes such as low-level data races and the subtleties of memory models associated with the current shared-memory paradigm [93]. Moreover, as a functional programming language, data is immutable in Erlang and messages between actors are copied in most cases, which further enhances this feature. Compared to languages where data are mutable and can be referenced by pointers, the aforementioned point could lead to inefficiency, which is a trade-off for fault-tolerance.

Erlang has strong fault-tolerance support since its background was rooted in the telecommunication industry, which may not be a common feature in other actor based solutions. The first writers of Erlang treat availability and reliability more important than other features in order to develop systems that "never stop". As a dynamic typed language, Erlang has facilities that help upgrade an online system without any

shut down time, which is also known as “hot code reload". When it comes to faults, instead of preventing errors and problems, Erlang assumes they happen from time to time and provides a good way to handle them. It is proved that the main sources of downtime in large-scale software systems are intermittent or transient bugs [20]. And errors which corrupt data should make the faulty part of the system to die as fast as possible in order to avoid propagating errors and bad data to the rest of the system [54]. So the Erlang way of handling failures is to kill processes as fast as possible to avoid data corruption and transient bugs. The sharing-nothing, immutable data, avoiding locks and other safeguards in Erlang ensures a crash is the same as clean shutdown [54]. And the ability of an Erlang process to receive signal when another process of interest terminates enables a supervision tree structure in an application, where the leaf nodes being workers who execute actual tasks and parent nodes being supervisors that can take immediate actions upon accidental termination of its children (worker nodes), such as reboot the worker to a known state. Such a structure effectively separates error handling and application logic. The idea is also called “Let it crash", which on one hand prevents programmers from over defensive-programming, on the other hand let the application only handle exceptional cases that are expected while hiding unexpected intra-system failures from the end users since they are already as self-contained as possible, improving the perceived reliability of the service. One can still dig into errors and failures afterwards to diagnose though since this mechanism does not aim at ignoring errors (they are properly recorded anyway) but saves the system from crash upon the very error occurs. The fault-tolerance model of Erlang is not a new one, robust computer systems use similar strategies more or less [52]. However, few environments provides such a finer-grained level of control over faults as Erlang does.

The actor model Erlang based on supports transparent distribution which makes it identical whether two communicating processes locate at the same machine or not. This is achieved by passing messages in a total asynchronous manner so that no assumption of the communication results is made [54]. The transparent distribution benefits both scaling and fault-tolerance. It naturally transfers to multicore processors in a way that is largely transparent to the programmer, so that one can run Erlang programs on more powerful hardware or over multiple machines without having to largely refactor them. Having concurrent Erlang VMs running and talking through message passing, the same pattern of communicating, detecting failures and relaunching or handling things can be applied on the far end. The asynchronous message passing also makes it possible for user shell or any other code, run as an Erlang process, to inspect the status of a remote virtual machine and manipulate the system with much less effort than other solutions.

Erlang runs on a virtual machine that has preemptive scheduling and per-process garbage collection built-in, which is vital for the runtime to achieve soft real-time, another telecom industry requirement. Preemptive scheduling is not as efficient as cooperative scheduling which is used in a language like Go, but is more consistent, meaning that millions of small operations can’t be delayed by a single large operation that doesn’t relinquish control.

The characteristics mentioned before not only render Erlang as a successful telecom industry language but also make it suitable for Web services where high concurrency and fault-tolerance are needed, such as

Web servers and chat service. In this work, it is argued that the Internet of Things shares similarities with these areas. With new paradigms such as Fog Computing emerging, IoT intends to be made up of more distributed computing force where latency is sensitive, scaling onto heterogeneous platforms ranging from embedded devices to cloud backends. Erlang should fit well in such circumstances.

Erlang itself is no silver bullet though. It is particularly inappropriate to use Erlang in signal/imaging processing, number crunching or any other CPU-intensive tasks. And it could be slower than other solutions because the language is dynamic typed and running on a virtual machine. Preemptive scheduling, as well as all other effort towards high concurrency and fault-tolerance, also brings overhead, which makes Erlang only perform better than other solutions under proper domain problems/workloads, such as busy server-side applications with lots of network traffic but few heavy computing tasks. However, since any non-trivial application is unlikely to be powered by single technology, the patterns used in Erlang are also applicable to other languages. For instance, Akka [4] is an open-source toolkit for building concurrent and distributed applications on the JVM, which is written in Scala and emphasizes the actor-oriented programming model over others. A .NET version is also available. Kilim [94][93] is a Java actor-oriented server framework which does the magic by transforming the Java bytecode. Project Loom [82] is a proposal that intends to introduce Fibers (similar concepts to actors) into the JVM. Complex server-side systems usually use a mixture of multiple languages and consist of isolated subsystems that communicate using well-defined messages [93]. This resembles actor-oriented Erlang system anyway. It is the maturity of the language and runtime that renders Erlang as the primary interest of research in this thesis.

4.1 Concurrency Model

The popular Java CoAP framework Californium was inspired by previous work for highly concurrent Internet services, in particular, SEDA and the PIPELINED architecture [68]. However, much of its assumption is invalid in a concurrency-oriented language context, since creation and synchronization of lightweight processes is much cheaper. Therefore, a more intuitive and straightforward architecture similar to Multi-Process (MP) is still an attractive option. The primary goal of the design is to allow scalability and fault-tolerance following the idiomatic concurrency-oriented language way, that is, isolation of processing, data, and faults.

It is a common design pattern among Erlang applications to model truly concurrent activity each as a separate process. Therefore it is straightforward to come up with the architecture shown in Figure 4.1. Conceptually, the ecoap prototype can be split up to a socket manager, an endpoint, and a handler, which are reusable for both server and client. When being used as a server, a registry maintaining routing rules is also included. The socket manager hides network details and would only dispatch received datagrams to endpoints. Thus its number depends on how the underlying transport protocol works. For plain CoAP which is built above connection-less UDP, there is a single instance of the socket manager. While for alternative transport such as DTLS, there might be one socket manager per endpoint. The endpoint represents an actual

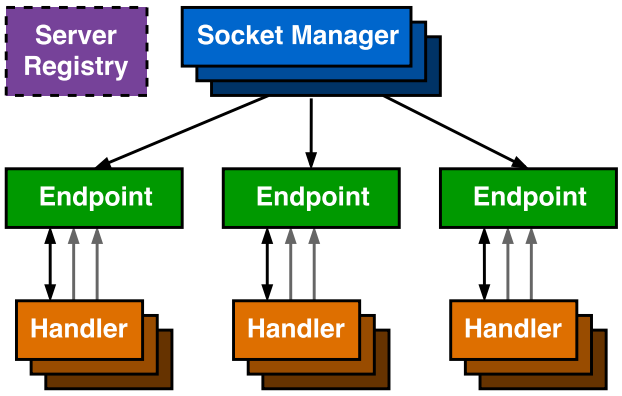


Figure 4.1: The logical architecture of ecoap. One or more socket manager(s) exist depending on the underlying transport protocol. One endpoint maps to exactly one remote CoAP endpoint. Each endpoint may have many handlers associated with it. Handler contains logic for server or client. A server registry is only used for routing in server.

remote CoAP endpoint and is created following a one-to-one mapping scheme. It executes the protocol and would let the handler to invoke business logic via a RESTful interface. Depending on the role of the system, different handlers are to be used. It should be noticed that not only the endpoint has a one-to-one mapping against real CoAP endpoint, the handler each maps to an independent CoAP operation as well, where an operation may contain one or more CoAP message exchanges that are logically related. Therefore it is possible to have thousands of endpoint processes each having one or more working handler process(es) all running concurrently and only communicate through message passing, which provides a fine-grained concurrency model that ranges from network level to business logic level. It should be further noted that while the handler executes the business logic, it only requires a pre-defined interface is implemented and has no other limits, which means the business logic can have its own concurrency model as well. The model is much like a pipeline besides all stages of the pipeline are spawned dynamically and have a clear mapping to real-world concurrent activities.

On the other hand, these processes will be organized in different ways depending on whether the system runs as server or client, and also some other application-specific requirements, primarily for fault-tolerance considerations. In order to compose a well-defined fault-tolerant application, supervisor processes are necessary to glue the above components together to eventually form a supervision tree. Only by combining the concurrency model with the supervision tree could the proposed prototype achieve essential scalability and reliability.

With the above in mind, the subsequent sections are organized as follows. Details of each component are discussed respectively from subsection 4.1.1 to subsection 4.1.4 while different supervision strategies and the structure of the whole application are introduced in subsection 4.1.5. Then subsection 4.1.6 analyzes how states are managed in such a share-nothing environment with special challenges coming with CoAP.

4.1.1 Socket Manger

The workhorse of the socket manager is an Erlang process which holds the socket. It is responsible for receiving binary data over the network and applying flow control if desired. It, therefore, abstracts the transport protocol, which is UDP in plain CoAP. Though out of the scope of this thesis, the component can be easily replaced by wrapping a socket that listens on DTLS port or over another transport layer. The main difference would be instead of a single instance that is known by all endpoints, a connection-oriented alike transport protocol will render more socket processes to match concurrent connections and therefore a one-to-one mapping for socket process and endpoint applies.

With plain CoAP, the socket manager becomes the only place where data traffic goes through, as the received datagrams are also sent as plain Erlang messages to the process by the runtime. In order to avoid the bottleneck, the process should do as little work as possible. When the socket process receives a datagram from a new remote endpoint as a server, it starts a local endpoint process and passes the datagram to it together with the source address and port. When a client intends to issue a request towards a server that is not touched before, the socket process starts a local endpoint process as well and passes the provided destination address and port to it so the client can use the component to send the request. The endpoint processes are monitored by the socket manager to maintain a dynamic dispatch table so that it can hand received datagrams to corresponding endpoint process immediately, as shown in Figure 4.2. The dispatching is based on the inner process dictionary of the socket process. A process dictionary is a destructive local key-value store in an Erlang process. It should be noted that the process dictionary destroys referential transparency and makes debugging difficult [39]. It is primarily used to store system information used by the VM that does not change a lot during the lifetime of a process. However, when being used with care (packaging its operation inside well-defined API which does not touch other states), process dictionary provides faster reading/writing performance than other key-value stores in Erlang. Thus the process dictionary is used as a fast dispatch table where the key is a tuple of remote endpoint address and port, and the value is the process identifier (PID) of the worker process that receives messages from the endpoint in the rest time of processing.

An interesting feature of the Erlang runtime is while only the owner process of a socket can read data from it, any other process can write to it as long as the process has access to the socket reference. This behaviour can be used to improve port parallelism. So, it is desirable that the socket reference is also passed to the endpoint process when it is created, which enables the worker to send messages directly over the network without further bothering the socket manager. A function that encapsulates sending operation is provided by the socket manager module, which can be invoked by endpoint processes. It is a direct function call without any message passing and hence leaves the socket process alone. This largely reduces the workload of the socket manager process and makes the dispatching between socket manager and endpoint processes in a totally asynchronous manner.

Though it is possible to improve data throughput by letting multiple processes listen on the same port, each holding a different socket, this behaviour largely depends in the underlying operating system and does

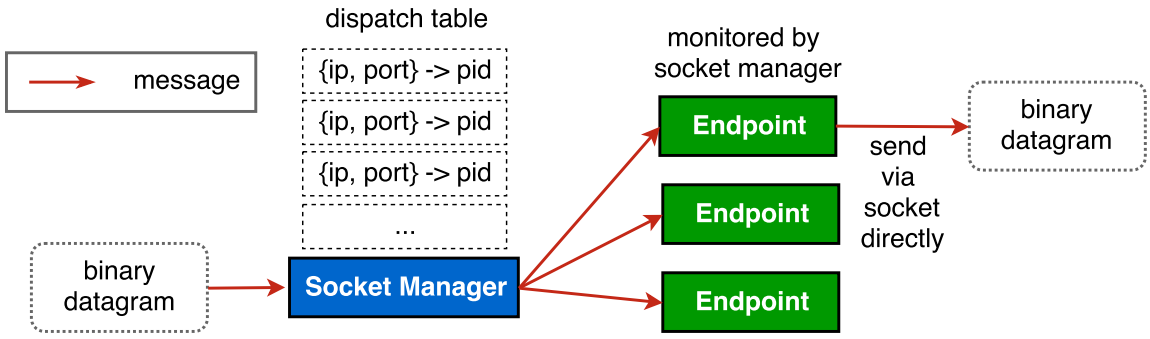


Figure 4.2: The socket manager receives datagram over network and dispatches them based on its endpoint dispatch table. The endpoint process sends datagrams directly using the socket reference passed by the socket manager. Every endpoint process is monitored by the socket manager so that the latter would notice its termination.

not behave well universally. On the other hand, having a single process instead of many simplifies the management of socket and data dispatching.

4.1.2 Endpoint

An endpoint process is the representation of the actual remote CoAP endpoint inside ecoap. Because of the low cost of creating and destroying processes in Erlang, a one-to-one mapping is taken here, which means CoAP messages from one remote endpoint are guaranteed to be handled by the same endpoint process during the active session.

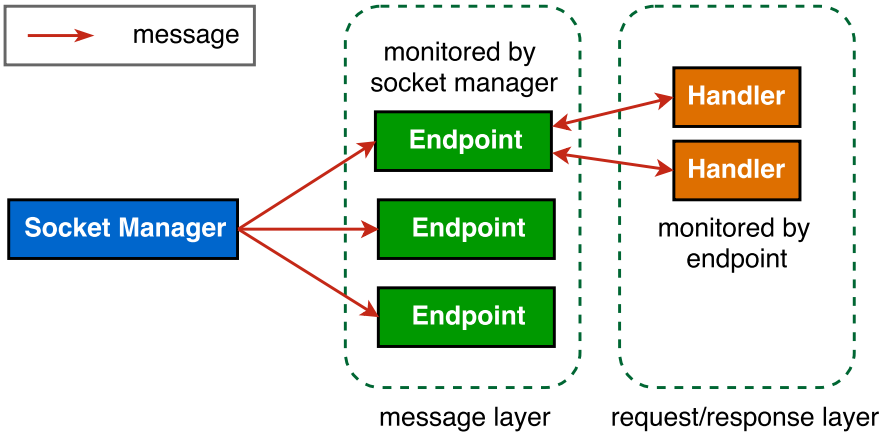


Figure 4.3: Relation between endpoint and handler

As shown in Figure 4.3 and Figure 4.4, the endpoint process wraps the implementation of CoAP message-layer. It is responsible for decoding and encoding CoAP messages, checking duplicates and optional retransmission for reliable message exchange. The process handles above tasks by tracking message exchanges and maintaining a state machine for each of them. An exchange should contain one request and its corresponding response or an empty message while different exchanges may have logical relations, for example, a block-wise

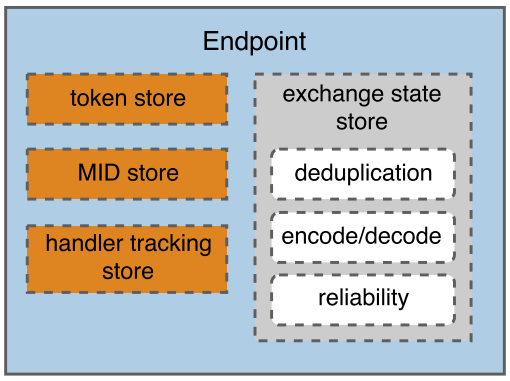


Figure 4.4: The function of an endpoint. The exchange state is the description of both the current state and the action that should be taken upon the state next. Therefore, deduplication, serialization and reliable transmission are all defined as part of the actions and are executed accordingly.

transfer or an observe notification renders a request followed by potentially many responses. Using state machine for protocol stage transition is relatively easy and clear and different types of message exchange can have different state machine implementations. However, it is unpractical to make every state machine an independent process because the potentially huge amount of message exchanges could lead to too many processes within the system. Instead, the message exchange state contains the description of both the current state and the next action to be taken upon the state. The endpoint process then stores the exchange state as part of its inner state. In such a way, it can process multiple message exchanges asynchronously without spawning new processes. For instance, an incoming confirmable message could render a state as "finishing message parsing and delivering to handler process but still waiting for the corresponding acknowledgement", together with the function name that triggers a transition to the next state. The very endpoint process can store this exchange state and process other CoAP messages while waiting. When the corresponding acknowledgement arrives as Erlang message, it can fetch the stored exchange state and continue executing it via invoking the state transition function by the fetched function name. The dynamic and functional nature of Erlang enables this feature without much effort. On one hand, the function name is part of the exchange state and can be stored as data. On the other hand, the runtime can use the name to refer to an executable function located in the same file as the exchange state is defined. In similar ways, retransmission can be done by setting timers that will trigger resending if no state transition that cancels them happens before the final timeout. Detailed state transition flowchart can be found in subsection 4.2.3.

The workflow can be briefly described as follows. After receiving the incoming datagram, the endpoint process checks duplication first and triggers corresponding actions upon the exchange state if it is a duplicate. Otherwise, a new state is initiated where the CoAP message is decoded into ecoap's inner presentation and handed to an appropriate handler process for further processing. After the handler process finishes its task, it passes the result back to the endpoint process which then triggers a state transition where actions including encoding the result and sending it over the network are executed.

The endpoint process sets a one-way monitor on every handler process so that it can be notified when

they terminate (either a clean finish or a crash). This has an important impact on the lifespan of an endpoint process, which will be discussed later in subsection 4.1.6.

The endpoint process also manages CoAP message identifiers and tokens when it needs to issue a CoAP message proactively, i.e. being a client or sending a non-confirmable/separate response as a server. The message identifier, or MID, is generated sequentially starting from a random number within the MID range, but independent among different endpoint processes. The share-nothing of Erlang renders MID management straightforward since there is no concern of collision out the scope of a single CoAP endpoint. Within one endpoint one should still make sure that a MID is not reused before corresponding exchange lifetime ends. The message token is generated as a strong random byte using Erlang's crypto library with configurable length. Tokens are also only produced by endpoint process independently. One could argue that randomly generating token does not fit in a variety of scenarios. However, using crypto strong random bytes as token within one endpoint's scope greatly simplifies the effort for synchronization and avoids a collision to a relatively high level, which is enough for the ecoap prototype. While MID is stored as an integer, the endpoint process stores the token in a key-value store which maps request token to the identifier of the request issuer. The token store is necessary for matching incoming responses to requests and dispatching them to corresponding issuers.

4.1.3 Handler

In general, a handler process serves as the request/response layer in CoAP as in Figure 4.3. Depending on the role of the system, the handler process can act as:

Server

An `ecoap_handler` process is used to execute server-side logic, including invoking CoAP resource handler functions, server-side block-wise transfer and observe management, as shown in Figure 4.5.

A CoAP resource provides a RESTful interface and can be accessed and modified through reacting to requests that carry one of the four request codes defined in CoAP: GET, POST, PUT, or DELETE. One can define a CoAP resource by implementing its handler functions as callbacks required by `ecoap_handler`, which is similar to an interface in object-oriented programming, and register the mapping of resource URI to the module name of the handler functions at the CoAP Registry. When a request arrives at the server, eventually a `ecoap_handler` process searches the registry for a resource that correspond to the destination URI of the request. If the `ecoap_handler` process finds the resource, the handler functions are executed to process the request and responds with an appropriate response code, options and payload according to the protocol specification, otherwise, a 4.04 (Not Found) error code is responded. The assembled response is then passed back to a corresponding endpoint process which sends it to the client via the network.

The `ecoap_handler` process is spawned by the endpoint process on demand and on the fly. For any ordinary request, there is usually one `ecoap_handler` process per request, and the process terminates imme-

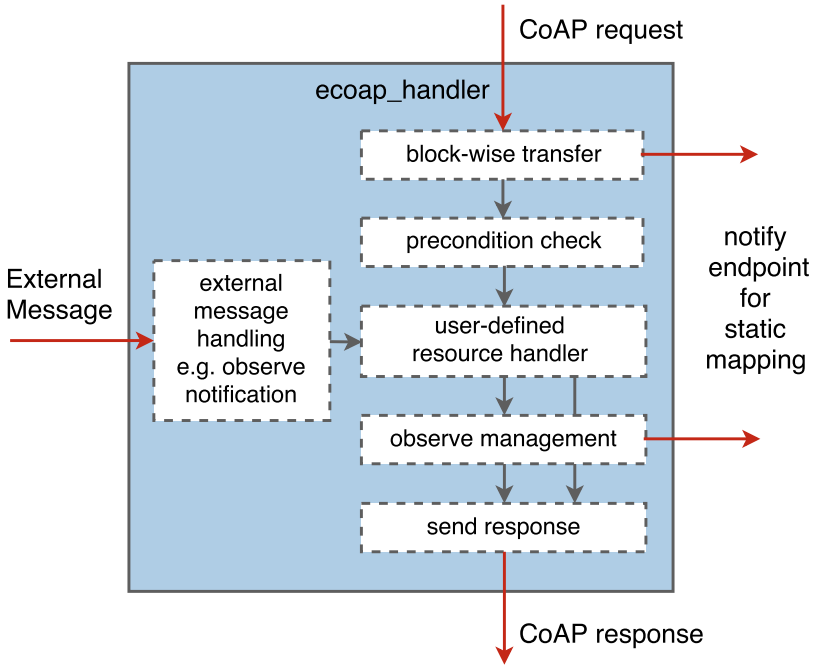


Figure 4.5: Workflow of an ecoap_handler process

diately after sending the response. No bookkeeping work other than monitoring is taken by the corresponding endpoint process. As a result, requests can be processed concurrently in an intuitive, sequential and blocking fashion. It should ease programming effort especially when the resource handler has to execute certain time-consuming tasks as it does not need to worry slowing down the server. The Erlang runtime scheduler will make sure all processes share the time slot fairly. If the request has to be responded timely, for instance, a confirmable request, its corresponding message exchange state machine will set a timer while waiting for the `ecoap_handler` process, so that an empty acknowledgement can be delivered to the client if the handler process fails to finish before timeout event. The late result automatically becomes a separate response that is included in a new message exchange.

However, for block-wise transfer and observe requests, an `ecoap_handler` process keeps serving following requests which query the same resource in the exact same way. In detail, subsequent requests to one particular resource with the same CoAP method and query (CoAP Uri-Query option) as the first request will be processed by the same `ecoap_handler` process as long as there is an ongoing block-wise transfer or observe relation towards that resource. The `ecoap_handler` process lives until the block-wise transfer completes, or the client cancels the observe relation. While requests not subjected to this limitation are handled as normal in newly spawned processes.

This is achieved as follows. After checking the request, the very `ecoap_handler` process informs the corresponding endpoint process its intention to be alive and the latter records its identifier as a combination of CoAP method and query so that subsequent requests matching the identifier can be delivered correctly. Since an endpoint process always monitors the handler process, it will be informed on termination of the

handler process and remove the mapping information. This serves two purposes. The first one is it greatly simplifies block-wise transfer and observe management. An `ecoap_handler` process handling a block-wise transfer works in an atomic fashion as defined in RFC7959 [17]. For a block-wise GET, it caches the complete representation of the resource at once for the client to retrieve step by step, in order to avoid unnecessary resource fetching or possible inconsistency caused by the resource changing frequently. For a block-wise PUT or POST, it gradually collects the data uploaded by the client and updates the target resource in one action, so that no intermediate state could be seen by others. The static mapping effectively avoids inconsistent resource state being spread over multiple places. On the other hand, when handling observe relation establishment, an `ecoap_handler` process registers itself to *pg2*, a distributed named process groups application that comes with standard Erlang distribution, with the resource URI as the group name. In such a way, `ecoap_handler` processes serving different clients can join one group if the clients intend to observe the same resource. One can then notify the clients recent update of the resource of interest by simply sending the update as Erlang message to the group name, which will be broadcasted to all registered processes by *pg2* afterwards. Eventually, each `ecoap_handler` process will assemble an observe notification and push it to the corresponding client. The static mapping again avoids separation of states and provides a clear model. Moreover, the *pg2* application ensures a group is properly cleaned when any of its member processes exit. It even works in a distributed environment out of the box, where processes located on different machines can join one group and receive messages, with basic race conditions and partitions handling mechanism [40], which enables ecoap to scale out with minimum effort. Other process group applications with similar functionality can be placed here as well. The second benefit is that requests that are not related to ongoing block-wise/observe activity can still be handled concurrently just as other ordinary requests, no matter whether they come from the same client or not. Since block-wise transfer and observe are essentially stateful operations, only employing static mapping for them renders each operation as isolated and self-contained as possible while obtaining a balance between high concurrency requirement and maintaining states on the server side.

Client

The `ecoap_client` process encapsulates request composing, client-side block-wise transfer and observe management.

Synchronous and Asynchronous requests are both supported and implemented via message passing. Any external Erlang process can act as the caller of an `ecoap_client` process and ask it to perform the request on behalf of the process. Synchronous calls block the caller process until a response is delivered. Asynchronous calls return immediately with a reference (a tag that is unique within an Erlang runtime). The response will later be delivered as a message to the caller process, which can be pattern matched against the request reference so that the request and response are associated.

Different from ordinary request calls, synchronous and asynchronous observe calls both have a reference

in their return values. The synchronous observe calls also return with the first notification along with the reference. Proactive observe cancellation calls are provided in the same manner as observe calls. When a user process asks the `ecoap_client` process to observe a resource when it is already being observed, the behaviour of `ecoap_client` process is compliant with the re-observe action defined in RFC7641 [53], that is, reusing the same request token.

The `ecoap_client` process also handles block-wise transfer in an atomic fashion, where the response is handed to user process only after the whole exchange completes. The concurrent block-wise transfer is an undefined behaviour [17]. The `ecoap_client` process deals with this by abandoning the ongoing block-wise exchange and continuing with the newly established one, which is essentially an overwrite and the desired result in many cases.

Moreover, one can use the reference obtained from an asynchronous call to cancel the issued request before the corresponding response is received. If the reference is from an observe call, the observe relation is cancelled in a reactive way, where the next notification from the server will be rejected by sending a reset message. Any ongoing block-wise transfer corresponding to the request is stopped immediately after invoking request cancel call. This is done by turning ongoing block-wise message exchange to a cancelled state where further events such as message retransmission are just ignored.

Multiple user processes can share one `ecoap_client` process since each request/response/observe/block-wise transfer tracking is self-contained and will not interrupt with each other. All tracking data are stored in a key-value store as part of the process inner state, providing fast reads and writes.

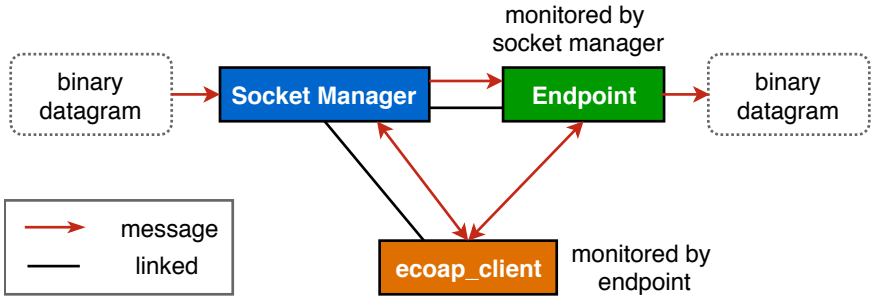


Figure 4.6: Structure of a standalone client. The ecoap_client process only uses message passing to ask the socket manager to start corresponding endpoint process.

A client can be started as a standalone one or an embedded one. As a standalone client, the `ecoap_client` process starts a socket manager and links to it. For the socket process, when the `ecoap_client` process issues a request, it directly starts an endpoint process and links to it as well (if there is not an existing one), instead of using any other supervision tree structure, as shown in Figure 4.6. Therefore a standalone client consists of one `ecoap_client` process, one socket manager and zero or one endpoint process. The client as a whole can be connected to any supervision tree as usual in a standard way so that one can compose a customized application. The link among processes is to ensure crash of any of them will definitely bring down others and thus avoid orphan process and inconsistent state.

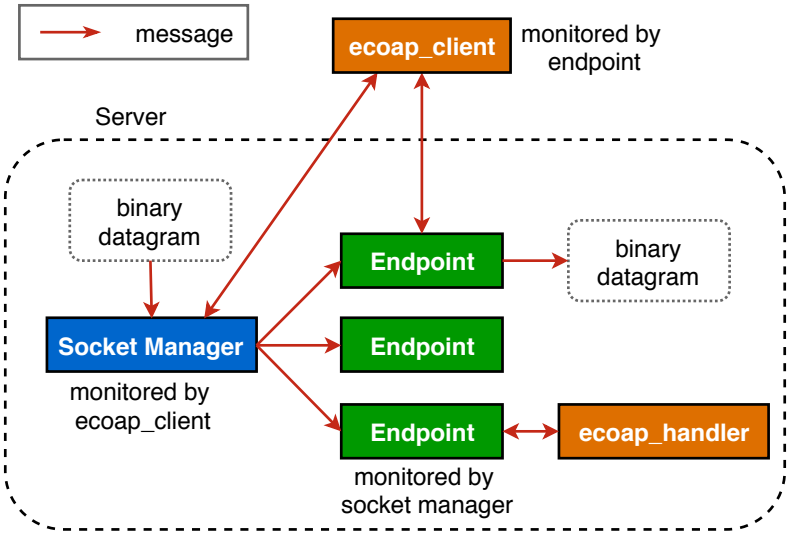


Figure 4.7: Structure of an embedded client. The ecoap_client process is not directly involved in the server but asks the server to send request on behalf of it, so that client and server can share the same network interface.

While an embedded client uses an existing socket manager instead of spawning one. This can be useful when an application wants to behave as server and client at the same time while sharing the single network interface, like the behaviour defined in OMA Lightweight M2M (LWM2M) [78], an IoT device management protocol built atop CoAP. For instance, an `ecoap_client` process can be started and specified to use the same socket manager of a running ecoap server. Then when issuing a request, the newly created endpoint process will be connected to the supervision tree of the server and will be used for both client role and server role which avoids duplication and confusion of states within the application. Also, depending on the requirement of the application, the `ecoap_client` process could be started either during the initialization of the server or inside a resource handler function.

The endpoint process monitors the `ecoap_client` process for a special purpose. Assume in the embedded mode an `ecoap_client` process exits due to any reason, the endpoint has to cancel all ongoing message exchanges that the client originated. The endpoint uses the monitor to track this extra information. It is inappropriate to let the endpoint process terminate with the client process as what would happen in a standalone client because the endpoint process might be used by the server concurrently. In the standalone mode, termination of an `ecoap_client` process will shut down the socket manager and all alive endpoint processes anyway, no matter the termination is normal or a crash. The monitor is more for a consistent reason other than the purpose stated above. On the other hand, since the socket manager is not started by the client process in the embedded mode, there is no direct link between them. Therefore the `ecoap_client` process should monitor the socket manager instead of in case it terminates unexpectedly.

The `ecoap_handler` and `ecoap_client` process both work as the handlers that provide an interface to business logic. They can both be considered as request/response layer of CoAP. The main difference is they

are connected in different ways to the corresponding CoAP endpoint component and supervision strategy varies. This is because server and client have reversed data flow, where the `ecoap_handler` process is the end of request processing in a server while the `ecoap_client` process is the one that begins a request.

4.1.4 Server Registry

A server registry is essentially a manager for routing rules of a server. For the sake of simplicity, a key-value based routing is used in the ecoap prototype, where the key is the URI of the resource presented as a list of path strings and the value is corresponding resource handler module name.

The routing rules are stored in an Erlang Term Storage (ETS), which is an efficient in-memory database built in Erlang VM that allows limited concurrent operations. In most cases, a data structure is held by an Erlang process as its internal state and the process acts upon it through message passing with other processes. However, when the data structure needs to be shared with many processes, this isolation makes the single process a bottleneck. The ETS provides a way to store large amount of data with constant access time. It also allows concurrent destructive operations with some restrictions (See [43] for more details). It acts like a separate process with its own memory but can be configured to allow direct accessing from other processes. In the case of this thesis, reading the routing rules occurs at most of the time instead of writing, which renders ETS a fast and reasonable solution.

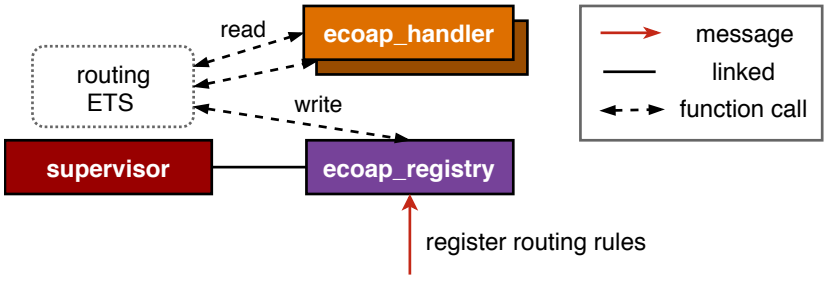


Figure 4.8: Supervision tree of the server registry. The routing ETS table is owned by the supervisor. The ecoap_registry process serialize all writes to the table including insert and delete. Many ecoap_handler processes can read from the table concurrently. Reads and writes are direct function call without accumulated messages in a single process.

An ETS table cannot be linked to or monitored as the normal process, but has the concept of ownership. The process that creates the table becomes the owner and the table will be removed after the process terminated or crashed. The table can be configured to allow only the owner to operate (private), only the owner to write and other processes to read (protected) which is the defult, or any process to read and write (public). In order to improve fault-tolerance, ecoap takes a workaround: let the ETS for routing rules be created by a supervisor process with public access control, but only allow one child process to perform actual write operations. This turns the child process, the `ecoap_registry` as shown in Figure 4.8, to a table manager which serializes all writes but still gives the permission of reading to all other processes. The structure is based on the assumption that updates to routings seldom happen and will not become a bottleneck. Since

the supervisor process does not take part in any other work than monitoring its children, it is unlikely to crash. When the `ecoap_registry` crashes due to any reason, it will be restarted by the supervisor while the table keeps alive. Therefore, instead of one process, the server registry consists of two including a supervisor.

The matching is done by first searching a key that equals to the resolved URI, and then finding the one with the longest prefix if the former step failed. This way one can have a handler for `/foo` and another for `/foo/bar`. One can also specify whether a wildcard in URI is allowed or not. If wildcard is used and a resource handler with the longest matching prefix is selected, the suffix of the resolved URI can be retrieved by the handler function for pattern matching. The matching is exported as a function executed in the context of `ecoap_handler` processes that directly operate on the ETS.

4.1.5 Supervision Tree

Fault-tolerance is the most important feature of Erlang apart from concurrency. The previous section analyzed the fault-tolerance of the server registry. In this section fault-tolerance policy of other components and the architecture of the whole prototype is presented.

Though true fault-tolerance could not be achieved without distribution and redundancy, the fine-grained fault-tolerance settings in Erlang ensure that a transient error could not bring down an application easily even when it runs on a single node. Fault-tolerance is primarily reached by supervision trees where each supervisor has a separate restarting policy towards their children.

Erlang follows the convention of an onion-layered approach and tries to identify the *error kernel* of an application. The *error kernel* is where the logic should fail as less as possible or even is not allowed to fail. In order to protect the most important data, as a general rule, all related operations should be part of the same supervision trees, and the unrelated ones should be kept in different trees. While within the same tree, operations that are more failure-prone can be placed deeper in the tree, and the processes that cannot afford to crash are closer to the root of the tree [54]. This approach decreases the risk of core processes dying until the system cannot cope with the errors properly anymore.

Before diving into details, some basic terminologies are given first. First of all, supervisor processes are the ones whose sole work is to make sure their children, the processes spawned by them, are under control when they terminate. By saying under control it means the very supervisor can take proper action upon the termination event, like restart the child process. Worker processes are the ones in charge of doing actual work and may crash while doing so. Supervisors can supervise workers and other supervisors but workers can only be positioned under one supervisor. They can then form a supervision tree.

There are basically two reasons why a proper Erlang program should always use a supervision tree if it is to spawn any process. The first one is only with supervision tree can all processes belong to one application be tracked. This is important since otherwise, one may silently leak process and eventually use up all available memory. The second reason is that the supervision tree makes sure an application can shut down in a determined order. It is simply done by asking the top supervisor of an application to terminate, which will

then asks all its children to shut down. If any of the children is also the supervisor, it does the same to its own children until all the tree terminates. In case of unexpected conditions such as a process can not terminate correctly, its supervisor has the option to directly kill the process.

Though there are many considerations when designing a supervision tree, the primary concerns here are the supervisor restart strategy and the child restart type. The supervisor restart strategy defines how a supervisor reacts to children termination in general, considering the relation between children processes. It can be one of the following:

- `one_for_one`, which means if a supervisor has many children and one of them terminates, only restart that one. It is usually used when processes are independent of each other.
- `one_for_all`, which means if a child process terminates, all other child processes are terminated, and then all child processes, including the terminated one, are restarted. This is used when processes under a tree depend on each other to work properly.
- `rest_for_one`, which means if a child process terminates, all child processes that are started after the one should be terminated as well and all of the child processes are restarted. It is a special case where processes have partial dependencies and start order matters.
- `simple_one_for_one`, which is a simplified `one_for_one` strategy, where all child processes are dynamically added instances of the same process. While the supervisor with other strategies can start its child processes accordingly during the initialization of the supervisor itself, a `simple_one_for_one` supervisor has no child at the start and will only add the child when being asked for. One of the big difference between this one and `one_for_one` is that `simple_one_for_one` is much more efficient when one has only one type of child process but needs to frequently start and shut down any number of them.

The child restart type, on the other hand, specifies what the supervisor should to when that particular child terminates. It can take one of the three values:

- `permanent`
- `temporary`
- `transient`

A permanent child is always restarted no matter what (be it a crash or a normal termination). It is often used for vital, long-living process of an application. A temporary child is never restarted no matter what (even if a `one_for_all` is used and a sibling death brings down the temporary process). It is for a short-lived worker processes that are expected to fail and few other code depends on them. While a transient child is restarted only when it terminates abnormally. It is used when a worker process needs to succeed in its job but will not be used later. An application can mix the child restart type with supervisor restart strategy as needed but one has to be aware of the final effect.

One can also tell a supervisor about the restart limits of its children. That is to say, the supervisor will give up and exit if more than maximum allowed restarts happen within a specified period of time (in seconds). This exit includes terminating all other child processes that may still be running. And eventually, this exit will be handled by the supervisor's supervisor (if there is any).

Other factors that can be taken by supervisors are not discussed here for simplicity and one may refer them to [96].

With the aforementioned guide in mind, the supervision tree of ecoap is shown in Figure 4.9. This is primarily a supervision tree of a CoAP server since clients structure depends heavily on use cases.

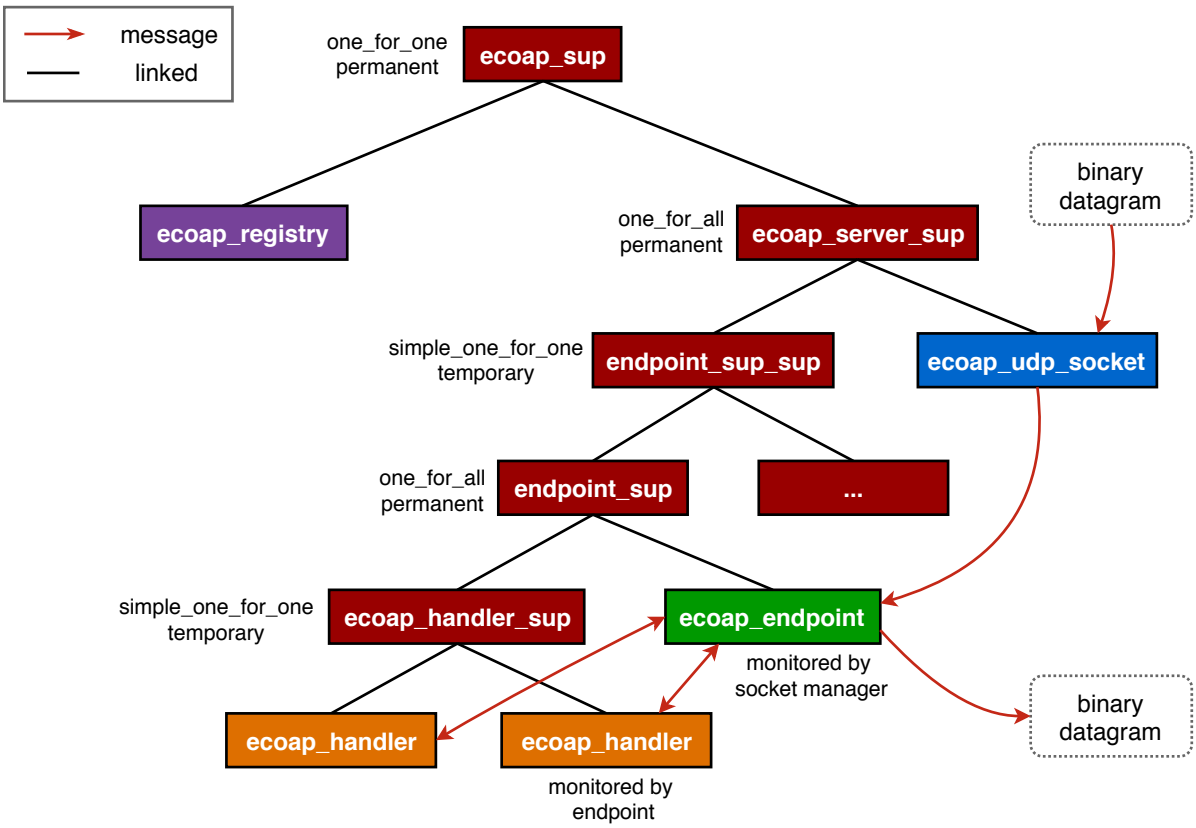


Figure 4.9: Supervision tree of ecoap as server

The `ecoap_handler` process is the deepest node in the whole supervisor tree as it executes user-defined resource handler functions which is a more risky operation. Its supervisor should make no assumption about it and ignores its crash since whether the process succeeded in sending the result is unknown. Instead, the CoAP protocol ensures retransmission of the message so that another `ecoap_handler` process is likely to be started later.

The characteristics of `ecoap_handler` are obvious: There is no dependency among multiple `ecoap_handler` processes; All `ecoap_handler` processes are dynamically added during runtime; No other type of child process under the same supervisor exists. Based on the above observation, a `simple_one_for_one`, `temporary` strategy is the most proper one for the `ecoap_handler` process. With this restart strategy, a crash of any

`ecoap_handler` process will not cause any restart and will not affect other running processes under the same supervisor.

An `ecoap_endpoint` process is logically connected to corresponding `ecoap_handler` processes since they all represent operations of an active remote CoAP endpoint. Therefore the `ecoap_handler_sup` process should be the sibling of the endpoint process and both under the same tree. The `ecoap_endpoint` process knows this supervisor and asks it to start handler processes.

The `ecoap_endpoint` process is the core of modeling a remote CoAP endpoint. The crash of it usually means all ongoing message exchanges towards the very endpoint becomes invalid. All processes related to the endpoint process should be terminated and the higher level supervisor is not supposed to restart them since all necessary data for that remote endpoint to continue have lost already. It is also impractical to have failover at this level because it adds obvious synchronization overhead. Instead, ecoap expects the remote endpoint to restart message exchange or retransmit the last message if it is still active and expects further actions. In such a case, a new endpoint process is spawned and everything goes on like normal except the loss of message exchange history, which can render re-execution of certain logic if the duplicate of a message that is received previous to the crash arrives. This can, however, be avoided by a well-defined RESTful interface which utilizes idempotent operations or by extra application-layer measures to protect important states. The above actions are needed anyway to implement a reliable service.

It requires the termination of an `ecoap_endpoint` process brings down its supervisor sibling and their common supervisor. While a supervisor process cannot be shut down easily without explicit command, a workaround as follows is taken. The top supervisor, namely `endpoint_sup`, sets `ecoap_endpoint` and `ecoap_handler_sup` both as `permanent` and uses a `one_for_all` restart strategy. Meanwhile, the restart limit is set to 0 times in 1 second. This way termination of any of the two child processes would trigger their common supervisor to restart them all but immediately reaches the restart limitation and directly terminate its rest child as well as itself.

If the tree of `endpoint_sup`, `ecoap_endpoint` and `ecoap_handler` is considered as a compact component, there needs to be another supervisor on top of it so that the socket manager can start new endpoint process via it. This supervisor, `endpoint_sup_sup` should therefore take a `simple_one_for_one`, `temporary` strategy because its situation is similar to `ecoap_handler_sup`.

Although the single `ecoap_endpoint` process approach is not fault-tolerant enough for a particular remote endpoint, it effectively ensures that faults are isolated between different remote endpoints, improving fault-tolerance performance of the whole system.

It is worth noting that, when an embedded client reuses the server socket process, any new created `ecoap_endpoint` process still has the `ecoap_handler_sup` sibling, though it may not be put into use.

The socket process, `ecoap_udp_socket` in the case of this thesis, is the *error kernel* of ecoap. Crashing of the socket process implies the socket is closed and any reference to it becomes invalid. In such a case, all endpoint processes cannot send CoAP message using the closed socket anymore and must terminate as well.

It is possible to make the socket process a named process and let endpoint processes send messages to the socket process which then doing the network sending on behalf of them. Then the crash of socket process does not affect endpoint processes since they only send messages to a named process who will be restarted soon and no other process will compete to register the name. However, this method introduces extra load to the socket process as it effectively serializes the outgoing CoAP messages. More importantly, the socket process maintains its dispatching table inside its process dictionary which is destroyed as soon as the process terminates. Thus the above method turns the system into a situation where only endpoint processes could send messages out but the restarted socket process can not identify them and ignore their existence, rendering an inconsistent state. It is only doable when the dispatching table is outside the socket process, such as in an ETS. Nonetheless, this approach adds the risk of race condition and is not considered.

As a result of that, the supervisor of `ecoap_udp_socket` and `endpoint_sup_sup`, namely `ecoap_server_sup`, takes a `one_for_all`, `permanent` strategy so that when any of these two crashes, they are both restarted immediately. This also ensures that any subtree under this layer is terminated as well during this restart. For instance, when the `ecoap_udp_socket` process crashes, `endpoint_sup_sup` is also terminated by their supervisor and both are restarted, which also brings down any child under `endpoint_sup_sup`, which then brings down their children accordingly until all nodes under `endpoint_sup_sup` are terminated (Note the processes marked as `temporary` will not be restarted). After the restart, no orphan endpoint process or blind socket process will exist. The same logic applies to any combination of crashes of the two processes.

If one considers the above structure with another level of abstraction, the server can actually listen on different ports simultaneously by starting multiple instances of the `ecoap_server_sup` supervisor with different parameters (e.g. the port number), which then starts their own `ecoap_udp_socket` and `endpoint_sup_sup` processes, ready for incoming requests respectively. This way requests towards different ports are also isolated and executed concurrently with minimal effort. And because of reusing existing components, all fault-tolerant features discussed above still apply.

On the other hand, in order to reduce complexity, the server registry does not need an extra supervisor. Instead, the routing table is kept by the root supervisor with the `ecoap_registry` process being a direct child of it. Since the registry worker process is independent from the rest of the application, the root supervisor can simply have the `ecoap_server_sup` supervisor as its another child and uses a `one_for_one`, `permanent` strategy. One can then start as many server instances as needed while sharing a single registry process. The root supervisor is then linked to the application controller of the Erlang VM. Crashing of the root supervisor implies failure of the whole application thus no further action is taken other than completely shutdown ecoap. This also agrees with the design of the server registry, as the routing table will only be destroyed when the root supervisor is down, in which case there is no point to continue service without human intervention.

In summary, the influence of a crash is limited as much as possible to avoid cascade failure and improves the availability of the entire server. Faults happened in one handler process will not affect another handler process, even when they are serving the same client. Similarly, failure of one endpoint process will not be

noticed by other remote endpoints. The server will only be considered unrecoverable when the faults causing too many restarts that exceed the preconfigured maximum restart limit on each layer of the supervision tree until eventually propagating to the top supervisor, which then terminates the whole application. Despite monitoring and restarting failed child processes, the supervision tree also clarifies the dependency of worker processes, making it easier to understand and extend the application.

4.1.6 State Management

Deduplication States and Lifespan of Endpoint

A CoAP endpoint might receive the same message multiple times. Two messages are equal, if their sources, destinations, and MIDs are the same within the same lifetime [68]. It is required that a CoAP endpoint remembers confirmable messages for an EXCHANGE LIFETIME (247 seconds) and non-confirmable messages for a NON LIFETIME (147 seconds) [89]. When a duplicate message arrives the server should respond with the same response without re-executing the request (at-most-once semantics). Thus a CoAP endpoint has to cache the response for transmission, along with the request information for filtering. The potential a large amount of ongoing requests and the memory they cost require the server to reduce the exchange states and drop them after finishing the requests as soon as possible.

It is decided that each `ecoap_endpoint` process should maintain a key-value store that maps MID to exchange state instead of using a central bookkeeping. It avoids centralized bottleneck and improves fault-tolerance since the message exchanges in different endpoints are effectively isolated. Each `ecoap_endpoint` process periodically iterates through all entries of the exchange store and checks their age. Any message whose lifetime has expired is removed from the store. It is proved that the above method is more efficient than creating a timer for every message. A timer is also implemented as a process in Erlang, using more timers than required would slow down the performance of the endpoint process handling normal CoAP messages. It can be argued that the periodical scan would block the endpoint process. However, the fine-grained concurrency model where one endpoint process maps to exact one remote CoAP endpoint makes it unlikely to accumulate too many entries within one process, rendering the scan blocking an acceptable trade-off in terms of performance. After all, a remote endpoint sending with high frequency could easily reuse MID too early which breaks the protocol anyway. Moreover, this design eliminates any need for synchronization or locking.

It should be noticed that CoAP server can further relax deduplication [89]. First of all, NON requests are less critical as their lifetime is shorter and the semantics indicate their unreliability: they can be designed so that no response needs to be cached and only duplicate filtering is needed. Furthermore, when processing only idempotent requests such as GET, PUT, and DELETE, no responses have to be cached and ecoap can be configured to completely disable deduplication. For computation intensive resources, results can be cached locally. On the other hand, POST requests can be optimized using application knowledge to implement a

more efficient deduplication strategy.

It is also important to determine when a remote endpoint is not active anymore, so that the corresponding `ecoap_endpoint` process can be safely shut down (with the sub supervision tree it links to) and resource being hold can be released. A remote endpoint is considered inactive when all of the past message exchanges have expired, no outgoing request is left uncompleted, and no `ecoap_handler` process is running. All the conditions have to be met at the same time because there are situations where ongoing exchanges exist but the exchange store is empty. For example, when a client is observing a resource that is seldomly updated, or waiting for a separate response of a time-consuming task, there is a chance that no new message ever arrives during the wait and all exchanges have expired and been removed. Apparently, the endpoint process should not terminate in such a case, be it either on the client side or on the server side.

On the one hand, in order to track outgoing requests, the endpoint process simply reuses its token store. Since the token store is cleaned up whenever a request is completed, checking whether it is empty is enough to make sure no request is left open. On the other hand, the endpoint process monitors all alive `ecoap_handler` processes and maintains a counter which tracks the number of them. As a result, it could then directly check whether the corresponding stores are all empty as well as the counter is set back to zero right after each exchange store cleanup to determine if it should terminate or not. However, it is not enough to examine only the above conditions when no deduplication is enabled or when message lifetime is configured to be strongly reduced. The endpoint process might falsely consider it is about to terminate while there is still ongoing traffic on the way because the exchange store is emptied too fast. To avoid this situation, the process should also track whether any message has ever arrived between each cleanup, which can be easily implemented as a flag that keeps being set each time a message is received. To sum up, an endpoint process could only safely terminate after a cleanup on its exchange store when the following items are all true,

- The last message exchange has expired.
- The handler process counter is zero.
- No outgoing request is left uncompleted.
- No message has arrived between two cleanups.

The four conditions are not valid in all cases, for example, the handler process counter is always zero for a client. But any violation of them implies the remote endpoint may further communicate with the application and corresponding processes should be ready for that. When an endpoint process does terminate, the socket manager will be notified on this event and remove its record from the dispatching table. The sub supervision tree where the endpoint process resides is automatically shut down as described in subsection 4.1.5.

Observe States

The basic observe workflow is introduced in subsection 4.1.3. Another challenge is the management of observe notification sequence. Since UDP datagrams can get reordered, observe notifications must carry

a strictly increasing sequence number in the Observe option. The client can eventually have the latest representation of a resource by selecting the notification with a higher sequence number and drop the old ones. Generating observe sequence number in a multi-threaded environment with shared memory is complex due to race conditions [63]. In ecoap, however, the problem is avoided by generating the sequence number within the `ecoap_handler` process. Observe notifications can only be sent by first going through the single `ecoap_handler` process that maintaining the observe relation for the very client, which effectively serializes the notifications in the exact order as they reach the message queue of the process. There is no chance that a first produced notification gets a higher sequence number than the actually latest one does. This method compliances with RFC 7641 [53], which says "The sequence number selected for a notification MUST be greater than that of any preceding notification sent to the same client with the same token for the same resource." On the client side, things are similar no matter if multiple notifications arrive at the same time or not, as they are received and queued in the single `ecoap_client` process. Regardless of the arrival order, there will not be concurrent processing against the notifications and the single client process can reorder them safely. Because of the fine-grained concurrency model, serialization of the above procedures derives a low cost but simple solution.

4.2 Implementation

Previous sections explain the architecture of ecoap and how the different parts work together to fulfill the processing chain of CoAP messages. In the following section, a more detailed introduction to how a CoAP message is modeled and how these components can be implemented is given.

4.2.1 Process Overview

Erlang provides the Open Telecom Platform (OTP) framework, which generalizes common design patterns into a set of libraries and exports them through Erlang behavioubehaviourrs. Behaviours are formalizations of these common patterns which divide the code of a process into a generic part (a behaviour module) and a specific part (a callback module). The OTP offers a bunch of basic behaviours such as supervisors, finite state machines, event managers and generic servers. One can then use the behaviour by implementing a customized callback module which exports a set of pre-defined callback functions, much like an interface implementation in an OOP context. The OTP and behaviours reduce the complexity in terms of code, testing, maintenance, and understanding. ecoap is built all based on standard OTP behaviours. Besides all supervisor processes, which are obviously implementations of the `supervisor` behaviour, the socket manager, endpoint and handler are all using the generic server, namely `gen_server` behaviour. While these components might be more efficient if written without using behaviours, it is considered the increased efficiency does not compensate for the lost generality in this work. On the other hand, following the spirit of OTP, ecoap provides its own behaviour as well. One can implement customized resource handler logic by implementing

the `ecoap_handler` behaviour, where callback functions for REST verbs and CoAP observe are defined in a consistent manner, as shown in subsection 4.2.4.

The socket manager, endpoint and handler processes communicate mostly in an asynchronous manner using one-way `gen_server` cast or direct message passing in order to avoid unnecessary blocking. For example, when the socket manager delivers a CoAP message to corresponding endpoint process, it does not care about the result of this operation and a blocking `gen_server` call serves no purpose here. In case of a process crash, the delivery can safely fail because related processes will be terminated or restarted anyway. The above also applies to the communication between the endpoint process and the `ecoap_handler` process. The only exceptions are with `ecoap_client` process and the server registry, as the former has to provide meaningful results to the client while the latter must ensure an operation such as registering a routing entry be a deterministic and atomic operation to avoid race conditions. Their exposed APIs are therefore `gen_server` calls under the hood, which will bring down the caller if a failure happens, a usual behaviour in many Erlang applications. All communication details are hidden by wrapping the casts/message passing in proper APIs though, so that further changes that are internal to the process can be made smoothly.

Figure 4.10 shows the supervision tree of a running ecoap example server through *Observer* [77], the graphical tool of Erlang which displays system and process information. The example consists of three active clients where one of the clients has established an observe relation towards certain resource hosted by the server.

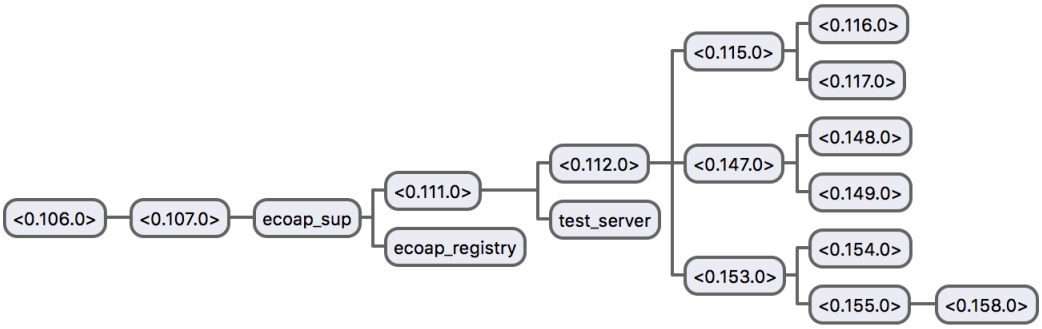


Figure 4.10: The figure shows the process tree of an example server. The *test_server* is an instance of the socket manager process. Process *<0.116.0>*, *<0.148.0>* and *<0.154.0>* are endpoint processes representing three active clients. Among them the client for process *<0.154.0>* has established an observe relation, which is handled by the handler process *<0.158.0>*.

4.2.2 Message

It turns out simple to implement binary based protocols such as CoAP with pattern matching in Erlang. The following example shows how one could extract different parts of a binary CoAP message with one line of code, where the binary is split to segments separated by a comma, pattern matched respectively and bound to the corresponding variables. `?VERSION` refers to a pre-defined macro that equals to the current CoAP version

and can be seen as a constant. Therefore, `?VERSION:2` matches the first two bits of the binary against the constant so that any binary starts with different two bits fails the match and will not be recognized as a proper CoAP message. After then, fields begin with a capital letter are free variables to be bound and the digit follows each field implies the length of the segment to be matched against. For example, `Type:2` binds the second segment (which is two-bits long) to variable `Type` and `Token:TKL/bytes` means binding variable `Token` to `TKL` bytes start right after the 16 bits message identifier (`TKL` is bound earlier in the match and is an integer). Assume `BinMessage` is a datagram received from the network, the binary pattern matching avoids unnecessary parsing effort while legal parts of a message can be obtained effectively without further twiddling with bits.

```
<<?VERSION:2, Type:2, TKL:4, Class:3, DetailedCode:5, MsgId:16, Token:TKL/bytes, Tail/bytes>> = BinMessage.
```

Listing 3: Example of parsing binary CoAP message through pattern matching

It is error-prone and obscure to use the above format all over the application despite the convenient binary syntax of Erlang. Therefore, a CoAP message is further decoded and represented as an Erlang record, which is a syntax sugar of tuple where attributes can be directly accessed by name. Listing 4 and Listing 5 show the definition of the record for a CoAP message and an example, respectively. Default values can be set within the definition as well as the type of each field. One can then get attributes by only pattern matching against desired ones. For instance, pattern `#coap_message{type=Type, id=MsgId, token=Token}` extracts `type`, `id` and `token` of a message while other parts are ignored. Updating a record can be done in a similar way.

```
-record(coap_message, {
    type = undefined :: coap_message:coap_type(),
    code = undefined :: undefined | coap_message:coap_code(),
    id = undefined :: coap_message:msg_id(),
    token = <<>> :: binary(),
    options = #{} :: coap_message:optionset(),
    payload = <<>> :: binary()
}).
```

Listing 4: Definition of the CoAP message record with default values and type information. It is defined as *attribute = default value :: type* where *type* can be native Erlang type such as *binary* or user-defined type expressed as *Module:Type()*. For example, attribute **token** and **payload** has a default value of empty binary while attribute **options** is of map type under the hood.

It should be noticed though that record syntax is only valid within the module it is defined. In order to access the definition globally, it must be put in a header file (like a C header file) that is explicitly included by other modules. It is argued that exposing a record across multiple places is not a recommended approach in Erlang applications. However, encapsulating it all by accessor functions reduces a lot convenience on direct pattern matching. Therefore ecoap makes the definition a project-wide header file meanwhile providing functions to manipulate the record, so that one still can obtain attributes without knowing the innards of it.

```
#coap_message{
    type = 'CON',
    code = 'GET',
    id = 8243,
    token = <<155,209>>,
    options = #{'Uri-Path' => [<<"example">>, <<"one">>]},
    payload = <<>>
}
```

Listing 5: Example of a CoAP message as a record. The message is a confirmable GET request for the resource located at "/example/one" with a MID of 8243 and a two bytes token of [0x9b, 0xd1] (Note each byte is represented as decimal in the figure).

It is a trade-off between expressing ability and maintainability.

4.2.3 Exchange

Instead of using the OTP `gen_statem` behaviour which defines finite state machine to be run as an independent process, the state transition of the message exchange is implemented using plain Erlang functions. Similar to a message, a record is used to hold useful information of an exchange, including the timestamp and lifetime of the message that initiates it, the current stage of the state machine, the cached result for deduplication, the timer for triggering any timeout event and finally the retransmission counter. The exchange state machine is then completed by combining the exchange record and state transition functions that accept the record as input and manipulate the state according to the information stored in the record.

The endpoint process uses the built-in `map` data structure to store the exchange records. In order to distinguish the incoming and outgoing messages, the exchange records are not indexed by just the MID, but a combination of the direction of the message and its MID, like `{in, MID}` or `{out, MID}`. When an endpoint process receives an incoming CoAP message delivered by the socket manager, or an outgoing CoAP message sent by the handler process, or a timeout message caused by a timer, it searches the map and invokes corresponding state transition function as specified by the stage stored in the exchange record. The full state transition flowchart is shown in Figure 4.11.

4.2.4 API Example

The following example shows how a simple CoAP server instance can be constructed with the ecoap prototype. One needs to implement resource handle functions as well as callbacks for observe push events. This is done by implementing the `ecoap_handler` behaviour along with application-specific code. The resource handler needs to be added to the server before it serves any request.

Resource handle functions have the remote endpoint address, prefix, and suffix of request URI and the request itself as parameters. The prefix is the path that represents the resource registered with the server registry while the suffix is any other segments following the prefix (if wildcards is allowed, otherwise empty).

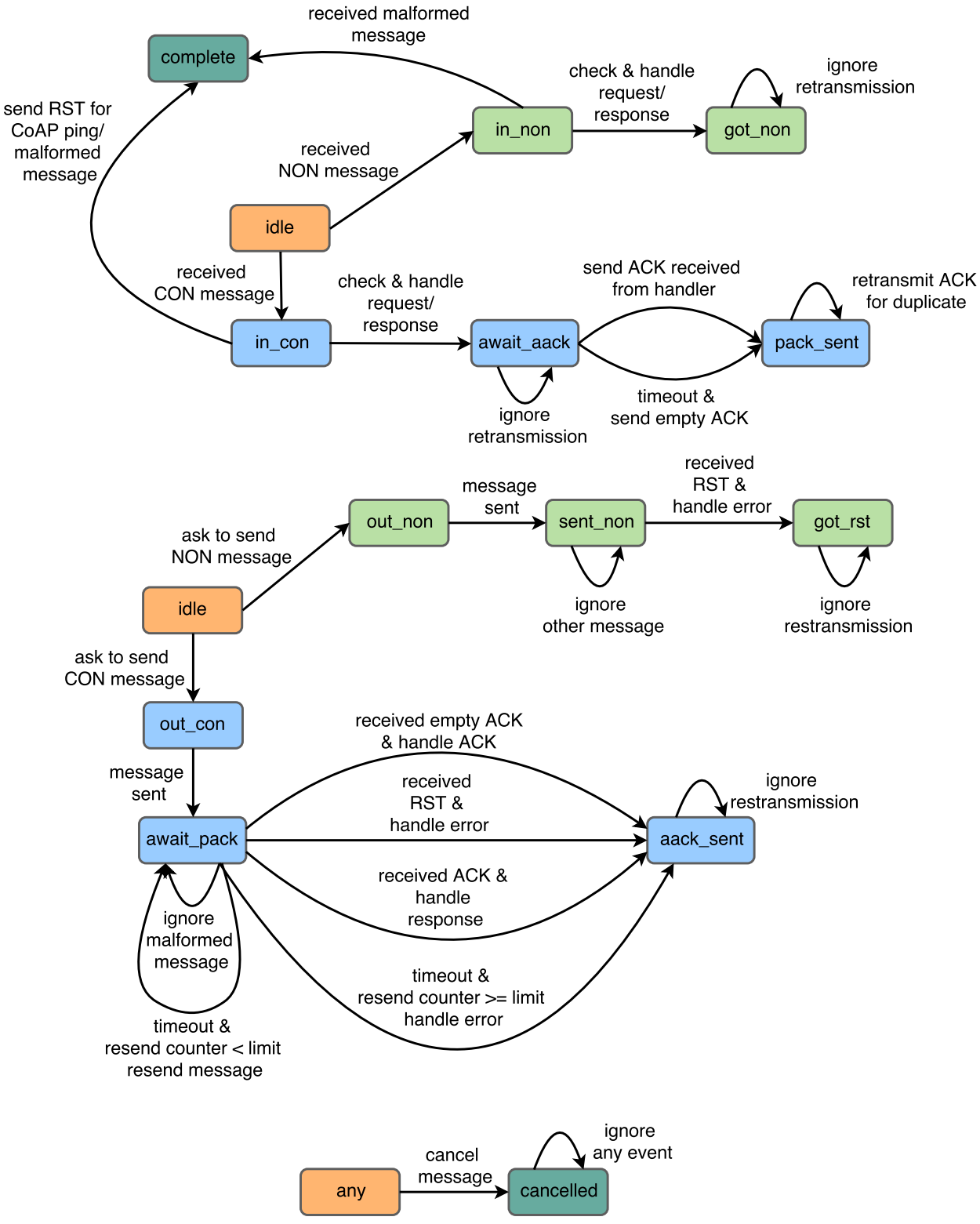


Figure 4.11: Exchange state diagram. All exchange starts from idle state. The complete state means the exchange is immediately removed.

```erlang
-module(example_server).
-export([coap_discover/1, coap_get/4, coap_put/4, coap_delete/4,
    coap_observe/4, coap_unobserve/1, handle_info/3]).
-export([start/0, stop/0]).
-behaviour(ecoap_handler).

start() ->
    % start ecoap and any application it depends on
    {ok, _} = application:ensure_all_started(ecoap),
    % maps path /HelloWorld to the resource handler that implements ecoap_handler behaviour
    Routes = [{[<<"HelloWorld">>], ?MODULE}],  % ?MODULE refers to current module
    % server is started with registered name hello_server
    ecoap:start_udp(hello_server, [{port, 5683}], #{routes => Routes}).

stop() ->
    application:stop(ecoap).

% resource operations

% function that returns the description of the resource, if any
coap_discover(Prefix) ->
    % the resource name is the same as its registered path with no extra description
    [{absolute, Prefix, []}].

coap_get(_EpID, _Pefix, _Suffix, _Request) ->
    % reply with 2.05 payload (text/plain)
    {ok, coap_content:new(<<"HelloWorld">>, #{'Content-Format' => <<"text/plain">>})}.

coap_put(_EpID, Prefix, Suffix, Request) ->
    Content = coap_content:get_content(Request),
    ecoap_handler:notify(Prefix++Suffix, Content),  % notify all observers
    ok.

coap_delete(_EpID, Prefix, Suffix, _Request) ->
    ecoap_handler:notify(Prefix++Suffix, {error, 'NotFound'}),
    ok.

coap_observe(_EpID, _Prefix, _Suffix, _Request) ->
    % do something
    {ok, dummy_state}. % {ok, SomeState} where SomeState could be anything

coap_unobserve(dummy_state) ->
    % do something
    ok.

% ObsReq is the origin request that starts the observe, State is the one returned by coap_observe/4
handle_info({coap_notify, Msg}, _ObsReq, State) ->  % observe notifications from ecoap_handler:notify/2
    {notify, Msg, State};  % directly send notification without modification
handle_info(_Info, _ObsReq, State) ->
    {noreply, State}.      % ignore other messages send to the very ecoap_handler process
```

Listing 6: ecoap server API

One can pattern match against the prefix and suffix to get dynamic URIs, for example, to extract the value of a wildcard or switch to alternative function clause if handlers of multiple routes are implemented in the same module. The `Request` is a CoAP message record underneath and can be used to obtain options and payload via helper functions. The `coap_discover/1` function is used to get the description of a resource as defined in Core Link Format. The `coap_observe/4` function is invoked when an observe relation is established with the resource and would return any useful state valid during the observe. The `coap_unobserve/1` is the opposite of the `coap_observe/4` and should do clean work if needed. The `handle_info/3` function is triggered when the `ecoap_handler` process representing the ongoing observe relation receives resource update message as a result of `ecoap_handler:notify/2` function call or any other message being sent to the process. One can manipulate the notification message in this function or directly forward it. It is also possible to generate a unique reference here if the notification is to be sent as a confirmable message. Then when the message is successfully acknowledged, `coap_ack/2` will be invoked with the very reference to verify this delivery. If no reference is explicitly specified, a dummy value is used.

In the example, the server is started with registered name *hello_server* which can be later used to locate the socket process. It can be stopped by calling the `stop/0` function. It has one entry in the routing table which routes requests to `/HelloWorld` to the handler functions implemented in the same module. The GET handle function responds with the text "HelloWorld". The PUT handle function invokes `ecoap_handler:notify/2` to send a new notification with the same content (payload and options) as in the request to all registered observers. The DELETE handle function sends a 4.04 (Not found) response to all observing clients. One may notice that the `coap_post/4` and `coap_ack/2` is not implemented. This is because ecoap sets all the callback functions as optional and a default implementation will be invoked if the user does not provide one. This minimizes redundant code template.

For clients, the `ecoap_client` module exports necessary functions to access a remote server. The module can be used in a similar way as OTP `gen_server`. One needs to call `ecoap_client:open/{1,2,3}` to spawn and links to a `ecoap_client` process and `ecoap_client:close/1` to stop it. The first argument of the `open` function is the hostname or address of the server in the form of `coap://example.me:5683` or `coap://192.168.0.12:5683`, where the port number can be omitted (the default port will be used). The other optional arguments are the local port to be bound or the socket process to be associated with as an embedded client, as well as other environment settings such as alternative transportation parameters of the protocol. Synchronous requests can be sent through `ecoap_client:get/{2,3,4}`, `ecoap_client:put/{3,4,5}`, `ecoap_client:post/{3,4,5}` and `ecoap_client:delete/{2,3,4}`. Arguments include the path (and query) for the resource, request payload, other CoAP options and timeout value if needed. All the above functions have their asynchronous versions. Similarly, there is `ecoap_client:observe_and_wait_response/{2,3,4}` and `ecoap_client:observe/{2,3}` as well as their unobserve counterparts. The only difference between the two is that the former will block until the first notification or another valid response is returned from the server while the latter is completely asynchronous. Since there are usually multiple versions of a func-

tion with different number of arguments, one can have more refined control over the requests. Finally, `ecoap_client:cancel_request/2` can be used to cancel an asynchronous request and `ecoap_client:flush/1` can get rid of unwanted messages caused a certain request or a certain `ecoap_client` process from the process mailbox.

When the optional timeout argument is not given, a synchronous call blocks the caller until a valid response is delivered. If the server is unreachable and the maximum retry time has been reached, a timeout error value will be returned instead. When the optional timeout argument is given, the caller will crash if no response arrives within this period of time. For asynchronous calls, since the response are delivered as a message, one can use either the `receive` primitive in Erlang or let an OTP process handle it via its relevant callback function.

Examples of synchronous and asynchronous client API are given in Listing 7 and Listing 8. In the observe example in Listing 9, a recursive function with selective receive spawned as a new process is used to illustrate how notifications can be received and how observe can be canceled. This is not the only way and unnecessary if being used within an OTP process where messages are processed one after another.

```
1  -module(ecoap_client_sync).
2  -export([example/0].
3
4  example() ->
5      % use a random port and default protocol transport parameters
6      {ok, Client} = ecoap_client:open("coap://example.com:5683"),
7      Response = ecoap_client:get(Client, "/HelloWorld"),
8      % print the response in console
9      case Response of
10         {ok, {ok, Code}, Content} -> io:format("Response with success code: ~p ~p~n", [Code, Content]);
11         {ok, {error, Code}, Content} -> io:format("Response with error code: ~p ~p~n", [Code, Content]);
12         {error, Reason} -> io:format("Fail: ~p~n", [Reason])
13     end,
14     % wrap the function call in a catch statement so the error can be captured instead of crash the caller
15     % when the request times out
16     case catch ecoap_client:put(Client, "/HelloWorld", <<"some payload">>,
17     #{'Content-Format' => <<"text/plain">>}, 5000) of
18         % request timeout
19         {'EXIT', {timeout, _}} -> io:format("Request time out after 5 seconds~n");
20         % the call failed due to other reason and should crash the caller as normal
21         {'EXIT', Reason} -> exit(Reason);
22         Response -> Response
23     end,
24     ok = ecoap_client:close(Client).
```

Listing 7: Synchronous client API

```erlang
-module(ecoap_client_async).
-export([example/0].

example() ->
    % use a random port and default protocol transport parameters
    {ok, Client} = ecoap_client:open("coap://example.com:5683"),
    {ok, Ref} = ecoap_client:get_async(Client, "/HelloWorld"),
    receive
        {coap_response, Ref, Client, Response} ->
            % successfully received response
            % do something
    after 5000 ->
            % optionally, specify a timeout after which the request is considered failed
            % do something
    end,
    {ok, Ref2} = ecoap_client:get_async(Client, "/HelloWorld"),
    ok = ecoap_client:cancel_request(Client, Ref2),  % cancel the asynchronous request just issued
    ok = ecoap_client:flush(Ref2),  % flush in case the response is delivered already
    ok = ecoap_client:close(Client).
```

Listing 8: Asynchronous client API

```erlang
-module(ecoap_client_observe).
-export([example/0].

example() ->
    % use a random port and default protocol transport parameters
    {ok, Client} = ecoap_client:open("coap://example.com:5683"),
     % start observe and receive notification in another process
    Pid = spawn(fun() -> start_observe(Client) end),
    % ...
    Pid ! proactive_cancel,  % send the cancel command
    % ...
    ok = ecoap_client:close(Client).

start_observe(Client) ->
    {ok, Ref} = ecoap_client:observe(Client, "/observable"),
    get_notification(Client, Ref).

% the {coap_notify, ...} and {coap_response, ...} message is sent by the ecoap_client process
get_notification(Client, Ref) ->
    receive
        {coap_notify, Ref, Client, Seq, Response} ->  % Seq is observe sequence number
            % do something
            get_notification(Client, Ref);  % continue the receive loop
        {coap_response, Ref, Client, Response} ->     % when the resource is deleted/not observable
            % do something
        proactive_cancel ->
            {ok, Ref2} = ecoap_client:unobserve(Client, Ref),  % cancel observe by issuing a GET request
            receive
                {coap_response, Ref2, Client, Response} ->     % the response is same as from a normal GET
                    % do something
            end;
        reactive_cancel ->
            ecoap_client:cancel_request(Ref)
    end.
```

Listing 9: Observe API

Chapter 5

Evaluation

The ecoap prototype implementation is used to evaluate the proposed system architecture for scalable and reliable IoT services. Benchmarks are performed against the state of the art CoAP implementation for both unconstrained and constrained environments. The evaluation scenarios, however, can vary depending on the many different communication models in IoT applications, including pure server, pure client and complex logic combining both of them [63].

For instance, a service could consist of resource directories (RDs) which are servers that manage the devices and provide discovery. The devices would first register themselves thereby issuing a request and periodically update their status [62]. Then other devices and services could contact the resource directory server for lookups. The devices can also be servers while a cloud service takes the role of client and observes resources hosted on many devices for monitoring and sensing. A combination of the two such as the OMA Lightweight M2M [78] specification acts as a resource directory and proxy at the same time, which means it not only receives registration and look-up requests from devices, but also issues requests to the resources of the devices. The devices in such case take both roles as well, since they expose their data as resources while using requests to register with the service.

To better evaluate the scalability and reliability of the proposed implementation, the pure server scenario is chosen because the other scenarios can have various concurrency requirements which are highly application dependent. Moreover, it allows for a direct comparison with the other mainstream CoAP implementation, the Californium (Cf) CoAP framework [18].

5.1 Experiment Setup

5.1.1 Benchmark Tool

It is considered that in a typical IoT scenario, many endpoints would exchange small messages with certain services, while each request and response is treated as a single, compact unit of information. Therefore the number of information that can be exchanged per time and latency are the most important factors for the measurement of a server.

However, there is few benchmark tool for CoAP measuring these properties. The one used by Californium is CoAPBench [19], which is a Java-based application similar to ApacheBench [2] and can be distributed over

multiple machines. It is decided to develop an Erlang counterpart which follows its test logic. This is for two purposes. Firstly, with the concurrency model of Erlang, the benchmark tool can achieve a higher level of load on one node thus avoid using multiple machines and simplify the test process. Second and the most important, the original CoAPBench does not provide a clear latency tracking functionality which is necessary in this evaluation, while integrating one with CoAPBench is not a trivial task. It is easier to reuse the components of ecoap to satisfy such a requirement.

Similar with CoAPBench, the Erlang benchmark tool uses virtual clients for concurrency factor. A virtual client is a simplified CoAP client which can be easily implemented as one Erlang process. It is argued that since CoAP is a connection-less protocol, it does not make a lot of difference whether messages come from one endpoint or many endpoints. However, the many virtual clients model shows more similarity to real-world use cases since it not only simulates all independent endpoints but also obeys the stop-and-wait nature of the protocol. High message rates in the IoT usually comes from millions of devices sending occasionally with intervals in minute or hour, instead of a small number sending at a very high rate constantly [63]. More importantly, the fault-tolerance of ecoap is built based on the assumption that the server communicates with endpoints each having a different identification (e.g. address). Therefore the virtual clients model also helps verify the fault tolerance behaviour in the experiment. A brief architecture of the benchmark tool can be found in Figure 5.1.

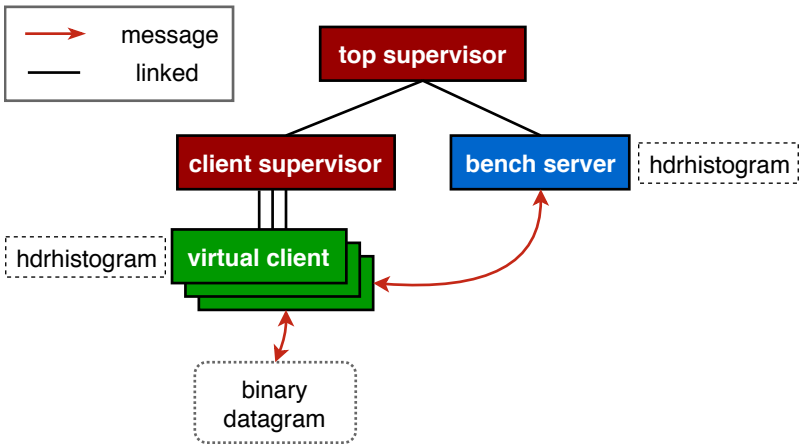


Figure 5.1: The supervision tree of the Erlang CoAP benchmark tool. A virtual client is a simplified CoAP client which only sends request in a closed-loop fashion, that is, waiting for a response of the previous request before sending next one. Each virtual client holds a HDR histogram which records response time of every request. The bench server process holds another HDR histogram which aggregates the contents of all virtual clients after a test has finished.

The benchmark tool works in a closed-loop fashion and adheres to basic congestion control. That is to say, each virtual client sends requests to the server as fast as the server can handle them: it sends confirmable requests and always waits for the response before the next request is issued. Retransmission is disabled in order to not blur the number of sent and successfully handled requests. A timeout of 10 seconds is applied in case of a message loss, which is recorded in a separate counter. After that, the virtual client continues with

a new request. Round trip time for each request is stored using Erlang ports of the High Dynamic Range (HDR) Histogram [56], which supports the recording and analyzing of sampled data value with dynamic range and pre-defined precision in constant memory footprint. One can further specify the number of virtual clients used in one test run and how long the test lasts. Throughput is therefore calculated as the sum of responses divided by the period of time of the test run. The benchmark tool reports the throughput, the number of timeouts and statistically interesting latency measurements including minimum, maximum and percentile values after the test completes.

5.1.2 Setup

The main interest of this work is the performance of the server on handling the protocol instead of complex business logic. Therefore the server under test simply holds a `/benchmark` resource which responds with a piggybacked "hello world" payload to confirmable GET requests. A growing number of concurrent clients are used to achieve desired concurrency factor, which increases stepwise from 10 to 10,000. All clients stress the server for 60 seconds with requests and then comes a 30 seconds cool-down interval. Test for each concurrency factor is performed five times. Server throughput is presented as average value while latency for each test run is recorded separately.

All benchmarks are performed using the Erlang benchmark tool. The test platform varies and is given for the individual experiments respectively.

5.2 Multi-core Scalability

This section presents the results of the experiments with ecoap and Californium (release 1.0.6), with focus on the investigation of scalability with respect to the number of concurrently active endpoints that communicate with the server and/or with respect to the number of available CPU cores. The experiments are conducted under unconstrained and constrained environment respectively.

5.2.1 Unconstrained Environment

The evaluation under unconstrained environment is done by measuring the throughput and latency on virtual machines with increasing capability in a cloud. The unconstrained cloud is chosen as the target platform because it should better reveal the scalability of the server compared with a constrained one, especially how well the server utilizes modern multi-core systems. Virtual machines hosted on Amazon Web Service (AWS) are used for the test since it is highly configurable. In order to reduce the disturbance in such a virtualized environment, all benchmarks run on dedicated instances [5] where no other users could share the resource simultaneously.

The benchmark tool is running on a m4.4xlarge instance with 16 vCPUs and 64 GB of RAM, while the server under test is running on an instance with increasing capability for each run, both with Ubuntu Server

16.04 LTS installed. As shown in Figure 5.2, the initial configuration of the server host is a m4.large instance with 2 vCPUs and 8 GB of RAM. For further tests the host is improved to a m4.xlarge instance with 4 vCPUs and 16 GB of RAM, a m4.2xlarge instance with 8 vCPUs and 32 GB of RAM, and a m4.4xlarge instance with 16 vCPUs and 64 GB of RAM, respectively. The client instance and the server instance are connected to one subnet exclusively through enhanced networking (up to 10 Gbps). Note that a vCPU usually refers to a single hardware hyper-thread and therefore in the scope of this work a CPU core means a logical CPU core instead of a physical one. More detailed information of aforementioned AWS instances can be found in [6].

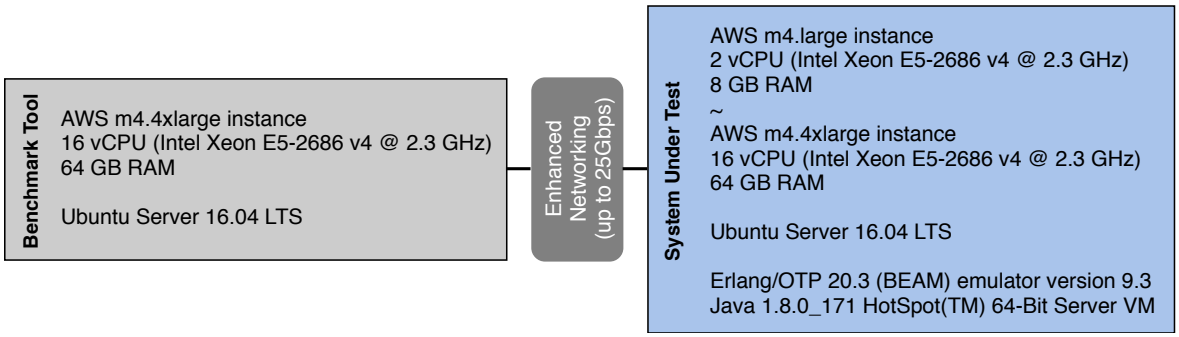
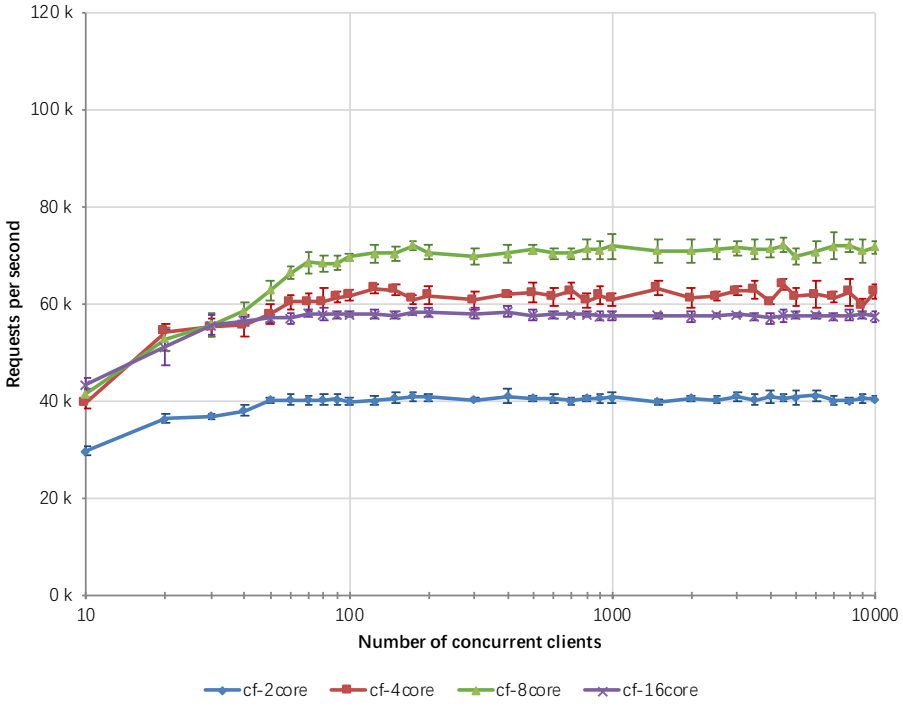


Figure 5.2: Experiment setup in unconstrained environment: The platform for the system under test starts with AWS m4.large instance and is improved for each series of tests, up to AWS m4.4xlarge instance. In general each capability upgrade doubles available vCPUs and RAM.

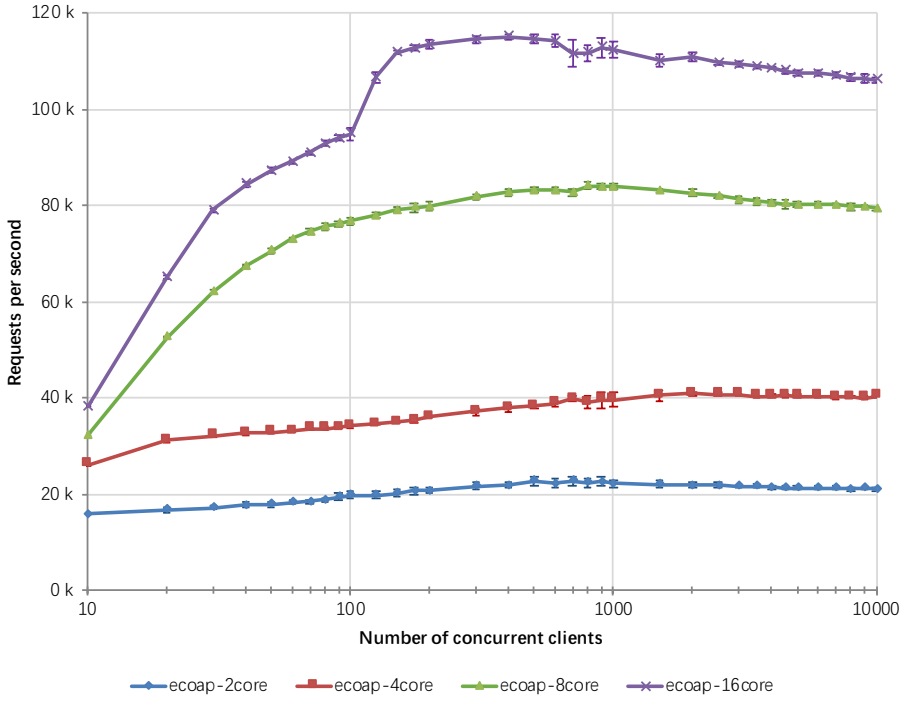
Since the server has to cache requests and corresponding responses to detect duplicates, the memory consumption soon becomes the majority of the load. Though CoAP allows relaxing duplicate detection for idempotent requests, it is decided to strongly reduce the lifetime of confirmable requests rather than completely disable the deduplication, because the latter removes most state management pressure which makes the experiment even further from any real-world use case. As a result, the EXCHANGE LIFETIME is tweaked to 1.5 seconds which leaves the server enough time to clean up all stale states during the interval between each test run, meanwhile ensuring the state management still be a part of the ordinary server activity.

The maximum number of open file descriptors is increased to allow enough active UDP sockets. The socket buffer is increased to 1 MB for both servers to reduce unnecessary message loss. The Erlang runtime is configured to bind all schedulers to available CPU cores and use kernel poll provided by the operating system.

Figure 5.3 shows the throughput of the two servers with increasing capability of the underlying instance. The same scale is used for both figures. Overall message loss is less than 1% and has a negligible influence on the experiment. With 2 cores and 4 cores, Californium scales well and outperforms ecoap. Its maximum throughput is about 1.5 times higher on 4 cores than on 2 cores. Though ecoap doubles the throughput of itself when upgrading from 2 cores to 4 cores, it still can not go beyond Californium, which almost has 2 times better performance. However, when the configuration reaches 8 cores and higher, ecoap scales much better than Californium. Californium performs around 1.2 times better on 8 cores than 4 cores, but performs



(a) Californium



(b) ecoap

Figure 5.3: Throughput on an AWS instance with increasing capability: ecoap is less performant with lower configuration but scales better with higher configuration. Californium scales well under 8 cores but does not perform well beyond that. It also has higher variance.

poorly on 16 cores. The throughput drops to a lower level than itself on 4 cores. On the other hand, ecoap doubles its throughput from 4 cores to 8 cores and has a 1.4 times improvement from 8 cores to 16 cores. With 8 cores, maximum throughput for Californium and ecoap is 72,000 requests per second and 84,000 requests per second, respectively. While with 16 cores, the two numbers are 58,000 requests per second and 110,000 requests per second. It is expected that with the increasing number of available cores, improvements become less obvious since not all tasks can be well parallelized. With that being said, the above results still show that ecoap has satisfying vertical scalability under unconstrained environment.

One observation from the test is Californium performs better with lower configuration. It also stabilizes at around 100 concurrent clients, earlier than ecoap which usually stabilizes after 500 clients. This can be explained as the overhead brought by the Erlang runtime, which always strives to schedule all processes fairly. As a result, the server can not fully utilize the advantage of the runtime without enough available resources and workload. In contrast, Californium has configurable threading model which might fit in such situations well.

Another observation is that Californium has a higher standard deviation during the test, as indicated by the error bars in the figure. This may have many reasons. After observing the system resource consumption during the experiments, it is found that the Erlang runtime occupies more CPU and saturate faster than the Java Virtual Machine (JVM), while the JVM frees memory much slower than the Erlang one. As a result, Californium consumes much more memory than ecoap, especially after long time testing. It is confirmed through both the Linux process viewer `htop` and the `jvisualvm` [61] virtual machine profiling tool. There is no related memory leak being observed though, as the local test proved that message exchange states are successfully removed after corresponding lifetime. It is inferred that the JVM needs more time to invoke a full garbage collection with a large RAM, meanwhile the high concurrency level leads to large amount of objects being created and deleted frequently, which eventually influenced the performance of the server. In addition, the virtual environment on AWS might have an undesired impact on the experiment. In contrast, the Erlang VM uses a per-process based generational garbage collection strategy. As the name indicates, it runs inside each Erlang process independently, making the VM release resource sooner after finishing the task and avoid stop-the-world freeze on applications as much as possible.

As mentioned before, Erlang uses various strategies to achieve soft-realtime, which has the most obvious impact on the latency of a high concurrency server. Figure 5.4 and Figure 5.5 presents the minimum round-trip time with respect to increasing concurrency factor and increasing instance capability. Both implementations give low enough latency while Californium has slightly better results. ecoap shows a more obvious trend that the latency decreases when more CPU cores are available. However, as Figure 5.6 and Figure 5.7 indicate, the maximum round-trip time starts to vary. The maximum round-trip time of ecoap is more centralized and gradually grows with the number of concurrent clients, while Californium has a more scattered result that is usually an order of magnitude larger than ecoap. With more CPU cores, the maximum latency of both servers generally goes down as expected. But even with 16 cores, Californium

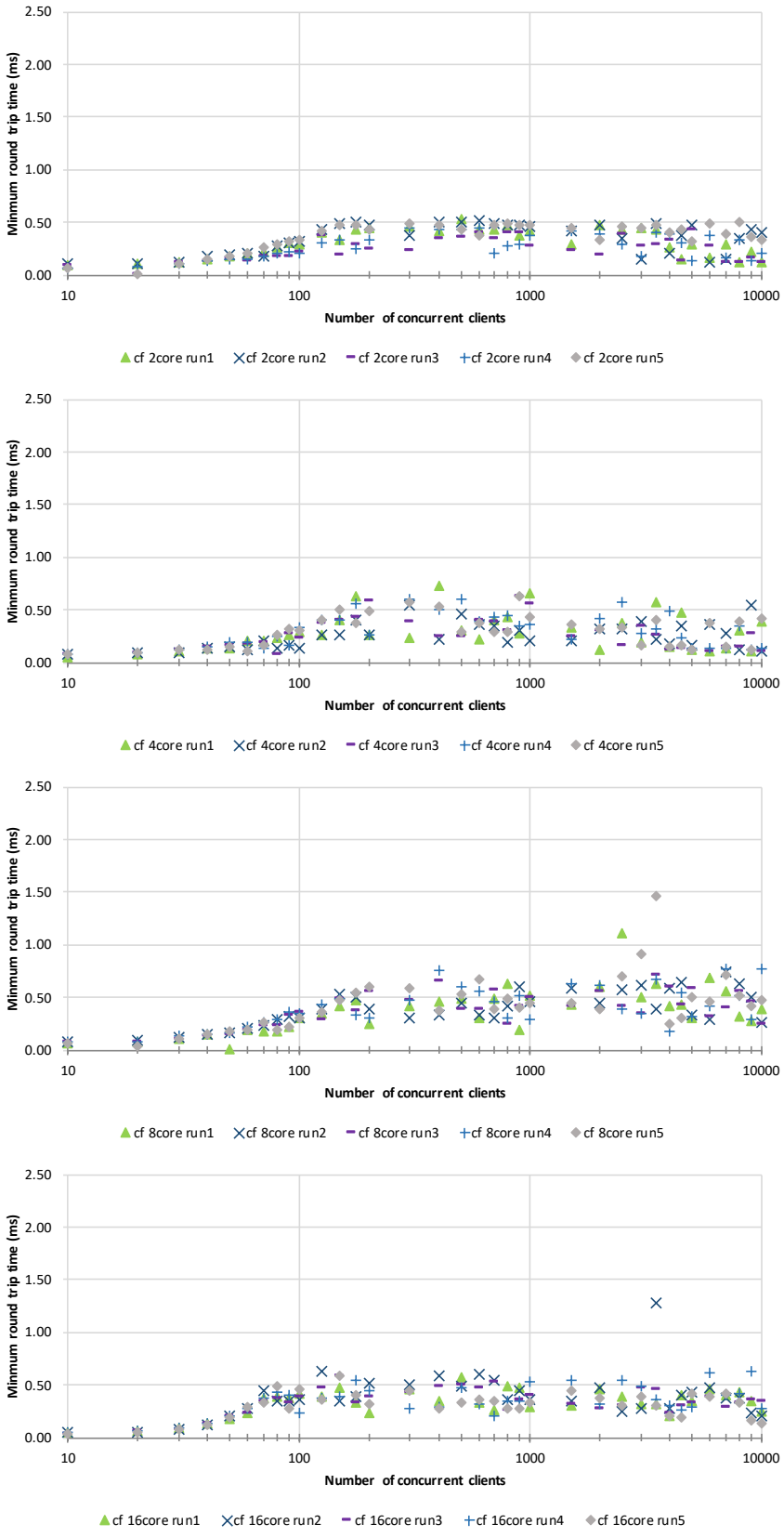


Figure 5.4: Minimum latency of Californium on an AWS instance with increasing capability: 2 cores, 4 cores, 8 cores and 16 cores from top to bottm

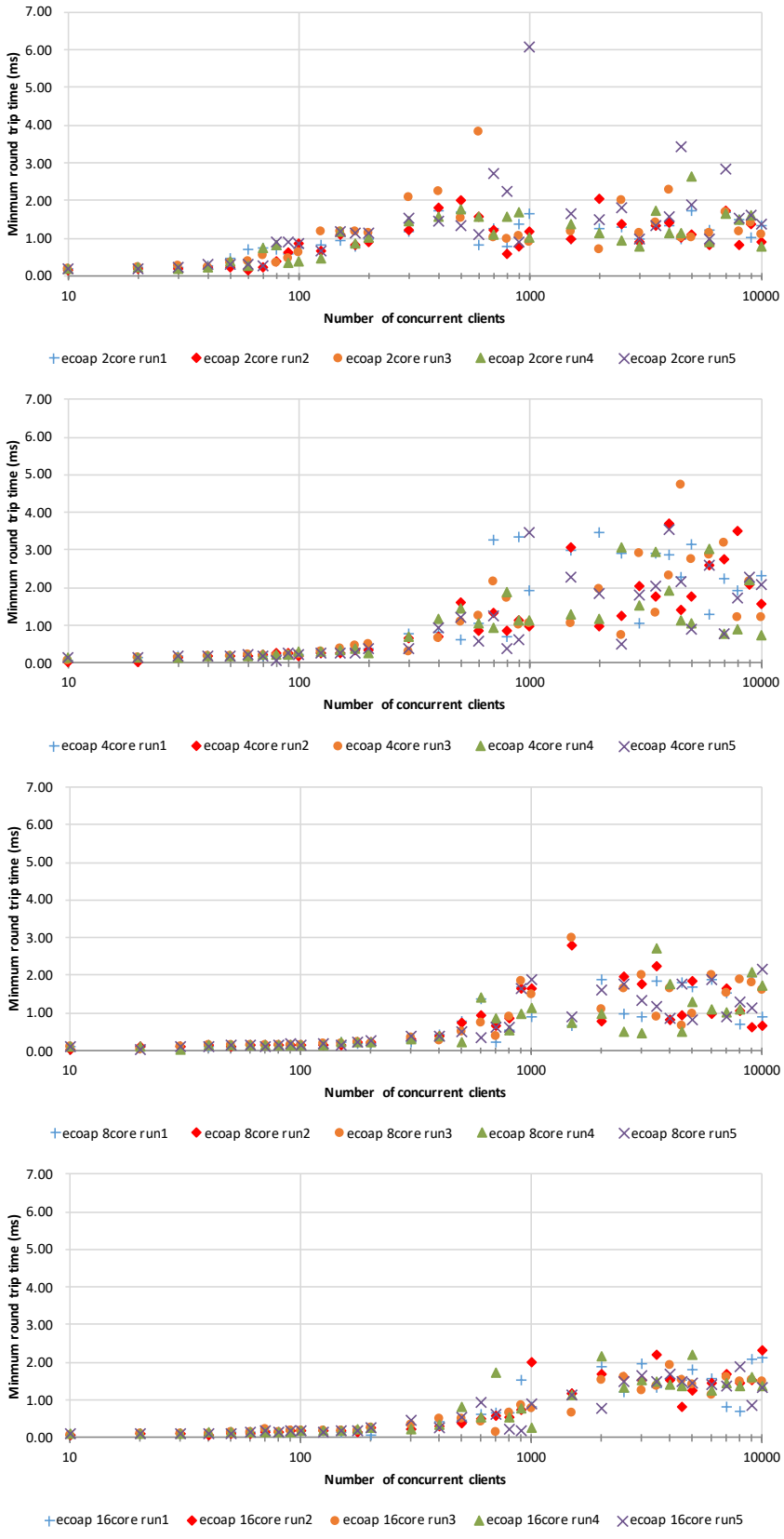


Figure 5.5: Minimum latency of ecoap on an AWS instance with increasing capability: 2 cores, 4 cores, 8 cores and 16 cores from top to bottom

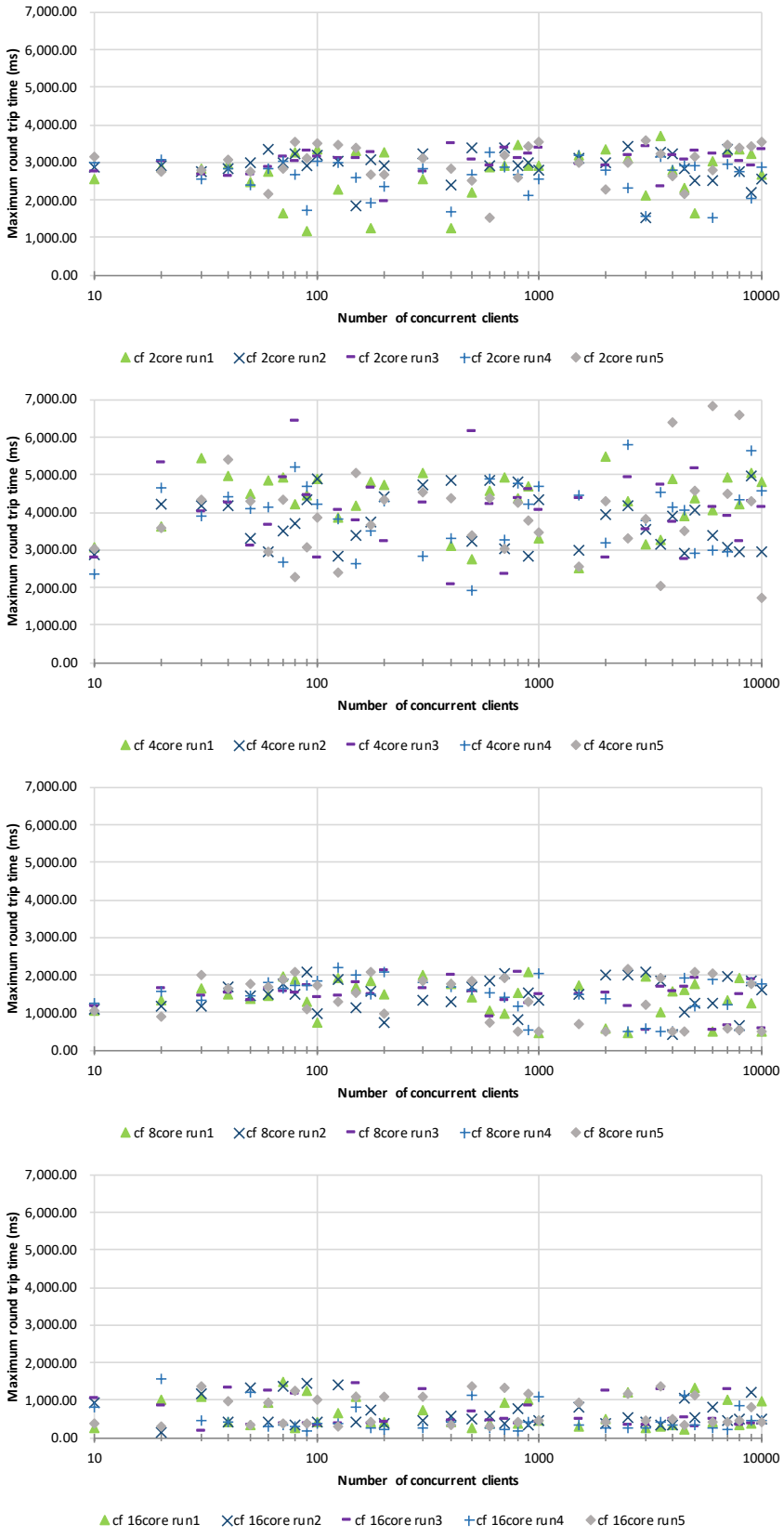


Figure 5.6: Maximum latency of Californium on an AWS instance with increasing capability: 2 cores, 4 cores, 8 cores and 16 cores from top to bottom

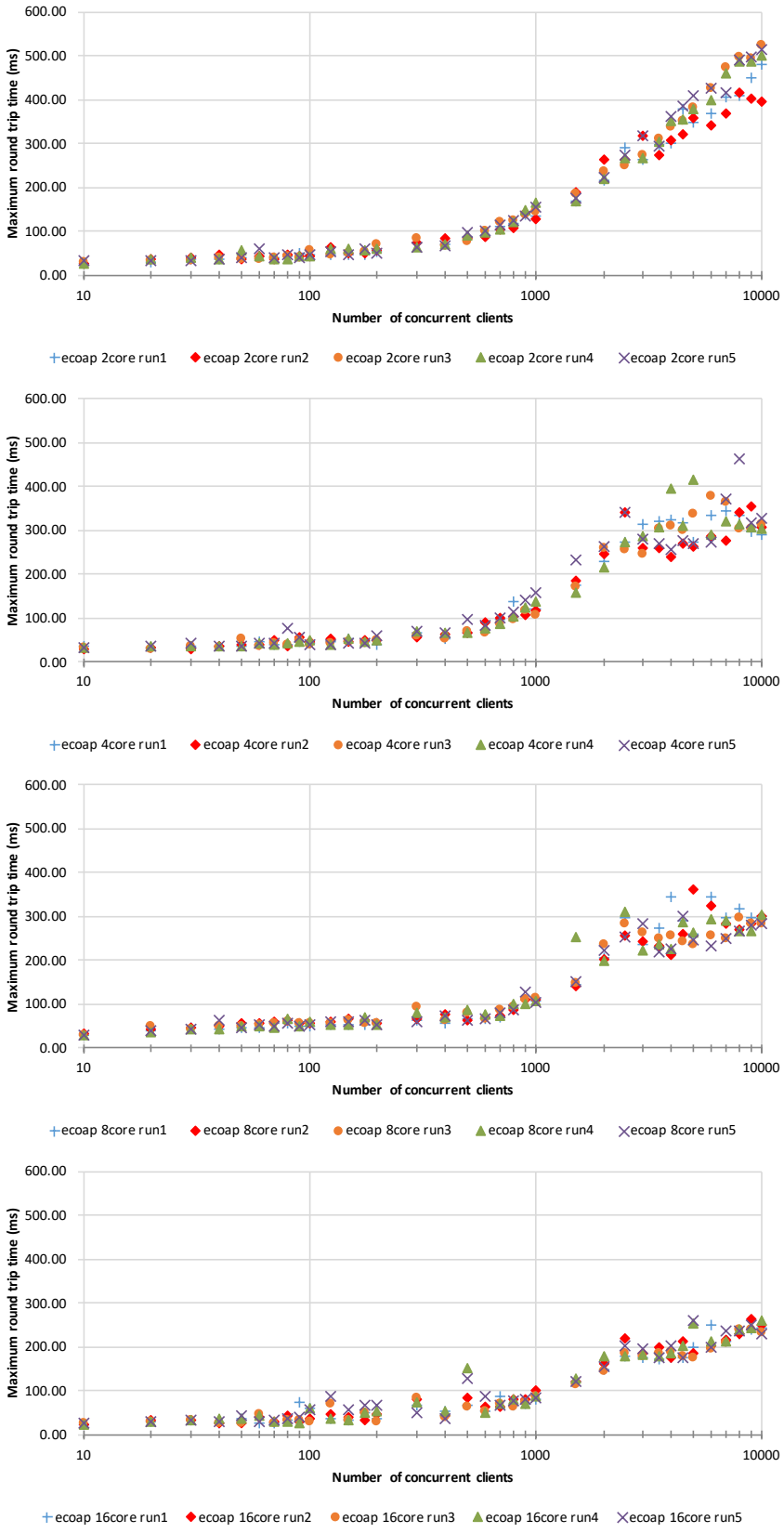


Figure 5.7: Maximum latency of ecoap on an AWS instance with increasing capability: 2 cores, 4 cores, 8 cores and 16 cores from top to bottom

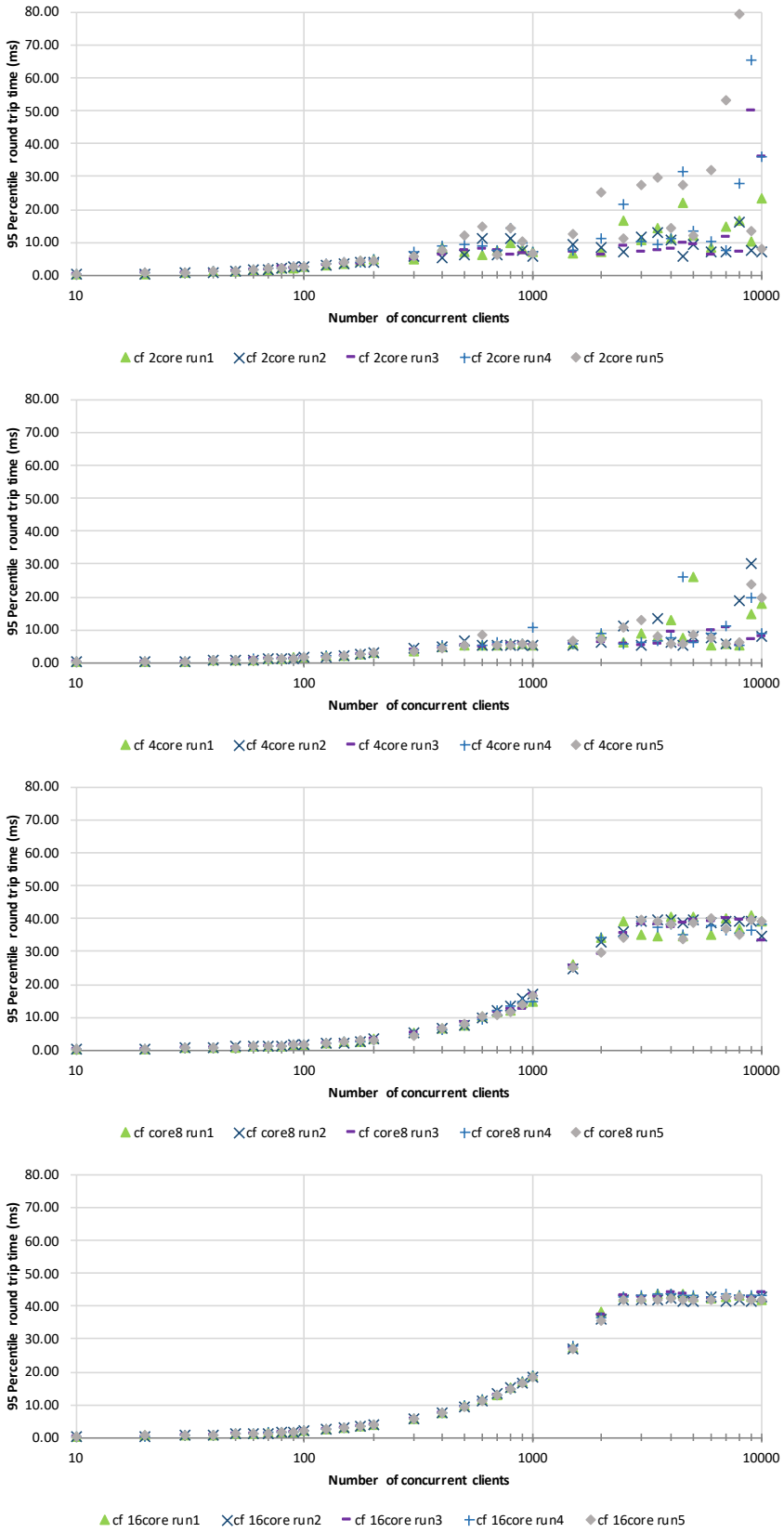


Figure 5.8: 95 percentile latency of Californium on an AWS instance with increasing capability: 2 cores, 4 cores, 8 cores and 16 cores from top to bottom

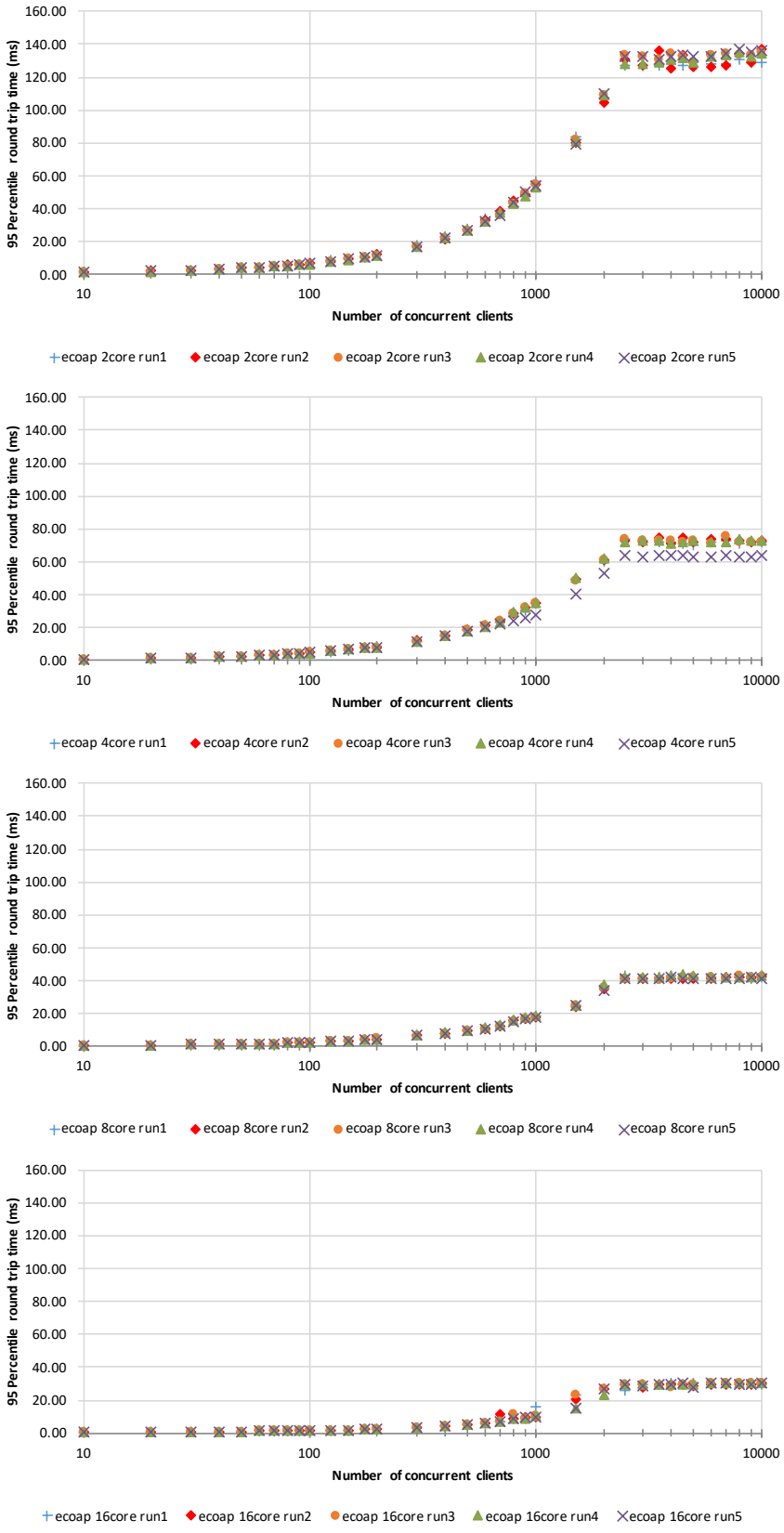


Figure 5.9: 95 percentile latency of ecoap on an AWS instance with increasing capability: 2 cores, 4 cores, 8 cores and 16 cores from top to bottom

still has maximum latency over 1 second from time to time whereas ecoap controls all latency below 300 milliseconds under the same condition. The difference of the results might also be related to the difference between garbage collection and memory management of the two implementations.

Figure 5.8 and Figure 5.9 gives the 95 percentile latency of the two servers. The 95 percentile latency gives the value that is larger than 95% of the entire dataset and is therefore a better metrics than pure average. As seen in the figures, both datasets have a trend which increases almost linearly as the concurrency level goes up (since the x-axis is of logarithmic scale). The latency values saturate at high concurrency factors (after 2,000 clients) because network limit has been reached. This is confirmed as message timeouts appear around the same point. An interesting fact is though Californium gives better latency than ecoap at the beginning, it does not improve a lot when the capability of underlying instance gets improved. On the other side, ecoap has a more predictable result where latency clearly drops given more computing power. When 16 CPU cores are provided, Californium ends up with 40 milliseconds and ecoap ends up with around 30 milliseconds. The 95 percentile latency proves both implementations can satisfy the timing requirements of the majority of tasks. Nevertheless, the Erlang runtime ensures ecoap has a more predictable behaviour, especially with worst cases.

It can be seen that the proposed server prototype has comparable performance in terms of throughput and latency to the mainstream implementation, Californium, when being deployed in the cloud environment. The ecoap server scales well with increasing CPU cores and shows an advantage when general low latency is desired.

5.2.2 Constrained Environment

It is interesting to see to what extend a solution could scale up and scale down when being targeted to platforms with great capability difference. Hence, a similar experiment is conducted in a more constrained environment, the Raspberry Pi 3 [83]. The Raspberry Pi is essentially an embedded computer which can run Linux compatible applications. It is never as constrained as sensors and low-power devices, however, such type of platforms are still widely used as gateways or local processing unit in many IoT applications. For instance, it is ideal for running a Fog node that encapsulates more dummy sensors and devices.

The Raspberry Pi 3 is equipped with a quad-core 1.2GHz Broadcom BCM2837 64bit CPU, with 1GB RAM and BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board. The experiment environment consists of one Pi running the CoAP server and a MacBook Pro running the benchmark tool, connected via a Gigabyte network switch, as shown in Figure 5.10. It is considered the MacBook Pro is powerful enough to generate loads that can saturate the Pi.

The test measures the average throughput as well as the latency of each server under stress, starting with 1 client and stepwise increasing the concurrency level to 10,000. It is of little interest to explore the vertical scalability on such a constrained platform. Therefore the evaluation is done with a fixed number of CPU cores. All other settings including the server tweaking and socket buffer tuning are the same as the AWS

experiment.

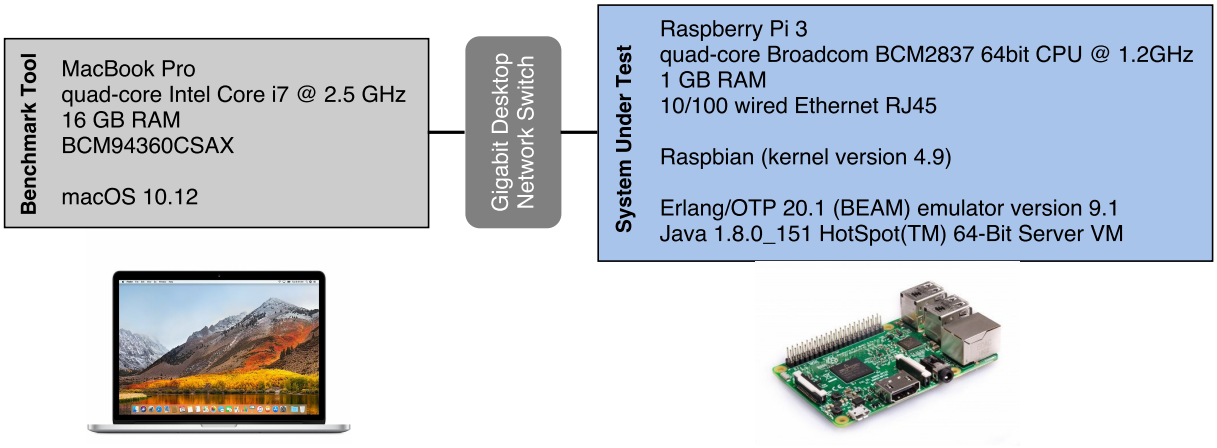


Figure 5.10: Experiment setup in constrained environment

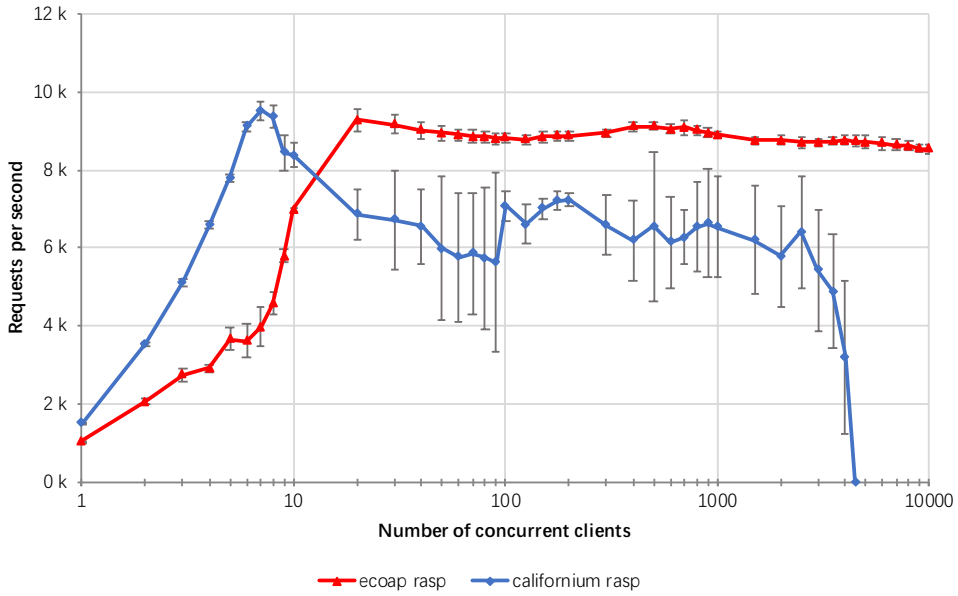


Figure 5.11: Throughput on a quad-core Raspberry Pi 3 at 1.2 GHz and 1GB RAM

Figure 5.11 shows the throughput of the two servers on the Raspberry Pi. The performance curves are not as stable as the ones under a cloud environment. This might be due to the limitation of the processing power of the Pi. ecoap achieves the highest throughput with about 9,000 requests per second. Interestingly, the throughput of Californium increases rapidly at low concurrency level but soon decreases afterward. The standard deviation is much higher than before, which implies it does not fully stabilize during the test. Californium eventually exits with an out of memory exception at the concurrency level of 4,500. Without further investigation, it is not obvious why Californium crashes during the test. It is clearly not designed for the constrained environment. But as stated in its introduction, Californium is also capable of running

on embedded platforms such as an Android smartphone, which provides similar processing ability as the Raspberry Pi. Again, it is most likely that without further tuning and optimization, the underlying JVM can not manage memory efficiently within such a limited RAM under high concurrency load. Several proofs can be found here. It is observed during the test that the JVM on Raspberry Pi starts in client mode by default, which renders the server crash at even lower concurrency level. Setting the VM to server mode and increase the heap size helps the server to run longer, but 4,500 clients are the final limitation here.

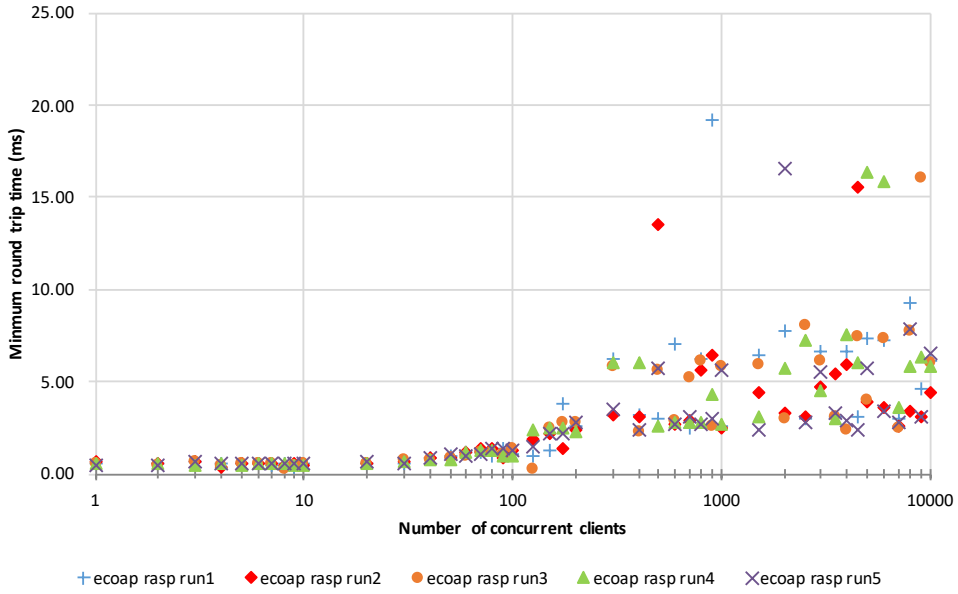


Figure 5.12: Minimum latency of ecoap on Raspberry Pi 3

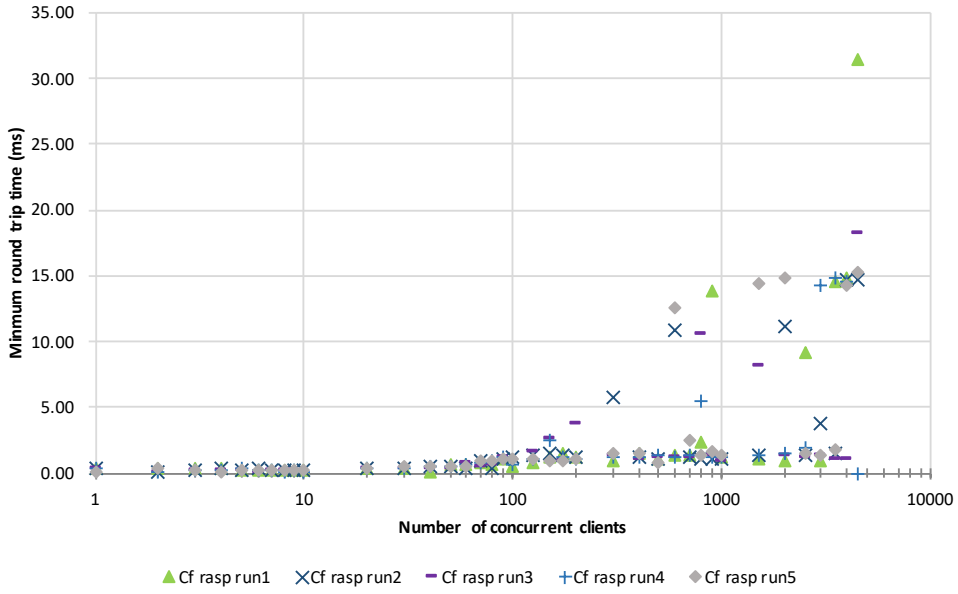


Figure 5.13: Minimum round trip time of Californium on Raspberry Pi 3

The general result resembles the trend of the previous experiment. Figure 5.12 and Figure 5.13 show the

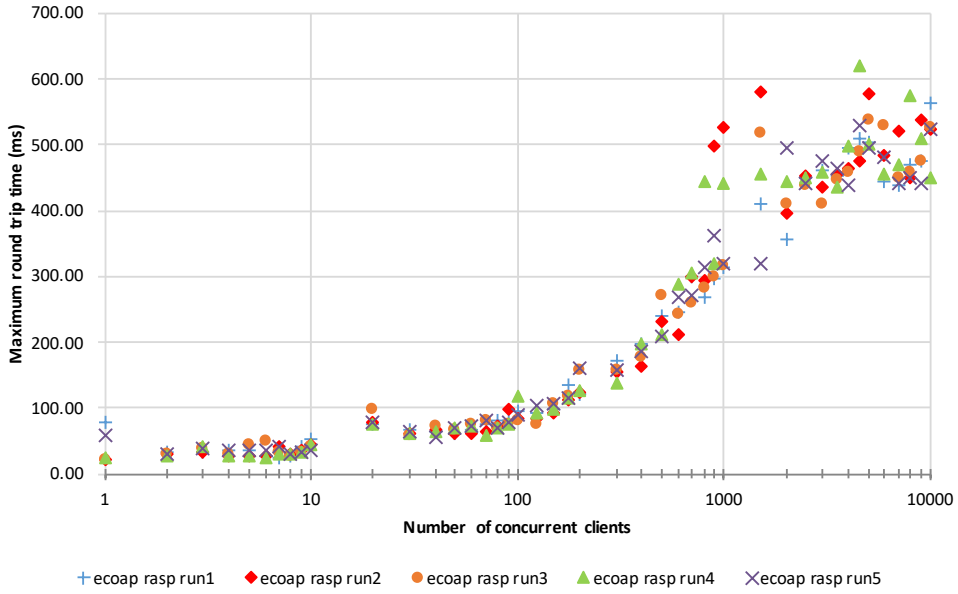


Figure 5.14: Maximum latency of ecoap on Raspberry Pi 3

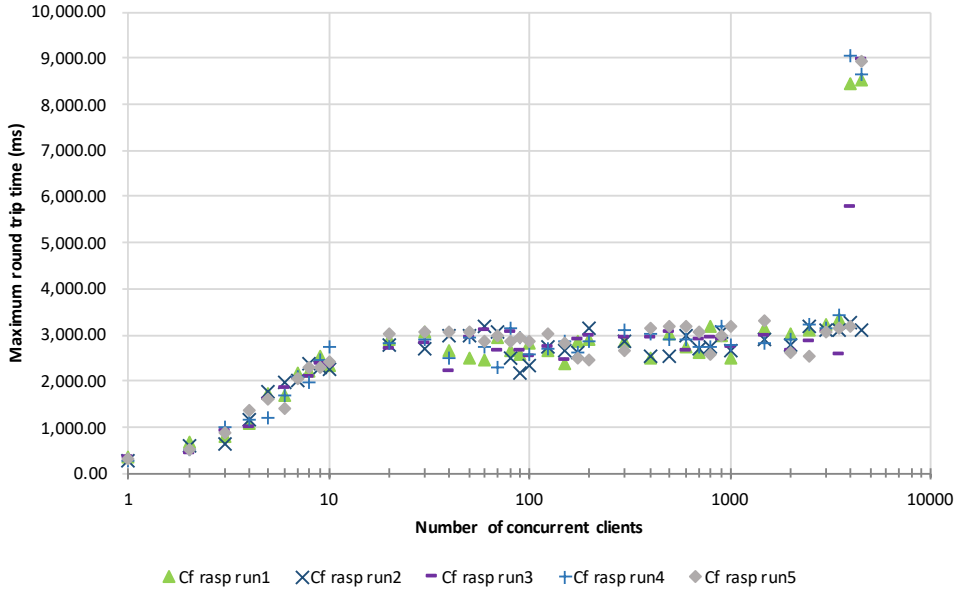


Figure 5.15: Maximum latency of Californium on Raspberry Pi 3

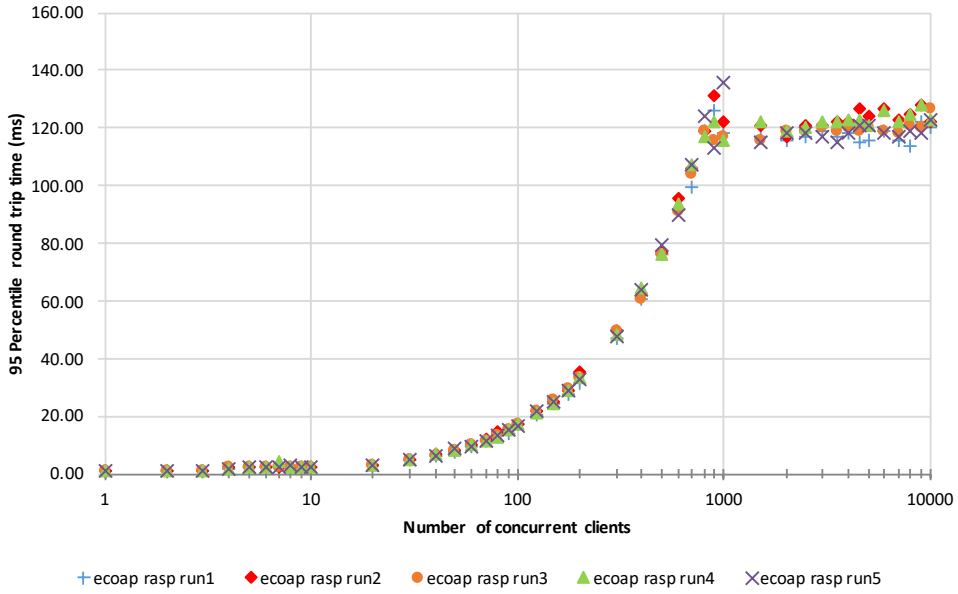


Figure 5.16: 95 percentile latency of ecoap on Raspberry Pi 3

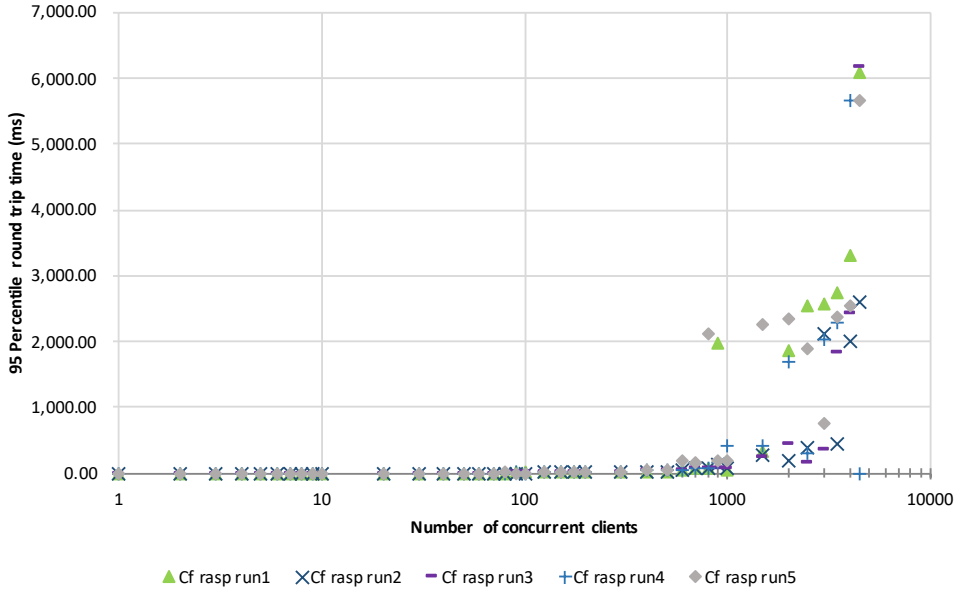


Figure 5.17: 95 percentile latency of Californium on Raspberry Pi 3

minimum latency clients can achieve in the experiment. When the concurrency level is high, both servers need around 20-30 milliseconds at least to process a request, which is reasonably much slower than the cloud. On the other hand, Figure 5.14 indicates a similar trend as the maximum latency of ecoap derived from the cloud experiment, though the largest value already exceeds 600 milliseconds. However, it is still more responsive compared to Californium, whose maximum latency is commonly over 2 seconds, as seen in Figure 5.15. When it comes to 95 percentile latency in Figure 5.16 and Figure 5.17, the performance of Californium is still acceptable, sometimes better than ecoap, until it hits a high concurrency factor. The ecoap server keeps stable during the intense stressing test with on average 3,000 more requests handled per second than Californium. Most of the requests are responded within 1 second even handling 10,000 concurrent clients.

To better illustrate the soft real-time characteristics of Erlang, a separate resource `\fibonacci` is hosted on the Raspberry Pi, which computes a fibonacci number in a highly inefficient recursive way. The input parameter of the computation is carried in the request query. Then 100 virtual clients would send requests to the `\fibonacci` resource in order to quickly saturate the server. Meanwhile another 100 clients would send requests to the `\benchmark` resource expecting responses as usual. Statistics of the non CPU intensive requests are given in Table 5.1. For Californium, all requests end up with timeout since the underlying threads that execute the CPU-intensive task block the whole server. However, under same load ecoap is still accessible for the non-CPU-intensive requests, at the cost of increasing response time and possible more timeout messages (though not shown in this test). It is the preemptive scheduling of Erlang that ensures no process should occupy too much CPU time and improves overall responsiveness. It is of no doubt that when eventually all CPU resource has been consumed, no server could process requests normally anymore. That being said, the Erlang runtime still improves the robustness of an application under heavy load.

	throughput	**timeout**	**min latency**	**max latency**	**95p latency**
Californium	0.00	100%	0.00	0.00	0.00
ecoap	404.93	0%	13.98	7405.57	546.00

Table 5.1: Result of fibonacci test on Raspberry Pi 3 which shows statistics of the requests hitting the benchmark resource. The throughput is measured as processed requests per second. Timeout shows the percentage of lost messages in all sent messages. All latency is measured in milliseconds. Californium is not responsive so no meaningful latency is recorded while ecoap is still available at the cost of degraded performance.

The concurrency model behind ecoap allows it to scale down to embedded platforms such as the Raspberry Pi without a problem. Since all concurrent activities are modeled as lightweight isolated processes, unnecessary synchronizations are avoided and the underlying runtime could schedule and manage the processes in a consistent manner. As a result, the benefits of fair scheduling and independent garbage collection that are discussed in the unconstrained experiment also apply here and have more explicit effects. Though other solutions can be highly customizable as well, for example, Californium supports the setting of different

number of threads in each processing stage. But as what the experiment results show, the limitations of the underlying concurrency model still reduces the overall flexibility.

5.3 Fault-Tolerance Test

Fault-tolerance is another design goal of ecoap. In order to verify the behaviour under various faults and failures, the faults need to be injected into a running server in a random manner. Chaos Monkey is a resiliency tool that helps applications tolerate random failures [23]. It originates from Netflix and aims at testing the fault-tolerance of a production environment by randomly terminating virtual machine instances and containers. That being said, this tool works at a coarse-grained level and does not provide finer control within one application instance. However, one Erlang application inspired by Chaos Monkey can do the trick. The application which is also called chaos monkey [24] can randomly terminate an Erlang process within another application at pre-defined rate. It therefore effectively tests the stability of the target application with any combination of failures.

The experiment is conducted on an AWS m4.2xlarge instance with 8 vCPUs and 32 GB of RAM with other settings the same as the previous AWS tests. To evaluate how fault-tolerant ecoap is, the experiment first starts the server and the benchmark tool as normal, and then randomly kills a process of the server every 2 seconds during an ongoing stress test. The interval of 2 seconds is the minimum value that can be used respecting the restart limit of supervisors inside ecoap. The chaos monkey is smart enough to only kill worker processes because it is where the majority of errors happen, while the supervisors are too strong to kill. Usually the crash of a supervisor implies an unrecoverable error just occurred and the entire supervision tree is abandoned.

In such a way, the throughput of the server under “attack" is shown in Figure 5.18. It can be seen clearly that when the chaos monkey application is active, the standard deviation gets quite large at low concurrency level. This is because, with low concurrency factor, fewer processes run in the server and killing one of them every 2 seconds is considered a non-negligible obstacle. The server survives the attack but cannot run as stable as usual. Nevertheless, the situation gets better when more concurrent clients are involved, since a system with a larger amount of processes is naturally resilient to fixed rate failure. Therefore, the throughput gets closer to the one without any fault as the number of concurrent clients grows.

Figure 5.19, Figure 5.20 and Figure 5.21 show the minimum, maximum and 95 percentile latency during the test respectively. It should be noticed that compared with the data with no fault, only slight difference exists. The server is overall not greatly influenced by the injected faults. On the other hand, the message timeout rate does not provide much information here. Though with injected faults there are message timeouts from the beginning of the test, the timeout percentage is in general similar to normal one or even lower. This is reasonable because the virtual clients work in a closed-loop fashion. The clients send fewer requests in total as a result of the faults, so as the number of timeout messages. Thus the percentage of lost messages

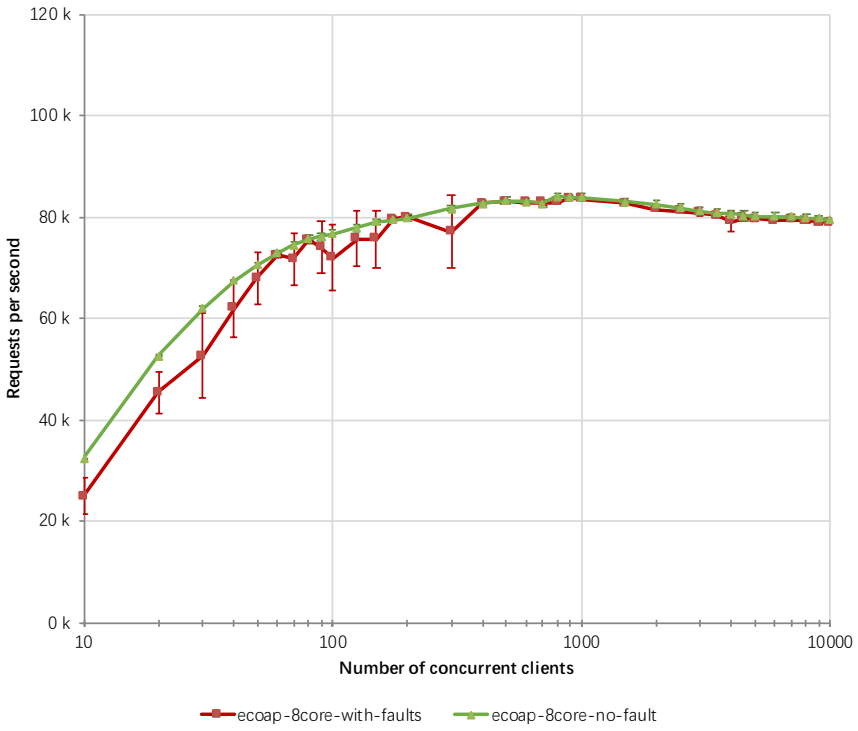


Figure 5.18: Throughput of ecoap with faults injected on an 8-core AWS instance: The chaos monkey application randomly kill a worker process of ecoap every 2 seconds. The performance is more disturbed at low concurrency factor.

will not change dramatically.

It is interesting enough to see that ecoap can keep providing service with such a high failure rate. Moreover, since the killed process is randomly selected, the experiment should prove that the system is recoverable and resilient no matter which stage went wrong. Without Erlang's share-nothing isolation and supervision tree mechanism, this fine-grained fault-tolerance is hard to achieve. It can be perceived that the high reliability of a single node would make it easier to build a high availability multi-node system.

Similar tests cannot be conducted with Californium, as it makes no sense to randomly terminate a thread within a threading concurrency model. It also means that in order to improve reliability, application with the traditional model must use defensive programming, carefully managed locks and asynchronous calls. While this can fit specific circumstances very well and outperform the proposed concurrency model, however, it is essentially not as intuitive as the latter and comes with much more maintainability overhead.

5.4 Discussion on Horizontal Scalability

Though one of the primary goals of this work is to verify vertical scalability of the proposed architecture, it is of interest how a language with transparent distribution support like Erlang deals with horizontal scalability. However, what Erlang provides are general tools for building distributed applications while the usage and

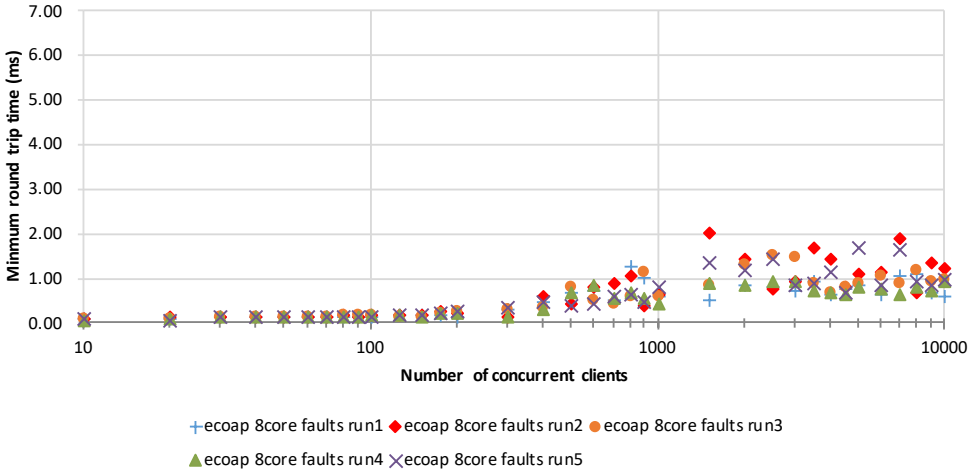


Figure 5.19: Minimum latency of ecoap with faults injected on an 8-core AWS instance

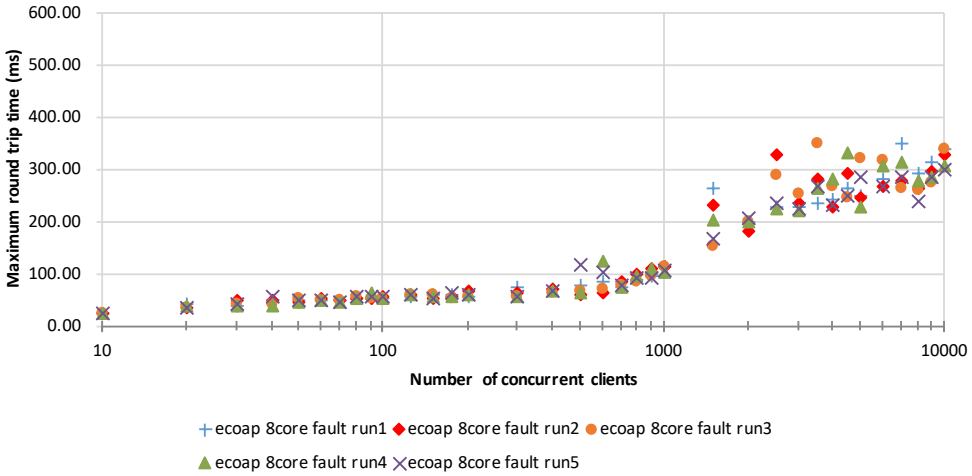


Figure 5.20: Maximum latency of ecoap with faults injected on an 8-core AWS instance

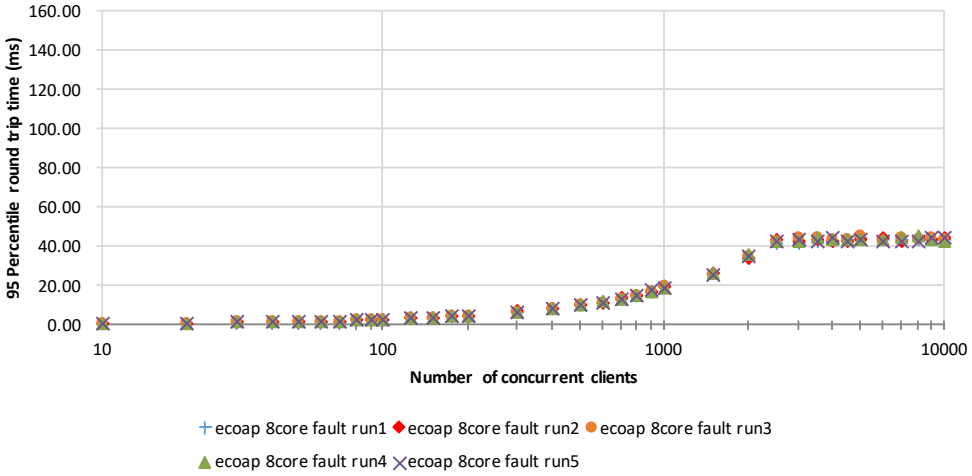


Figure 5.21: 95 percentile latency of ecoap with faults injected on an 8-core AWS instance

range of application vary a lot.

When it comes to a CoAP server, firstly common patterns for scaling Web servers should apply, such as adding a load balancer in front a farm of servers. Since this approach does not require synchronization among different server instances, all servers can simply run concurrently and therefore maximize the performance benefit. It should be noted that one difference between CoAP servers and HTTP servers is some extensions of CoAP are essentially stateful. For example, observe and block-wise transfer both require maintaining states on the server side. If the implementation does not employ a certain type of distributed storage then these states have be local on each server. Therefore a load balancer that supports "sticky" mode is preferred, since it can forward subsequent requests from one client always to the same target server.

For ecoap, block-wise GET requests can still work without the support of "sticky" mode because it is a safe operation [89]. The downside is reduced efficiency as multiple copies of the resource might be read by a number of servers. Block-wise PUT/POST can not work without this support due to the fact that they are implemented following the atomic fashion [17]. Similarly, an extra deduplication mechanism has to be provided for observe to avoid duplicate observe relations being established over multiple servers.

During the experiments of this work, a preliminary attempt was made to utilize the transparent distribution of Erlang to dispatch requests instead of using a load balancer. The attempt involves a small modification to the server so that it can act as a dispatcher while still being a functional server. It is simply done by exposing the API for starting the sub supervision tree that represents a remote endpoint. After starting multiple Erlang VMs each on a different machine and connecting them into a mesh network using the tool provided by the runtime, one can have the ecoap application running on each node. Then one of the server instances can act as the entry point and receive incoming requests. For any new client, the socket manager on that node calls the exposed API to start the endpoint supervision tree randomly on one of the connected nodes and forward the request to it, just as it does with a local endpoint supervision tree. After requests have been processed, all responses have to be sent back the entry socket manager, which then send them over the network. This is because sockets cannot be shared among nodes. Despite this, all other functionalities that use message passing underneath work in the same way as before. Because of the fact that Erlang uses the same model for concurrency and distribution, the above modification does not include any large refactoring but manages to dispatch requests evenly among many servers.

However, some simple tests immediately reveal the limitation of this approach. In order to have a comparison with the load balancer approach, all tests are conducted on Google Compute Engine [51] since it supports built-in UDP load balancing while AWS does not. The virtual machine instance for test consists of 2 vCPUs and 7.5 GB RAM (n1-standard-2 instance). Initial tests show that with load balancing, ecoap server gives the throughput of around 2,3000 requests/second with one instance, around 4,5000 requests/second with two instances and around 7,2000 requests/second with three instances, all stressed with 1,000 concurrent clients using the Erlang CoAP benchmark tool running on another powerful instance. As expected throughput increases almost linearly with this approach. While with the server-dispatching approach, the throughput

goes to 2,2000 requests/second with one instance, 3,1000 requests/second with two instances and 3,5000 requests/second with three instances. The second approach is clearly less performant and the bottleneck is obvious: the single socket manager still has to handle all network lifting (even worse than usual since it also handles the outgoing traffic). Furthermore, Erlang messages that are sent to remote process involves network latency, which has to be counted for every request whose processing logic is non-local.

It is considered that without more complex business logic, it is of little benefit to scale the CoAP server out utilizing distributed Erlang. In such a case, using a load balancer directly is more efficient. Nonetheless, distributed Erlang can largely simplify application logic when the system does involve many different components. For instance, a real-world case might use the server in pair with mnesia [72], the Erlang's built-in database that can be configured to replicate its state over multiple nodes for scalability and reliability using nothing else but pure distributed Erlang.

CoAP is a relatively young protocol and clustering of CoAP services is still an area of active research. Leshan [36] is an OMA Lightweight M2M (LWM2M) implementation in Java. It is built on top of the Californium framework. There is a demand for using Leshan servers in a cluster and it faces similar challenges as it uses CoAP underneath. The wiki page of the project [100] states information and concerns for CoAP clustering, which suggests that there is still much room for improvement in this area.

5.5 Summary

This chapter answers the question of how CoAP can be efficiently implemented using a concurrency-oriented language like Erlang, and how the system can automatically scale up to powerful cloud environment as well as scale down to constrained platform. The ecoap prototype gives an architecture for scalable and reliable IoT services. Furthermore, a detailed evaluation of the proposed prototype in both constrained and unconstrained environment is presented. The ecoap prototype consists of reusable components for both server and client, which then forms a flexible concurrency model where protocol processing is clearly divided and parallelized. Details of different components and how they work together are also explained in this chapter, with an emphasis on modelling and managing various states efficiently within CoAP constraints. The evaluation shows that the concurrency model fits modern multi-core systems well. The implementation has similar or better performance compared with mainstream CoAP solution.

In unconstrained environment, ecoap has comparable level of scalability as Californium with respect to a growing number of concurrent clients. Both implementations maintain a high throughout even with high concurrency factor and only latency increases. However, ecoap delivers in general more stable and responsive performance as a result of the underlying concurrency model. The latency results indicate that ecoap is more suitable for applications with real-time requirements, since it ensures the worst cases are still within an acceptable range. Furthermore, ecoap shows overall better scalability with respect to increasing computing power, which is a more valuable feature in multi-core and many-core era.

In constrained environment, ecoap outperforms Californium in both throughput and latency. It automatically scales down without extensive optimization while keeping consistent behaviour as in unconstrained environment. Other solutions may have a concurrency model that is more advanced in certain scenarios, however, they might not benefit from it when the environment changes. ecoap and its concurrency model gain more flexibility in general.

Moreover, ecoap is more resilient in terms of unexpected faults and failures. The fine-grained control over errors and faults and recover process can not be easily achieved in other solutions. As the number of participants of the IoT increases rapidly, such feature becomes more important because it gives more alternatives to achieve reliability.

Chapter 6

Conclusion

IoT devices often exchange real-time sensing and control data as small but numerous messages, requiring any backend service handling the huge amount of devices be extremely scalable and reliable to support large-scale applications. To cope with the challenge, new paradigms such as the Fog Computing has been studied, which attempts to distribute possible centralized bottleneck. The new trend raises a new question besides the existing scalability problem, that is, to what extent could the same solution scales up and down while showing a consistent behaviour.

In this thesis, in order to explore the possibility of building such scalable and reliable IoT services using concurrency-oriented language, an Erlang based CoAP server/client prototype called ecoap is presented. Following idiomatic design patterns, the ecoap architecture is divided to socket manager, endpoints and handlers, which can represent any number of real world concurrent activities. As a result, better scalability and reliability can be achieved into different environments. Processing chain of the protocol is separated in different components. Pattern matching of Erlang naturally fits with the binary encoded CoAP while message exchange states including deduplication and retransmission are implemented considering the dynamic nature of the language. APIs are provided in a straight forward manner and allows implementing customized functionality that exposes a RESTful interface to outside world without much effort. It is possible to send requests synchronously or asynchronously to CoAP endpoints, and establish observe relations. The client component can also be combined with a server instance which enables complex business logic. Moreover, the consistent concurrency model of Erlang implies the business logic can be implemented and tuned to any desired concurrent environment.

To benchmark the prototype server, an Erlang CoAP benchmark tool was developed, which is the simplified version of the existing Java benchmark tool, CoAPBench. To have a clearer impression of how the server performs, it is compared with the mainstream CoAP implementation, Californium. Results in unconstrained cloud environment indicate that ecoap has comparable performance to Californium. In particular, ecoap scales smoothly up to 16 cores, delivering stable throughput and low latency even when 10,000 clients concurrently communicate with the server. ecoap also shows in general better response time. In contrast, Californium reaches limitations of scaling up during the evaluation, and suffers from inconsistent latency. It is inferred that the JVM's concurrency model and memory management might not cope with the constraints of CoAP very well. On the other hand, ecoap outperforms Californium when both run on Raspberry Pi,

which is a constrained platform. ecoap has higher throughput and lower latency while Californium crashed under high load. It is considered that the problem of Californium which leads to its instability in cloud environment eventually made the server discontinued on a platform with too limited resources. Whereas the fair scheduling and process independent garbage collection ensures ecoap to be responsive and efficient in memory management in this case. In general, ecoap gives consistent behaviour in both environments and can automatically scale up and scale down. The flexibility of such a model reduces the effort of building scalable services. Moreover, the fault-tolerance test shows ecoap could keep service available even under high failure rate. It is considered as an evidence that the idea behind Erlang applies to IoT applications, which greatly eases the burden of achieving high reliability.

The IoT essentially requires more scalability and reliability since it shares similarity with the Web while being a more complex and dynamic system. The thesis demonstrates that the philosophy of Erlang does not only apply to large-scale Web systems but also to the emerging IoT, and how combination of the two can lead to scalable and reliable IoT services.

Limitations and Future Work

There are primarily four limitations of this work. First, the scope of this thesis is limited to vertical scaling, which means optimizing performance that can be achieved on a single machine. Though the results show the prototype could already handle a large amount of devices, horizontal scaling is necessary to further improve system capacity and reliability. It is of more practical value to continue to investigate this area, possibly based on the preliminary discussions in section 5.4.

Second, the experiments in unconstrained environment is conducted using virtualized cloud service, which might have unexpected effect on certain applications. Although it is a common trend to deploy services in cloud and the experiments revealed performance in such a scenario, it might be more appropriate to study the behaviours in a fully controlled environment such as a dedicated physical machine.

Third, fault-tolerant tests conducted in this work is still in its initial stage. That is to say, a more sophisticated scenario has to be developed for faults injection instead of fixed rate error. Furthermore, considering the first point, faults under a distributed environment might be of greater research interest.

The last limitation is security aspects are not considered in this thesis, in particular, the DTLS layer. The extra layer might add undesired overhead to the system. Similar experiments need to be run with the secured version of CoAP against other implementations in further studies.

References

[1] S. B. I. (Firm) and N. I. C. (U.S.) *Disruptive Civil Technologies: Six Technologies with Potential Impacts on US Interests Out to 2025 : Biogerontechnology, Energy Storage Materials, Biofuels and Bio-based Chemicals, Clean Coal Technologies, Service Robotics, the Internet of Things*. National Intelligence Council, 2008. URL: https://books.google.ca/books?id=6qi-nQAACAAJ.

[2] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html. (Visited on 02/2018).

[3] G. A. Agha. Actors: a model of concurrent computation in distributed systems, 1986.

[4] Akka - a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala. https://akka.io. (Visited on 02/2018).

[5] Amazon EC2 Dedicated Instances. https://aws.amazon.com/ec2/purchasing-options/dedicated-instances/. (Visited on 02/2018).

[6] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/. (Visited on 02/2018).

[7] ARM mbed Client Guide. https://docs.mbed.com/docs/mbed-client-guide/en/latest/. (Visited on 02/2018).

[8] J. Armstrong. Concurrency oriented programming in erlang. *Invited talk, FFG*, 2003.

[9] L. Atzori, A. Iera and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2010.05.010. URL: http://www.sciencedirect.com/science/article/pii/S1389128610001568.

[10] A. Banks and R. Gupta. MQTT Version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf, OASIS, November 2014.

[11] T. Berners-Lee, R. Fielding and L. Masinter. RFC 3986 Uniform Resource Identifier (URI): Generic Syntax. https://tools.ietf.org/html/rfc3986, Internet Engineering Task Force (IETF), January 2005.

[12] A. B. Bondi. Characteristics of Scalability and Their Impact on Performance. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 195–203, Ottawa, Ontario, Canada. ACM, 2000. ISBN: 1-58113-195-X. DOI: 10.1145/350391.350432. URL: http://doi.acm.org/10.1145/350391.350432.

[13] F. Bonomi, R. Milito, P. Natarajan and J. Zhu. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*, pages 169–186. Springer, 2014.

[14] F. Bonomi, R. Milito, J. Zhu and S. Addepalli. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, Helsinki, Finland. ACM, 2012. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. URL: http://doi.acm.org/10.1145/2342509.2342513.

[15] C. Bormann, A. P. Castellani and Z. Shelby. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Computing*, 16(2):62–67, March 2012. ISSN: 1089-7801. DOI: 10.1109/MIC.2012.29.

[16] C. Bormann, M. Ersue and A. Keranen. RFC 7228 Terminology for Constrained-Node Networks. https://tools.ietf.org/html/rfc7228, May 2014.

[17] C. Bormann and Z. Shelby. RFC 7959 Block-Wise Transfers in the Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7959, Internet Engineering Task Force (IETF), August 2016.

[18] Californium (Cf) CoAP framework. http://www.eclipse.org/californium/. (Visited on 02/2018).

[19] Californium (Cf) Tools. https://github.com/eclipse/californium.tools. (Visited on 02/2018).

[20] G. Candea and A. Fox. Crash-Only Software. In *HotOS*, volume 3, pages 67–72, 2003.

[21] Canopus. https://github.com/zubairhamed/canopus. (Visited on 02/2018).

[22] cantcoap – CoAP implementation that focuses on simplicity by offering a minimal set of functions and straightforward interface. https://github.com/staropram/cantcoap. (Visited on 02/2018).

[23] Chaos Monkey - a resiliency tool that helps applications tolerate random instance failures. https://github.com/Netflix/chaosmonkey. (Visited on 02/2018).

[24] chaos_monkey - Erlang application for stability test. https://github.com/dLuna/chaos_monkey. (Visited on 02/2018).

[25] M. Chiang and T. Zhang. Fog and IoT: An Overview of Research Opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, December 2016. ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2584538.

[26] S. Cirani, M. Picone and L. Veltri. mjCoAP: An open-source lightweight Java CoAP library for Internet of Things applications. In *Interoperability and Open-Source Solutions for the Internet of Things*, pages 118–133. Springer, 2015.

[27] coap – A CoAP Python library. https://github.com/openwsn-berkeley/coap. (Visited on 02/2018).

[28] CoAP – Constrained Application Protocol – Implementations. http://coap.technology/impls.html. (Visited on 02/2018).

[29] CoAPSharp. http://www.coapsharp.com/. (Visited on 02/2018).

[30] CoAPthon. https://github.com/Tanganelli/CoAPthon. (Visited on 02/2018).

[31] Constrained Application Protocol – Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Constrained_Application_Protocol#Implementations. (Visited on 02/2018).

[32] Constrained Application Protocol Client and Server for go. https://github.com/dustin/go-coap. (Visited on 02/2018).

[33] Copper (Cu) CoAP user-agent. https://github.com/mkovatsc/Copper. (Visited on 02/2018).

[34] Data Distribution Services Specification, V1.4. http://www.omg.org/spec/DDS/1.4/, Object Management Group (OMG), March 2015.

[35] David - a CoAP server with Rack interface. https://github.com/nning/david. (Visited on 02/2018).

[36] Eclipse Leshan - an OMA Lightweight M2M (LWM2M) implementation in Java. https://github.com/eclipse/leshan. (Visited on 02/2018).

[37] Eclipse tinydtls. https://projects.eclipse.org/projects/iot.tinydtls. (Visited on 02/2018).

[38] Erlang. http://www.erlang.org/. (Visited on 02/2018).

[39] Erlang Advanced Topics - Process Dictionary. https://www.erlang.org/course/advanced#dict. (Visited on 02/2018).

[40] Erlang pg2 Failure Semantics. http://christophermeiklejohn.com/erlang/2013/06/03/erlang-pg2-failure-semantics.html. (Visited on 02/2018).

[41] C. Esposito, S. Russo and D. D. Crescenzo. Performance assessment of OMG compliant data distribution middleware. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008. DOI: 10.1109/IPDPS.2008.4536566.

[42] C. Esposito. Data distribution service (DDS) limitations for data dissemination wrt large-scale complex critical infrastructures (LCCI). *MobiLab, Università degli Studi di Napoli Federico II, Napoli, Italy, Tech. Rep*, 2011.

[43] ets - Erlang built-in term storage. http://erlang.org/doc/man/ets.html. (Visited on 02/2018).

[44] P. T. Eugster, P. A. Felber, R. Guerraoui and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: http://doi.acm.org/10.1145/857076.857078.

[45] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[46] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. RFC 2616 Hypertext Transfer Protocol – HTTP/1.1. https://tools.ietf.org/html/rfc2616, Internet Engineering Task Force (IETF), June 1999.

[47] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2444095.

[48] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[49] Generic Erlang CoAP Client/Server. https://github.com/gotthardp/gen_coap. (Visited on 02/2018).

[50] Golang. https://golang.org. (Visited on 02/2018).

[51] Google Compute Engine. https://cloud.google.com/compute/. (Visited on 02/2018).

[52] J. Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986.

[53] K. Hartke. RFC 7641 Observing Resources in the Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7641, September 2015.

[54] F. Hébert. *Learn You Some Erlang for Great Good!* William Pollock, 2013.

[55] R. Hiesgen, D. Charousset and T. C. Schmidt. Embedded Actors - Towards distributed programming in the IoT. In *2014 IEEE Fourth International Conference on Consumer Electronics Berlin (ICCE-Berlin)*, pages 371–375, September 2014. DOI: 10.1109/ICCE-Berlin.2014.7034296.

[56] High Dynamic Range HDR Histogram for Erlang/OTP, Elixir & LFE. https://github.com/HdrHistogram/hdr_histogram_erl. (Visited on 02/2018).

[57] J. Hui and P. Thubert. RFC 6282 Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. https://tools.ietf.org/html/rfc6282, Internet Engineering Task Force (IETF), September 2011.

[58] U. Hunkeler, H. L. Truong and A. Stanford-Clark. MQTT-S – A publish/subscribe protocol for Wireless Sensor Networks. In *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, pages 791–798, January 2008. DOI: 10.1109/COMSWA.2008.4554519.

[59] jCoAP. https://code.google.com/archive/p/jcoap/. (Visited on 02/2018).

[60] S. Jucker. Securing the Constrained Application Protocol. *no. October*:1–103, 2012.

[61] jvisualvm - Java Virtual Machine Monitoring, Troubleshooting, and Profiling Tool. https://docs.oracle.com/javase/7/docs/technotes/tools/share/jvisualvm.html. (Visited on 02/2018).

[62] M. Koster, Z. Shelby, C. Bormann and P. V. d. Stok. CoRE Resource Directory draft-ietf-core-resource-directory-12. https://tools.ietf.org/html/draft-ietf-core-resource-directory-12.html, October 2017.

[63] M. Kovatsch. *Scalable Web Technology for the Internet of Things*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 22398, 2015.

[64] M. Kovatsch, S. Duquennoy and A. Dunkels. A low-power CoAP for Contiki. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pages 855–860. IEEE, 2011.

[65] M. Kovatsch, M. Lanter and Z. Shelby. Californium: Scalable cloud services for the internet of things with coap. In *Internet of Things (IOT), 2014 International Conference on the*, pages 1–6. IEEE, 2014.

[66] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker and C. Görg. Implementation of CoAP and its application in transport logistics. *Proc. IP+ SN, Chicago, IL, USA*, 2011.

[67] N. Kushalnagar, G. Montenegro, D. E. Culler and J. W. Hui. RFC 4944 Transmission of IPv6 Packets over IEEE 802.15.4 Networks. https://tools.ietf.org/html/rfc4944, Internet Engineering Task Force (IETF), September 2007.

[68] M. Lanter. Scalability for IoT Cloud Services, 2013.

[69] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006. ISSN: 0018-9162. DOI: 10.1109/MC.2006.180. URL: http://dx.doi.org/10.1109/MC.2006.180.

[70] D. Lund, C. MacGillivray, V. Turner and M. Morales. Worldwide and regional internet of things (iot) 2014–2020 forecast: A virtuous circle of proven value and demand. *International Data Corporation (IDC), Tech. Rep*, 1, 2014.

[71] M. Michael, J. E. Moreira, D. Shiloach and R. W. Wisniewski. Scale-up x Scale-out: A Case Study using Nutch/Lucene. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007. DOI: 10.1109/IPDPS.2007.370631.

[72] Mnesia. http://erlang.org/doc/apps/mnesia/users_guide.html. (Visited on 02/2018).

[73] H. Muller. A CoAP Server with a Rack Interface for Use of Web Frameworks such as Ruby on Rails in the Internet of Things, 2015.

[74] nCoAP. https://github.com/okleine/nCoAP. (Visited on 02/2018).

[75] M. Nottingham. RFC 5988 Web Linking. https://tools.ietf.org/html/rfc5988, October 2010.

[76] M. Nottingham and E. Hammer-Lahav. RFC 5785 Defining Well-Known Uniform Resource Identifiers (URIs). https://tools.ietf.org/html/rfc5785, Internet Engineering Task Force (IETF), April 2010.

[77] Observer User's Guide. http://erlang.org/doc/apps/observer/users_guide.html. (Visited on 02/2018).

[78] OMA Lightweight M2M. http://openmobilealliance.org/iot/lightweight-m2m-lwm2m. (Visited on 02/2018).

[79] OpenWSN. http://www.openwsn.org/. (Visited on 02/2018).

[80] G. Pardo-Castellote, B. Farabaugh and R. Warren. An introduction to DDS and data-centric communications. *Real-Time Innovations. OpenURL*, 2005.

[81] T. Pötsch, K. Kuladinithi, M. Becker, P. Trenkamp and C. Goerg. Performance Evaluation of CoAP Using RPL and LPL in TinyOS. In *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on*, pages 1–5, May 2012. DOI: 10.1109/NTMS.2012.6208761.

[82] Project Loom: Fibers and Continuations for the Java Virtual Machine. http://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html. (Visited on 02/2018).

[83] Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. (Visited on 02/2018).

[84] E. Rescorla and N. Modadugu. RFC 6347 Datagram Transport Layer Security Version 1.2. https://tools.ietf.org/html/rfc6347, Internet Engineering Task Force (IETF), January 2012.

[85] J. Sanchez-Monedero, J. Povedano-Molina, J. M. Lopez-Vega and J. M. Lopez-Soler. Bloom filter-based discovery protocol for DDS middleware. *Journal of Parallel and Distributed Computing*, 71(10):1305–1317, 2011.

[86] T. Savolainen and B. Silverajan. CoAP Communication with Alternative Transports draft-silverajan-core-coap-alternative-transports-10. https://tools.ietf.org/html/draft-silverajan-core-coap-alternative-transports-10, July 2017.

[87] Scala. https://www.scala-lang.org. (Visited on 02/2018).

[88] Z. Shelby. RFC 6690 Constrained RESTful Environments (CoRE) Link Format. https://tools.ietf.org/html/rfc6690, Internet Engineering Task Force (IETF), August 2012.

[89] Z. Shelby, K. Hartke and C. Bormann. RFC 7252 The Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7252, Internet Engineering Task Force (IETF), June 2014.

[90] H. Shi, N. Chen and R. Deters. Combining Mobile and Fog Computing: Using CoAP to Link Mobile Device Clouds with Fog Computing. In *2015 IEEE International Conference on Data Science and Data Intensive Systems*, pages 564–571, December 2015. DOI: 10.1109/DSDIS.2015.115.

[91] A. Sivieri, L. Mottola and G. Cugola. Drop the Phone and Talk to the Physical World: Programming the Internet of Things with Erlang. In *Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications*, SESENA '12, pages 8–14, Zurich, Switzerland. IEEE Press, 2012. ISBN: 978-1-4673-1793-1. URL: http://dl.acm.org/citation.cfm?id=2667049.2667051.

[92] SMCP. https://github.com/darconeous/smcp. (Visited on 02/2018).

[93] S. Srinivasan. Kilim: A server framework with lightweight actors, isolation types and zero-copy messaging. Technical report UCAM-CL-TR-769, University of Cambridge, Computer Laboratory, February 2010. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-769.pdf.

[94] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP*, volume 8, pages 104–128. Springer, 2008.

[95] A. Stanford-Clark and H. L. Truong. MQTT For Sensor Networks (MQTT-SN) Protocol Specification Version 1.2. `http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf`, International Business Machines Corporation (IBM), November 2013.

[96] Supervisor Behaviour - OTP Design Principles. `http://erlang.org/doc/design_principles/sup_princ.html`. (Visited on 02/2018).

[97] G. Tanganelli, C. Vallati and E. Mingozzi. CoAPthon: Easy development of CoAP-based IoT applications with Python. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, pages 63–68, December 2015. DOI: `10.1109/WF-IoT.2015.7389028`.

[98] The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification - Version 2.2. `https://www.omg.org/spec/DDSI-RTPS/2.2/`, September 2014.

[99] Twisted framework. `https://twistedmatrix.com`. (Visited on 02/2018).

[100] Using Leshan server in a cluster. `https://github.com/eclipse/leshan/wiki/Using-Leshan-server-in-a-cluster`. (Visited on 02/2018).

[101] S. Vashi, J. Ram, J. Modi, S. Verma and C. Prakash. Internet of Things (IoT): A vision, architectural elements, and security issues. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 492–496, February 2017. DOI: `10.1109/I-SMAC.2017.8058399`.

[102] J. R. Von Behren, J. Condit and E. A. Brewer. Why Events Are a Bad Idea (for High-Concurrency Servers). In *HotOS*, pages 19–24, 2003.

[103] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero and M. Nemirovsky. Key ingredients in an IoT recipe: Fog Computing, Cloud computing, and more Fog Computing. In *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 325–329, December 2014. DOI: `10.1109/CAMAD.2014.7033259`.

[104] C. Zhou and X. Zhang. Toward the Internet of Things application and management: A practical approach. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*, pages 1–6, June 2014. DOI: `10.1109/WoWMoM.2014.6918928`.