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

Abstract

This is the abstract of my thesis.

Acknowledgements

Acknowledgements go here. Typically you would at least thank your supervisor.

Contents

List of Tables

List of Figures

List of Abbreviations

CoAP	Constrained Application Protocol
CoRE	Constrained RESTful environments
CSP	Communicating Sequential Processes
DDS	Data Distribution Service
DETS	Disk Erlang Term Storage
DICE	DTLS In Constrained Environments
DTLS	Datagram Transport Layer Security
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IoT	Internet of Things
M2M	machine-to-machine
MQTT	Message Queue Telemetry Transport
NAT	Network Address Translation
REST	Representative State Transfer
OTP	Open Telecom Platform
RTPS	Real-Time Publish-Subscribe
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

Chapter 1

Introduction

With the rise of the Internet of Things (IoT), more smart devices (sensors, actuators, etc.) will be deployed in people's life everywhere. 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 [8] have been proposed, which moves computing 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 cloud but must execute some complex tasks on behalf of many low-end devices puts some performance demand on it, similar to traditional backend services.

Another challenge is that existing communication protocols may not fit well in the IoT space. Typical IoT applications intend to consist of a lot of 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 constrained environment. Therefore popular web protocols like 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 CoAP [60], MQTT [7] and DDS [26]. Among them, The Constrained Application Protocol (CoAP), as a lightweight and efficient application protocol, targets the problem by using low-overhead UDP as its transport layer to ease the stress in constrained environment. It also embraces REST style so that interactions with existing web becomes easier. Beyond that, CoAP supports asynchronous message notification, a.k.a. Observe, to fit in subscriber models used by many IoT applications.

Regardless of paradigms and protocols, IoT applications in general place high requirements on infrastructures. Scalability and reliability become more important than ever. One constrained device may not be able to send large amount of data at once or send data very frequently, but millions of them could result in huge amount of ongoing data 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, uninterrupted and safe operation is often the top priority [17]. In another word, downtime due to failure of subsystems should be minimized. This is where reliability must be emphasized more than before.

An industry with totally different targets but similar requirements is telecommunication, since 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, thus 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 [28]. Erlang approached the problem by following the famous Actor Model [1], 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 COP (Concurrency-Oriented Programming) [4] and has influenced many subsequent languages such as Go [36] and Scala [58].

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 real world, the AXD301 is a fault-tolerant carrier-class ATM (Asynchronous Transfer Mode) switch manufactured by Ericsson Telecom AB, which has the measured reliability quoted as being 99.9999999% (9 nines) corresponding to a down time of 31 ms per year [4]. Nevertheless, very few research of applying Erlang or COP in general in the IoT area have been presented. A. Sivieri et al. [63] proposed an Erlang-based development framework called ELIoT which aims at coordination of wireless sensor network. R. Hiesgen et al. [40] introduced a modified Actor Model and runtime environment based on C++ to IoT application programming, where the authors listed Erlang as a design reference and gave an example implementation utilizing CoAP as communication channel among actors.

The similarity between the requirements of IoT applications and the design goal of the language leads us to this research. In order to explore the use space of COP language in the IoT world, we attempt to use the typical COP language Erlang to model a CoAP infrastructure (server and client) prototype 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 detailed definition of the research problem. Chapter 3 presents the literature review of the state-of-the-art work, including comparison of popular IoT application protocols, description of important features of CoAP that are relevant to this work and summary of existing CoAP implementations. It is followed by chapter 4 which argues the difference between the current paradigms of mainstream server-side design and typical Erlang applications first, and then presents the architecture of the proposed solution as well as implementation details. After that chapter 5 demonstrates a scalability benchmark (comparisons to Californium, a popular Java CoAP implemantation, on both constrained and cloud platform) and a fault-tolerance benchmark against ecoap itself. At 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 typical COP language like Erlang to model scalable and reliable IoT infrastructure utilizing CoAP?

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

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

1. Scalability can refer to vertical scalability and horizontal scalability. It is particularly interesting to evaluate horizontal scalability in IoT scenarios, especially when Erlang comes into play, since it is also famous for its transparent distribution support. However, horizontal scalability is also heavily determined by application specific requirements, which makes it difficult to model and test generally. Therefore, for simplicity, only vertical scalability is considered in this work. It is measured by how many concurrent requests a system can handle on average under certain load. Thus concurrency and scalability are used interchangeably in this work.

2. Reliability can refer to many things. In this work, availability is more a research interest. It is measured by observing whether a system behaves as expected when random faults happen within any of its subsystems, so that the whole system remains available. This is sometimes also referred 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 must be stable facing large number of concurrent clients/requests and behave as expected when faults occur. On the other hand, with Fog Computing paradigm in mind, the objective of the research can also be stated as building

1. How can The Constrained Application Protocol be implemented in the context of Erlang?

2. What architecture should be used to provide both concurrency and fault-tolerance?

3. What interface should be given for potential application developers for easy integration?

4. How can a benchmark tool be constructed to evaluate the implementation?

Chapter 3

Related Work

In this chapter, the background of the emerging Internet of Things (IoT) and Fog Computing is introduced in section 3.1, popular IoT protocols are discussed and compared with an emphasis on The Constrained Application Protocol (CoAP) in section 3.2 and 3.3, and at the end some existing implementations of CoAP are listed and discussed in section 3.3.6.

3.1 Internet of Things (IoT) and Fog Computing

The Internet of Things (IoT) is a novel paradigm that is rapidly gaining ground in the scenario of modern wireless telecommunications. The basic idea of this concept is the pervasive presence around us of a variety of things or objects – such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, mobile phones, etc. – which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbors to reach common goals, without human involvement [5]. The current revolution in Internet, mobile and machine-to-machine (M2M) technologies can be seen as the first phase of the IoT [33]. A growing number of physical objects are being connected to the Internet at an unprecedented rate realizing the idea of IoT. A basic example of such objects includes thermostats and HVAC (Heating, Ventilation, and Air Conditioning) monitoring and control systems that enable smart homes. Survey [5] gives a thorough analysis on the potential application areas in IoT, including but not limited to, transportation and logistics, healthcare, smart environment (home, office, plant), industrial automation and emergency response to natural and man-made disasters where human decision making is difficult. The US National Intelligence Council (NIC) foresees that “by 2025 Internet nodes may reside in everyday things – food packages, furniture, paper documents, and more” [5]. The IoT provides a great market opportunity for equipment manufacturers, Internet service providers as well as application developers. The IoT smart objects are expected to reach 212 billion entities deployed globally by the end of 2020 [34]. By 2022, M2M traffic flows are expected to constitute up to 45% of the whole Internet traffic [33]. Economic growth of IoT-based services is also considerable for businesses. Healthcare and manufacturing applications are projected to form the biggest economic impact. The whole annual economic impact caused by the IoT is estimated to be in range of $2.7 trillion to $6.2 trillion by 2025 [33].

Over the past decade, an important trend is moving computing, control, and data storage into the cloud. However, the emerging IoT introduces many new challenges that can not be adequately addressed by today‘s Cloud Computing models alone. These challenges [17] include but not limited to, stringent latency requirements under certain environment (such as industrial control systems), 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.

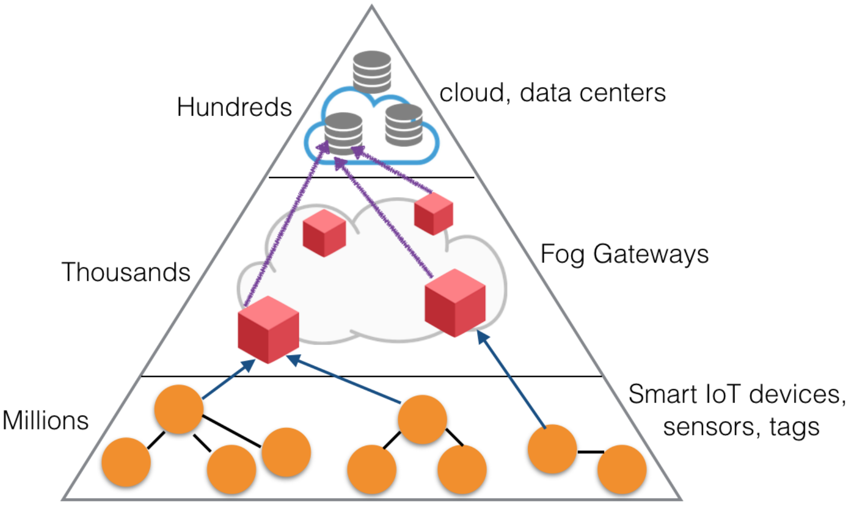


Figure 3.1: The role of the Cloud and Fog play in the delivery of IoT services. [33]

In order to filling the technology gaps in supporting the IoT, a new architecture - Fog - was first introduced. In contrast to traditional cloud model, where many sensors upload raw data directly to a central cloud infrastructure for further processing and analysis, Fog is a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing data centres, which, typically but not exclusively located at the edge of network [8]. 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 control centre 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 a large number of small Fog nodes may form a large Fog system [17].

Due to the characteristics of Fog, it enables IoT applications which require low-latency, location-awareness, real-time interactions and analytics and mobility support. Fog also enhances scalability and resilience of IoT applications by its nature [33]. Various services and computing tasks can run on a fog node, depending on how much resource a node owns and application requirements. Applications could on 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 can not be moved to cloud due to latency constraints or any other reason, 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 [8][17]. 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 as it tries to perform part of high level services which are offered by cloud inside the local resources [33]. Typical Fog applications include Connected Vehicle (CV), Smart Grid and wireless sensors (actuators) networks [8]. A real world success of Fog has been discussed in [17], 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. Scalability and reliability is still required on a Fog node especially when complex tasks and services being deployed. The Fog node is highly possible a less-constrained embedded device such as the Raspberry Pi [56], which has more resources than common sensors and actuators yet is much less powerful than the Cloud. Moreover, since CoAP is a standard M2M protocol, it surely has its use space under a Fog environment. Thus the combination is not only a valid use case in terms of Fog Computing, but also falls into the scope of this thesis. A detailed evaluation of the proposed CoAP server on Raspberry Pi is stated in chapter 5.

3.2 Application Protocols for IoT

The IoT will be made up of large number of heterogeneous devices, which are equipped with extremely diverse capabilities, in terms of processing power, connectivity, availability, and mobility. In order to effectively allow and foster the growth of new applications and services, it is necessary to provide appropriate standards that can guarantee full interoperability among existing hosts and IoT nodes [18].

Many standards around the IoT have been introduced to facilitate and simplify application programmers' and service providers' jobs. While many efforts have been made to bring Internet Protocol (IP) to all kinds of devices so that they can be part of the existing Internet, application layer protocols are another important part of these standards.

In general, current IoT application protocols can be divided into 3 types: message-oriented, data-oriented, and resource-oriented [61]. Representative protocols of the 3 types are Message Queue Telemetry Transport (MQTT) [7], Data Distribution Service for Real Time Systems (DDS) [26] and The Constrained Application Protocol (CoAP) [60]. 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)

Publish/subscribe messaging is one of the most common message-oriented architectures, where subscribers specify their interest in messages of certain type or topic, and receive messages asynchronously once a publisher publishes a message on the registered interest [72]. Usually the only property a publisher needs in order to communicate with a subscriber is the name and definition of the data. The publisher does not need any information about the subscribers, and vice versa [53]. Under message-oriented paradigm, MQTT is outstanding for its simplicity and efficiency, which costs only a small footprint and low power consumption on embedded devices, meanwhile guarantees the reliability and flexibility for message distribution.

MQTT is a messaging protocol that was introduced by Andy Stanford-Clark of IBM and Arlen Nipper of Arcom (now Eurotech) in 1999 and was standardized in 2013 at OASIS [7]. It aims at connecting embedded devices and networks with applications and middleware. The connection operation uses a routing mechanism (one-to-one, one-to-many, many-to-many) and enables MQTT as an optimal connection protocol for the IoT and M2M [33].

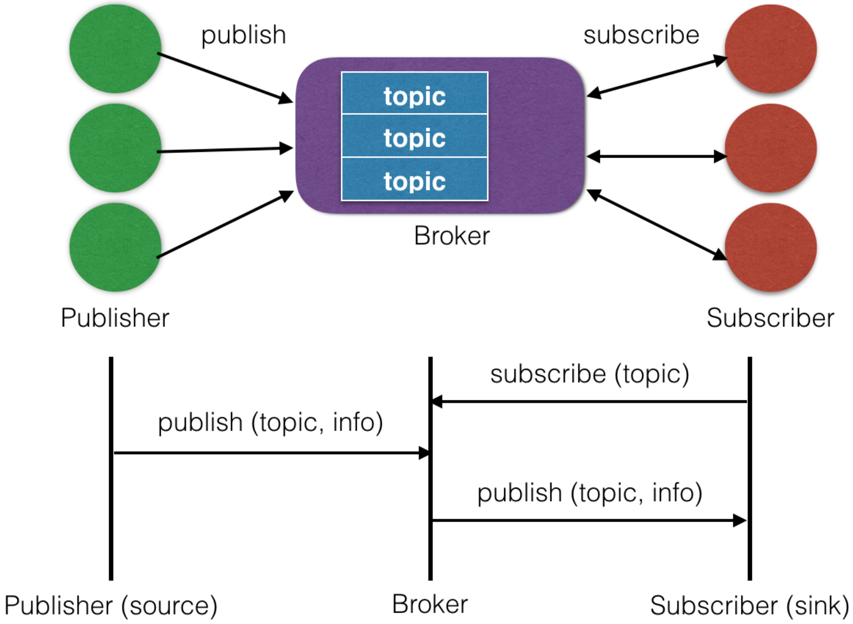


Figure 3.2: The architecture of MQTT and its publish/subscribe process

MQTT is based on topic based publish-subscribe architecture with a message-broker to bridge between the publishers and subscribers, as shown in figure 3.2. It exploits the decoupling in space, time and flow as the core of its working philosophy. An interested device would register as a subscriber for specific topics in order for it to be informed by the broker when publishers publish topics of interest. The publisher acts as a generator of interesting data. After that, the publisher transmits the information to the interested entities (subscribers) through the broker. Furthermore, the broker achieves security by checking authorization of the publishers and the subscribers [41][6].

MQTT supports 3 QoS (Quality of Service) levels for message delivery with enhanced reliability and has a low header overhead [7]. MQTT is a connection oriented protocol as the publisher or the subscriber both needs persistent connection to the message broker. It relies on the underlying TCP layer for connection features [6]. This in turn can lead to challenges with respect to communication costs. Consequently a UDP based MQTT for sensors (MQTT-S) [41] was developed.

3.2.2 Data-oriented: Data Distribution Service (DDS)

Tijero [69] and Pardo-Castellote [53] pointed out that Data Distribution Service for Real Time Systems (DDS) applies a data-oriented/data-centric paradigm. Data-centric communication provides the ability to specify various parameters like the rate of publication, rate of subscription, how long the data is valid, and many others. These Quality of Service (QoS) parameters allow system designers to construct a distributed application based on the requirements for, and availability of, each specific piece of data.

DDS is a publish-subscribe protocol for real-time M2M communications that has been developed by Object Management Group (OMG) [26]. In contrast to other publish-subscribe application protocols like MQTT, DDS relies on a broker-less architecture and uses multicasting to bring excellent Quality of Service (QoS) and high reliability to its applications [33]. Its broker-less architecture fits well with the real-time requirements for IoT and M2M communications. DDS supports 23 QoS policies by which a variety of communication criteria like security, urgency, priority, durability, reliability, etc. can be addressed by the developer [33]. In short, DDS has the following advantages [53]:

- Based on a simple “publish-subscribe” communication paradigm
- Flexible and adaptable architecture that supports “auto-discovery” of new or stale endpoint applications
- Low overhead – can be used with high-performance systems
- Deterministic data delivery
- Dynamically scalable
- Efficient use of transport bandwidth
- Supports one-to-one, one-to-many, many-to-one, and many-to-many communications
- Large number of configuration parameters that give developers complete control of each message in the system

It is important to note that OMG’s DDS specification is silent on the wire-protocol. This in turn has led to the problem that DDS products from different vendors face interoperability issues. To overcome these interoperability issues OMG promotes the use of Real-Time Publish-Subscribe protocol (RTPS) [68] as a wire-protocol for DDS. RTPS itself relies on the use of UDP and multicast to deliver the data from publishers to subscribers. Depending on the network-topology and routers used, RTPS can deliver impressive throughput.

An experimental evaluation of two implementations of DDS [30] points out that this protocol scales well when the number of nodes is increased. While DDS inspired/compatible protocols have been proposed for wireless and sensor scenarios, there seems to be a lack of successful deployments [61], partly due to its complexity.

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 much more. 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 already undertaken much standardization work to make it happen. The IPv6 over Low-Power Wireless Area Networks (6LoWPAN) standards (RFCs 4944 and 6282) now enable IPv6 even on very constrained networks, which allows for seamless integration of sensor and actuator nodes into the Internet [9].

For full convergence, however, devices and services must also interoperate at the application layer [46]. When it comes to mashing up of different services, the World Wide Web has proven its scalability and flexibility. Many applications today depend on the Web architecture, using HTTP to access information and perform updates. HTTP over TCP has problems in constrained environment, though, not only because its overhead in implementation code space, but also low network resource usage due to the fact that high packet error rates and lossy links are common in constrained networks. To come over such a gap, the IETF designed a new Web protocol from scratch, the Constrained Application Protocol (CoAP).

CoAP follows the Representational State Transfer (REST) [32] like HTTP, but is tailored to the requirements of constrained devices and networks. A compact binary format is employed and the protocol runs over UDP (or Datagram Transport Layer Security (DTLS) when security is enabled). A messaging sub-layer provides duplicate detection and optional reliable delivery of messages, based on a simple stop-and-wait mechanism for retransmissions. On top of it, the request/response sub-layer enables RESTful interaction with the web in a similar fashion as HTTP [46]. That being said, CoAP is primarily based on patterns from the Web: a client/server interaction model between application endpoints, resources that are addressable by Uniform Resource Identifiers (URIs), stateless exchange of representations that decouple client and server, uniform interfaces with standardized Internet Media Types for wide interoperability, and caching and proxying to enable high scalability [44].

CoAP is not a mere compression of HTTP, but goes beyond that. For instance, it supports group communication through the underlying UDP; resources are *observable* [38], that is, extra responses continuously push state changes to all registered clients; it includes a machine-to-machine discovery mechanism to find matching resources based on Web Linking [59]; application-layer fragmentation allows blockwise on-the-fly processing of messages that would otherwise exceed the maximum transmission unit (MTU) [11]; alternative transports such as Short Message Service (SMS) or Unstructured Supplementary Service Data (USSD) are also possible and are under standardization [62].

Since both CoAP and HTTP all follow the REST style, CoAP can easily be mapped to HTTP and be connected via transparent proxies and integrated into the same system. A typical CoAP application architecture is shown in figure 3.3. The core of the protocol is specified in RFC 7252 [60], important extensions are in various stages of the standardization process.

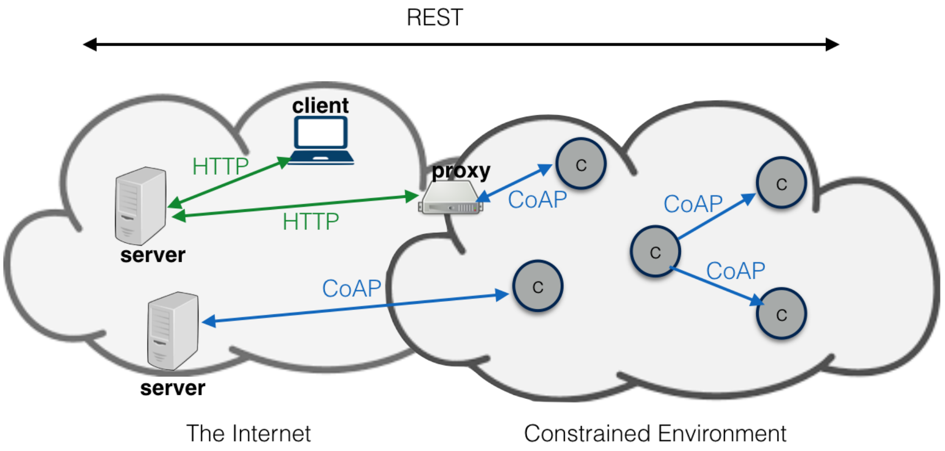


Figure 3.3: A typical CoAP application architecture

3.2.4 Comparison

Speaking of CoAP and MQTT, CoAP seems to have a wider acceptance for constrained devices due to the following facts: ease of integration with 6LowPAN, easy portability with HTTP, UDP based operation with low connection overhead, support for several features like sleeping nodes [60](without requiring persistent connection), support for both request-response and resource-observe mode. On the other hand, MQTT is ideal for integration with existing cloud service since TCP is already employed in most infrastructures.

MQTT is a message-oriented protocol which means that it is content agnostic and only focusses on the delivery of messages, while CoAP is all about operations against the representation of resource since it follows the REST style. Resource observing in CoAP is similar to publish/subscribe feature 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. As a result of forementioned difference, MQTT is simple to implement on the sensor/device side since all relevant part is about message delivery. Reference [72] pointed out several disadvantages of resource-oriented IoT protocol 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 in currently most common IPv4 subnet environment.

However, CoAP has a clear semantics of operations against a resource and different CoAP method can be used to differ read and write. While 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 differs a lot in terms of finding services. With MQTT, it is necessary to first find the central server, a.k.a., the broker and messages always go through the broker. There is no need for a central endpoint for CoAP since it supports multicast to discover available services. And after that, the read/write semantics of CoAP ensures independent communication endpoint to endpoint. That is to say, it requires less effort to make CoAP scale up than MQTT. Matthias Kovatsch [44] stated similar argument that the main drawback of MQTT is missing extensibility, as MQTT clients must be pre-configured with a dedicated service like HTTP clients for IoT, which makes it hard to adapt to an evolving environment.

There are few research publications as well as industry deployments about DDS, which implies it may be still under exploration and optimization stage. On the other hand, there are known scalability [31] and performance [57] 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 [61].

	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	Simple stop-and-wait retransmission reliability with exponential back-off
HTTP	TCP	✓	✗	✓	-	-	SSL	-

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

	Typical application
MQTT	Topic based real-time messaging kind of application using pub/sub requiring persistent connection with the server. Message broker is responsible to weave the sensors with the rest of the Web.
DDS	Applications and system-of-systems using pub/sub that have to be able to support dynamically changing environments and configurations, be constantly available, and be instantly responsive. Integrating data across many platforms and disparate systems is also necessary.
CoAP	Applications with sensors running RESTful web-services to achieve direct connectivity to the Web with HTTP like methods and URI. Applications integration with HTTP based Web.

Table 3.2: Potential usage of MQTT, DDS and CoAP

Table 3.1 provides a brief comparison between MQTT, DDS, CoAP and HTTP in terms of architecture. HTTP is also included here as a contrast to CoAP since they share many similarities. Table 3.2 provides 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 [9] and so is the IoT. It is easy for such a pattern to interwork with existing Web. 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 about read/write and representation state (controlled by Internet Media Types).

A particularly interesting aspect of the resource-oriented communication is the natural emergence of Fog Computing. Since the resources need to process the requests, they must have some basic processing capabilities. This in turn allow us to distribute the processing load and to reduce the load on backend-services. In addition, since the requests have a clear read/write semantic, it becomes possible to add infrastructure support in form of caching and reverse proxies thus allowing for more distribution of load and network traffic [61].

There is no IoT application protocol that fits all situations. As A. Al-Fuqaha et al. [33] 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. Speaking of CoAP, it makes it easy for applications that are more likely deployed in unconstrained environment to talk to constrained devices. It also provides observe functionality when publish/subscribe pattern is more a desired goal. More importantly, CoAP has its position in both Fog and Cloud and consequently has been selected as the protocol used in this thesis.

3.3 CoAP Fundamentals and Implementations

This section gives an inspection of some key features and concerns of CoAP, which will define the important terminology for this research and give insight of ideas and trade-off made in the proposed architecture that is discussed with more details in later 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 defined in 3.3.1; basic semantics of the core protocol is introduced in 3.3.2; an important extension of the core protocol - observe is discussed in 3.3.3; security issue is covered in 3.3.4; a more detailed comparison with HTTP is stated in 3.3.5 which also acts as a brief summary of above sections; and a summary of current implementations is in 3.3.6.

3.3.1 Constrained RESTful Environments

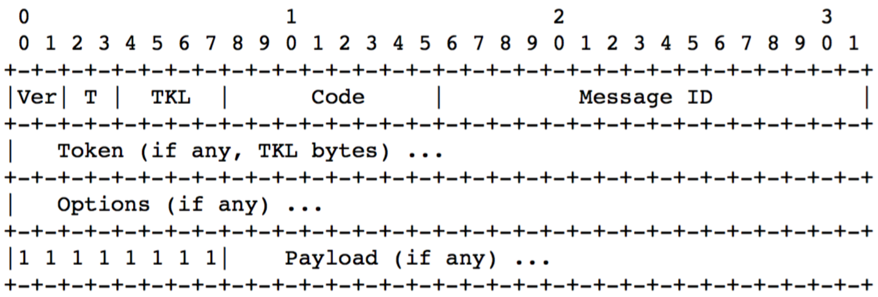
In order to provide a framework for applications that embrace constrained IP networks as found in the IoT, the working group for Constrained RESTful Environments (CoRE) considers it is necessary to classify resource-constrained devices according to their capabilities. RFC 7228 on terminology defines the following 3 classes [10][44]:

- **Class 0** devices are very constrained sensor-like motes. They are so severely constrained in memory and processing capabilities that most likely they will not have the resources required to communicate directly with the Internet in a secure manner. To participate in Internet communications, Class 0 devices require the help of larger devices acting as proxies, gateways, or servers. Obvious examples are proprietary temperature sensors that send their readings wirelessly to an indoor weather station or bedside alarm clock. Their memory sizes are usually in the order of hundreds of bytes only.
- **Class 1** devices are most resource-constrained devices that can directly connect to the Internet with on-board security mechanisms, which requires about 100 KB of ROM and about 10KB of RAM. They can not employ a full protocol stack such as using HTTP, Transport Layer Security (TLS), and related security protocols and XML-based data representations, due to limited memory size and processing power. Hence they require lightweight protocols that have low memory footprints and parsing complexity, such as CoAP.
- **Class 2** devices are less constrained and almost show the characteristics of full-fledged Internet nodes like smartphones or notebooks. This becomes possible at about 250 KB of ROM and about 50 KB of RAM. Yet, even these devices can benefit from lightweight and energy-efficient protocols and from consuming less bandwidth, in order to free resources for the application or reduce operational costs. Constrained devices with capabilities more than Class 2 exist but they can still be constrained by a limited energy supply.

According to above definitions, sensors and actuators are more likely to be class 0 devices. And some System on Chip (SoC) falls into class 1 and may operate on behalf of the class 0 devices and expose their resources to outside world. Other embedded platform such as mobile phones and the Raspberry Pi [56] 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 hierachical structure also naturally fits into the definition of a Fog. In the thesis, the Raspberry Pi is used as the evaluation platform for constrained environment since it is more likely to deal with lots of concurrent clients/servers and has the potential to do some simple preprocessing other than just protocol handling.

3.3.2 Core Protocol

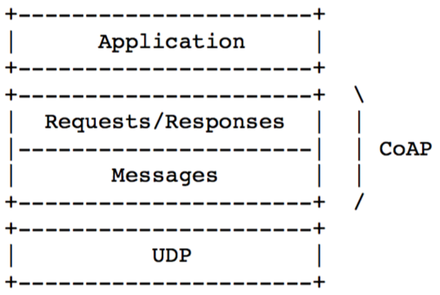
CoAP is designed to use minimal resources, both on the device and on the network. Instead of a complex transport stack, it gets by with UDP on IP. A 4-byte fixed header and a compact encoding of options enables small messages that cause no or little fragmentation on the link layer. Figure 3.4 [60] shows the four-byte base header of CoAP, which can be followed by the variable-length token, multiple header options, and a payload carrying the representations mentioned above.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.4: Message format

An entity participating in the CoAP protocol is called an endpoint [60]. 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 [60].

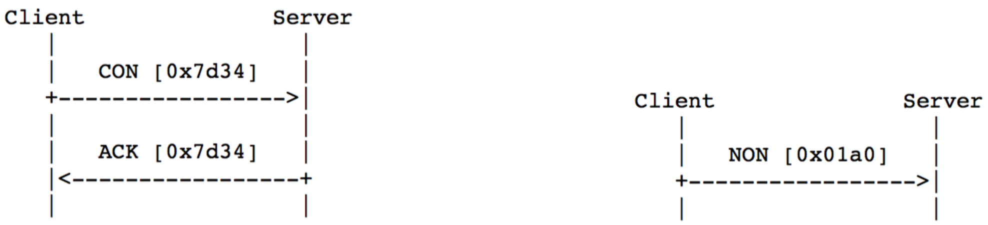


```
+----------------------+
|      Application     |
+----------------------+
+----------------------+  \
|  Requests/Responses  |  |
|----------------------|  | CoAP
|       Messages       |  |
+----------------------+  /
+----------------------+
|          UDP         |
+----------------------+
```

Figure 3.5: Abstract layering of CoAP

Messages can either be confirmable (CON), non-confirmable (NON), acknowledgements (ACK) or resets (RST). Confirmable and non-confirmable messages carry requests or responses. When an endpoint receives a confirmable message, it replies with an acknowledgement. The response to a confirmable request can be sent with the ACK, which is also called piggybacked response, or in a separate confirmable response. An endpoint retransmits confirmable messages with an exponentially increasing back-off timer until it receives an acknowledgement, a reset or the maximum retransmission count is reached (which is typically 4). If an endpoint receives a CON or NON that it does not know how to process, it rejects it with a RST. A message is identified by a message ID (MID) and an endpoint needs to temporarily remember incoming MIDs to detect duplicates. For a confirmable request, one endpoint only considers successfully receiving an acknowledgement when receiving ACK with the same MID as in the request, which is 0x7d34 in the example in figure 3.6a [60]. For a non-confirmable request, no ACK is required and MID is only used to detect duplicates, as shown in figure 3.6b [60].

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 token, chosen by the client, serves as identifier for a request. The server endpoint must include the request

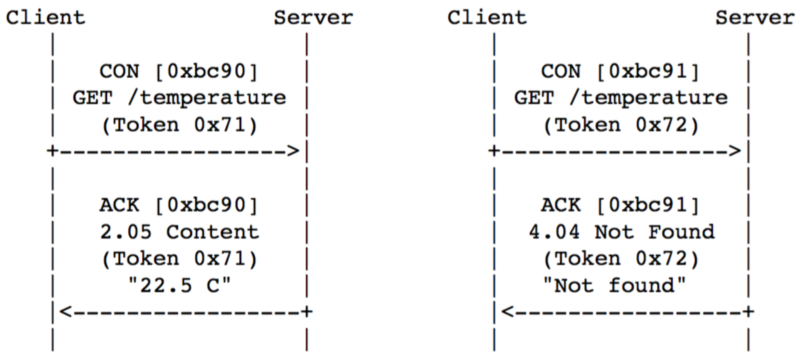


```
Client              Server
   |                   |
   |   CON [0x7d34]    |
   +------------------>|
   |                   |
   |   ACK [0x7d34]    |
   |<------------------+
   |                   |
```

```
Client              Server
   |                   |
   |   NON [0x01a0]    |
   +------------------>|
   |                   |
```

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

Figure 3.6: Reliable and unreliable message transmission



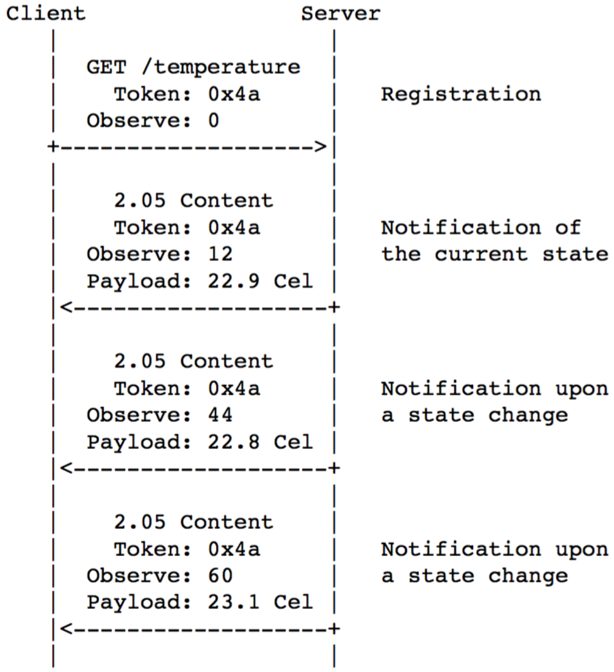
```
Client              Server       Client              Server
   |                   |            |                   |
   |   CON [0xbc90]    |            |   CON [0xbc91]    |
   | GET /temperature  |            | GET /temperature  |
   |   (Token 0x71)    |            |   (Token 0x72)    |
   +------------------>|            +------------------>|
   |                   |            |                   |
   |   ACK [0xbc90]    |            |   ACK [0xbc91]    |
   |   2.05 Content    |            |  4.04 Not Found   |
   |   (Token 0x71)    |            |   (Token 0x72)    |
   |     "22.5 C"      |            |    "Not found"    |
   |<------------------+            |<------------------+
   |                   |            |                   |
```

Figure 3.7: Two GET requests with piggybacked responses

token in the response so that the client endpoint knows to which request the response belongs to. Additionally, CoAP requests and responses can be accompanied by simple options, similar to HTTP header options. For example, options may describe the content format or destination URI. Two examples for a basic GET request with piggybacked response are shown in Figure 3.7 [60], one successful, one resulting in a 4.04 (Not Found) response.

3.3.3 Observing Resources

A key feature for the IoT is observing resources. The observe extension enables efficient server push notifications based on the observer pattern [44]. It is designed as an optional feature on top of GET with an elective Observe option that is set to zero by the client. If a server does not support it, this is as answering a normal GET request and clients can repeat requesting to polling. When the server supports this feature, it will respond with this option, which turns the response into a notification. The server promises to keep the interested client on its list of observers as long as possible and will push new representations whenever the observed resource changes. This extends the request-response pattern to a request/multiple-response pattern, where all notifications are correlated as usual through the token. CoAP notifications also make use of cache control, that is, they have a valid lifetime defined by the Max-Age option and are cacheable. Usually the server sends a new representation before Max-Age expires. When a representation becomes stale, the client assumes that it was dropped by the server (e.g., because of a reboot) and can re-register by sending another observe request using the same token. Also the options must be identical to the original observe registration, so that the request matches the cache key in case intermediaries are involved.



```
Client               Server
   |                    |
   |  GET /temperature  |
   |    Token: 0x4a     |   Registration
   |  Observe: 0        |
   +------------------->|
   |                    |
   |    2.05 Content    |
   |    Token: 0x4a     |   Notification of
   |  Observe: 12       |   the current state
   |  Payload: 22.9 Cel |
   |<-------------------+
   |                    |
   |    2.05 Content    |
   |    Token: 0x4a     |   Notification upon
   |  Observe: 44       |   a state change
   |  Payload: 22.8 Cel |
   |<-------------------+
   |                    |
   |    2.05 Content    |
   |    Token: 0x4a     |   Notification upon
   |  Observe: 60       |   a state change
   |  Payload: 23.1 Cel |
   |<-------------------+
   |                    |
```

Figure 3.8: CoAP observe synchronizes the local state with the resource state at the origin server by sending push notifications. [38]

In case the client is no longer interested, there are two possibilities to end an observe relationship [44]:

- **Re-active cancellation:** Observers can simply remove the local relationship states, which leads to a "garbage collection" at the server: When the next notification arrives, the client cannot match the token and will reject it. Since every once in a while the server must use a CON notification to detect orphans, it will eventually receive a RST that tells the server to remove the client from its list of observers. The same happens when a CON transmission times out, usually caused by a client that shut down (orphan).

- **Pro-active cancellation:** Some applications require a more timely cancellation to save resources. In this case, clients can send a cancellation request by setting the Observe option to one and using the token associated with the relationship.

3.3.4 Security

The security model of CoAP is similar to traditional Web services: Transport Layer Security (TLS). Due to the resource constraints and UDP binding, CoAP depends on Datagram Transport Layer Security (DTLS) for security (see section 9 of [60]). It provides the same flexibility with a variety of cipher suites, which define the set of cryptographic algorithms used. 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. The integration of DTLS into the Java CoAP implementation Californium [12] is discussed in a master's thesis from 2012 [43]. Since DTLS adds extra complexity and may degrade performance of a CoAP server, various optimization should be taken into account and thus it is not a focus of this work.

3.3.5 CoAP vs. HTTP

It is important to note that CoAP is more than just a compressed form of HTTP and moreover provides several features that are beneficial in an M2M application. Reference [48] gives a comprehensive comparison between CoAP and HTTP in terms of protocol concept as well as the processes for handling requests on server side. It shows how CoAP + UDP are superior to HTTP + TCP in constrained environments. In short, the similarities and differences are,

- **Fewer messages**: A typical CoAP exchange consists of 2 messages, i.e., a request and a response. In contrast, an HTTP request first requires the client to establish a TCP connection and later terminate it. CoAP's blockwise transfer[11] though, requires an acknowledgement for each block and leads to more messages and higher transfer time. But since most CoAP messages are short, this is not a big concern.

- **Compressed format**: CoAP encodes option values in binary format while an HTTP request is one large, verbose text. A minimum CoAP header is only 4 bytes long and a minimum UDP header is only 8 bytes long. In contrast, a minimum TCP header alone is 20 bytes long plus what comes from HTTP. A bare CoAP request is not human-readable though.

- **Observe pattern**: Observe pattern is basically such a process: a client declares its interest in the occurrence of a specific type of event to a server and is notified by the server when such an event occurs. In CoAP, a client can establish such an observe relation with a resource which sends a notification when its state changes. This is not available in HTTP.

- **Resource discovery**: CoAP defines a well-known URI */.well-known/core* which lists the URIs to available resources on a CoAP endpoint. URIs and descriptions of resources are encoded in the Core Link Format [59] and can be requested by a GET request from a client. This mechanism allows autonomous devices and services to efficiently discover other CoAP resources in a uniform and standardized way. While crawling achieves similar goal with HTTP, but is highly inefficient in terms of data exchange.

- **Group communication**: CoAP supports making requests to an IP multicast group [60]. It allows a client to address multiple servers at once. This can obviously save some effort for the client and can especially be useful for discovery. IP multicast violates TCP's connection oriented paradigm, and is therefore not applicable for HTTP.

- **Deduplication**: A disadvantage of CoAP is that it has to detect and filter duplicates on its own, unlike HTTP, which inherits the reliability guarantees from TCP. A CoAP server identifies a message by the pair of its source and message identifier (MID) and has to remember it for a specific time (247 seconds for confirmable messages and 145 seconds for non-confirmable messages). This may add extra overhead in terms of memory consumption and book-keeping effort.

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

Table 3.3: Structures of the processes for handling CoAP and HTTP requests

And in terms of request processing, table 3.3 [48] lists different steps that would happen when handling a request of CoAP/HTTP. Note that a traditional HTTP server is used for loading files from local disks and serving them online for clients, if the files are not dynamically generated web pages, while a CoAP server usually holds a data structure (maybe in memory) of the resources or generates a response using other resource on the fly.

3.3.6 CoAP Implementations

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

CoAPBlip for TinyOS [54], SMCP [64], libcoap [47], Erbium (Er) for Contiki [45] and CoAPSharp [21] for the .NET Micro Framework are optimized for embedded devices [48]. Cantcoap [16] is a CoAP implementation that focuses on simplicity by offering a minimal set of functions and 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 [52] 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 [19] to implement backend services. It primarily targets easy interaction with OpenWSN devices and is not designed for scalability.

CoAPthon [67][22] is another Python library for CoAP built on top of the Twisted framework [70]. 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 [44]. At the time of writing, it has become part of the ARM mbed platform [3] since the Sensinode start-up has been acquired by ARM at 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 targeting stronger embedded devices such as Raspberry Pi and middle-class smartphones [18][44]. Its design goals include interoperability, development simplicity, and code reusability rather than scalability.

There are two open-source Java frameworks jCoAP [42] and nCoap [51] which target unconstrained platforms. The former one only implements an early draft version of CoAP at the time of writing thus is incompatible with current CoAP standard. While the latter one, according to [48] and [44], may still be in early optimization stage and have problems that affect its performance.

Californium (Cf) [12] developed in ETH, Switzerland is a modular, open-source framework that facilitates deployment of backend services, serving as intermediary between the logic of a service and the IoT [48] [46][44]. With its core based on Java, it aims at providing high scalability as backend for CoAP in the cloud. 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.

Henning [50] 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 [27]. 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 [25] 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 makes it a useful testing tool for application as well as protocol development [43]. Copper is meant to run only as a client.

A brief summary and comparison between major CoAP implementations is shown in table 3.4. Some details about certain implementations are temporarily not available at the time of writing. The column 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 [24] and on the website coap.technology [20].

Table 3.4 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.

	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 3.4: 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.

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 [44]. On the other hand, there is a silence on fault-tolerance feature. Among these popular implementations, the lack of 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 of 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 [35], 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 design of gen_coap has to be reconsidered in the context of this thesis. For instance, the relationship between different components, 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 language such as Go [36] exist, namely go-coap [23] and canopus [15]. However, both projects are incomplete.

Different from implementations listed in table 3.4, this work aims at evaluating to what extent an implementation could both scale up (in the cloud) and scale down (under relatively constrained environment). Meanwhile fault-tolerance is checked through comparing performance under ordinary operation and when one or more faults are injected on purpose. As the state-of-the-art implementation which was designed with scalability in mind and has verified high performance over other implementations (as stated in [46][44]), Californium is selected as the performance reference during the evaluation of this work.

Few benchmark tools for CoAP server performance testing have been presented, among which, CoAPBench from Californium (Cf) Tools [13] 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 chapter 5.

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 hardwares 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 [44], which is more a requirement for 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 [48] and M. Kovatsch [44] 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 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 [65]. Nevertheless, using threads directly as a programming model adds complexity in both data and control plane [65][49]. 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 are 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 [49]. Threads are also not a practical abstraction in distributed computing as the effort to make a shared memory illusion is expensive [49]. 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 [65]. However, event-driven inverts the control flow and effectively turns a program into reactive style, which makes it more complicated to block and harder to understand when the problem space increases [71]. 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 [71].

Concurrency in software is difficult and using threads as a concurrency abstraction makes it worse [49]. 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 can not 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 [55]. Moreover, since today's modern hardwares 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 hardwares [65].

With all above being said, as Edward Lee [49] pointed out, 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 [65]. 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 forementioned point could lead to inefficiency, which, however is a trade-off for fault-tolerance.

Erlang has strong fault-tolerance support since its background was rooted in 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 good way to handle them. It is proved that the main sources of downtime in large-scale software systems are intermittent or transient bugs [14]. 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 [39]. 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 [39]. And the ability of an Erlang process to receive signal when other process of interest terminates enables a supervision tree structure in an application, where the leaf nodes being workers who execute actual tasks and higher 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 recorded accordingly) 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 strategy more or less [37]. However, few environment provide 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 [39]. 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 failure 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 code, 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 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 in Web service 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 share similarities with these areas. And with new paradigms such as Fog Computing emerging, IoT intends to be made up of more distributed computing force where latency is sensitive and scales from embedded platforms to cloud backends. Erlang should fit well in such circumstances.

Erlang itself is no silver bullet. 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 [2] 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 ; Kilim [66][65] is a Java actor-oriented server framework which does the magic by transforming the Java bytecode; Project Loom [55] 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 [65]. This resembles actor-oriented Erlang system anyway. It is the maturity of the language and runtime that renders Erlang as the primary environment in this thesis.

4.1 Concurrency Model

The popular Java CoAP server/client framework Californium was inspired by previous work for highly concurrent Internet services, in particular SEDA and the PIPELINED architecture [48][44]. 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 straight forward architecture like 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 straight forward to come up with the architecture shown in figure 4.1. Conceptually, the *ecoap* prototype can be split up to socket manager, endpoint and handler, which are reusable for both server and client. When being used as server, a registry maintaining routing rules is also included. The socket manager hides network details and would only dispatching 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 remote CoAP

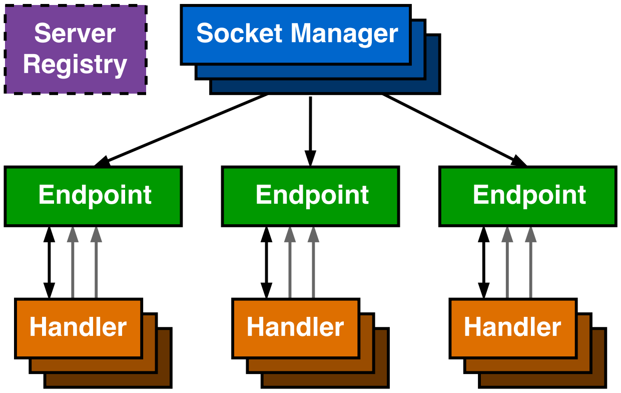


Figure 4.1: The logical architecture of *ecoap*. One or more socket manager(s) exist depending on the underlying transport protocol. One endpoint maps to exact one remote CoAP endpoint. Each endpoint may have many handlers associated with it. Handler contains logic for server or client. A server registry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

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 though 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 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 in 4.1.1, 4.1.2, 4.1.3 and 4.1.4 while different supervision strategies and the structure of the whole application are introduced in 4.1.5. Then 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 scope of this thesis, the component can be easily replaced by wrapping a socket that listens on DTLS port or over other 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 bottleneck, the process should do as little work as possible. When the socket process receives a datagram from a new remote endpoint as 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 hands 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 process dictionary destroys referencial transparency and makes debugging difficult []. 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.

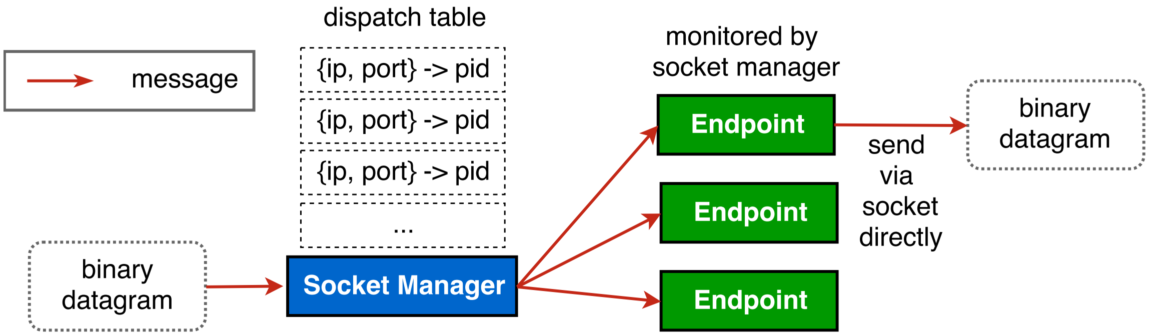


Figure 4.2: The socket manager receives datagram over network and dispatches them based on its endpoint dispatch table. The endpoint process sends datagrams directly using the socket reference passed by the socket manager. Every endpoint process is monitored by the socket manager so that the latter would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

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 encapsulate 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 work load of the socket manager process and makes the dispatching between socket manager and endpoint processes 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 underlying operating system and does not behave uniformly. 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.

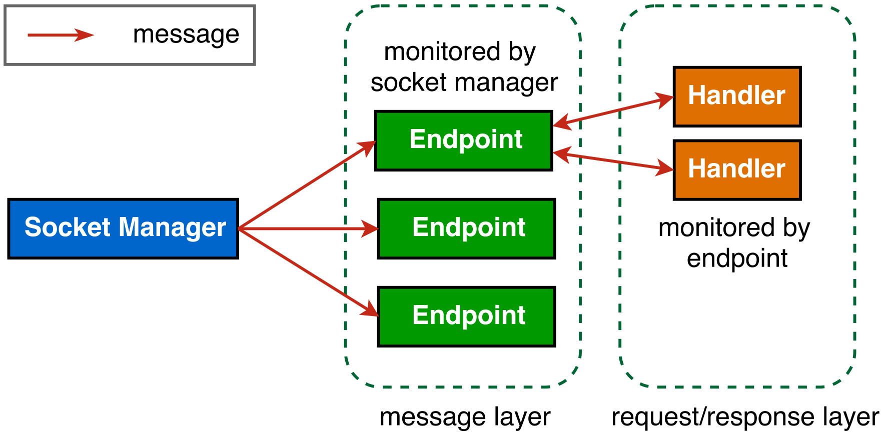


Figure 4.3: Relation between endpoint and handler

As shown in figure 4.3 and 4.4, the endpoint process, `ecoap_endpoint`, 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 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

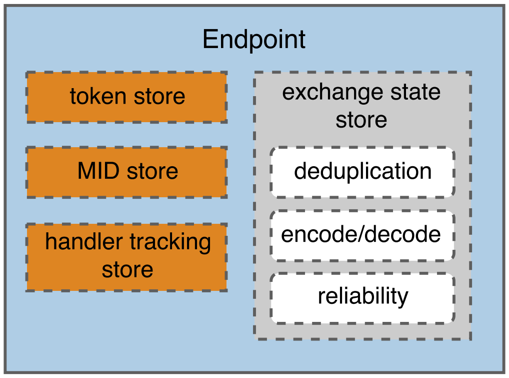


Figure 4.4: The function of an endpoint. The exchange state is the description of both the current state and the action that should be taken upon the state next. Therefore, deduplication, serialization and reliable transmission are all defined as part of the actions and are executed accordingly.

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, the `ecoap_endpoint` process 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 corresponding acknowledgement”, together with the function name that triggers transition to next state. The very `ecoap_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 time out. Detailed state transition flowchart can be found in 4.2.3.

The work flow 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 appropriate handler process for further processing. After the handler process finishes its task, it passes the result back to the `ecoap_endpoint` process which then triggers a state transition where actions including encoding the result and sending it over the network are executed.

The `ecoap_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). A counter of currently running handler processes is maintained by the endpoint process. This has an important impact on the lifespan of an endpoint process, which will be discussed later in 4.1.6.

The `ecoap_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 `ecoap_endpoint` processes. The share-nothing of Erlang renders MID management straight forward 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. On the other hand, the message token is generated as a strong random bytes using Erlang's crypto library with configurable length. Tokens are also only produced by `ecoap_endpoint` process independently. One could argue that randomly generating token does not fit in a variety scenarios. However, using crypto strong random bytes as token under one endpoint's scope greatly simplifies the effort to synchronize and avoids collision to a relatively high level, which is enough for the *ecoap* prototype.

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 acts 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.

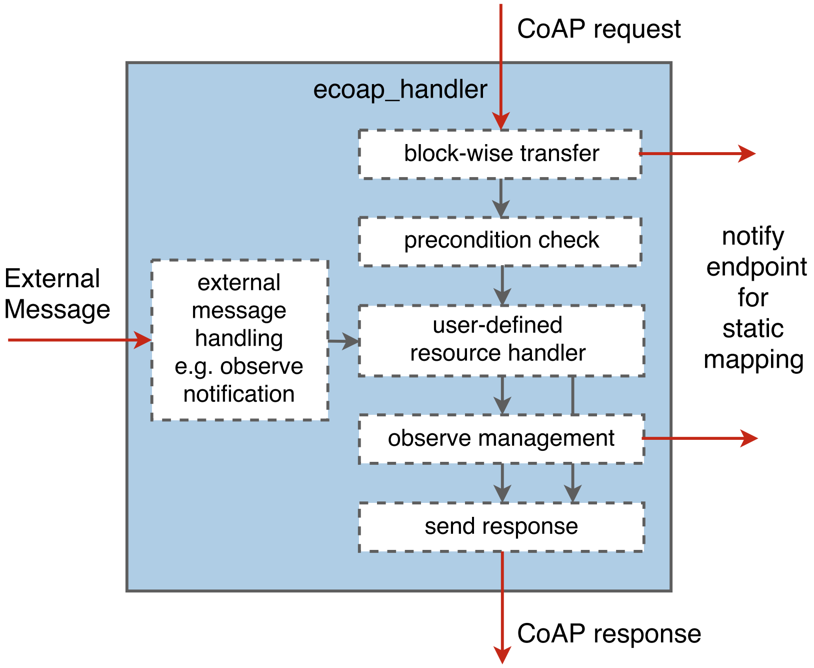


Figure 4.5: Workflow of an ecoap_handler process

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 corresponds 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 corresponding `ecoap_endpoint` process which sends it to the client.

The `ecoap_handler` process is spawned by the `ecoap_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 immediately after sending the response. No bookkeeping work other than monitoring is taken by the corresponding `ecoap_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 task 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 sets 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 time out 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 `ecoap_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 a `ecoap_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 [11]. 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 inconsistence 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 process exits. 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 [29], 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 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 the it to perform 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 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 [38], 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. Concurrent block-wise transfer is an undefined behaviour [11]. 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 a desired result in many cases.

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 records are stored in Erlang maps (which are essentially hash tables) as process inner state, providing fast reads and writes.

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.

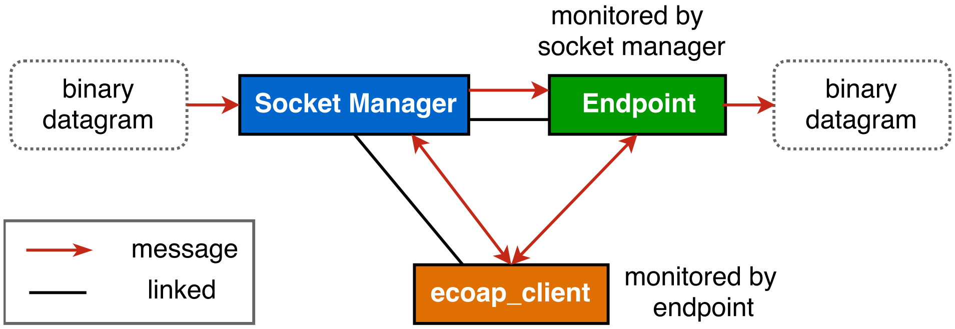


Figure 4.6: Structure of a standalone client. The ecoap_client process only uses message passing to ask the socket manager to start corresponding endpoint process.

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 (with configurable port) and links to it. For the socket process, when the ecoap_client process issues a request, it directly starts an ecoap_endpoint process and links to it as well, instead of using any other supervision tree structure, as shown in figure 4.6. Therefore a standalone client usually consists of one ecoap_client process, one socket manager and one or many ecoap_endpoint process(es). 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.

While an embedded client uses 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), an IoT device management protocol built

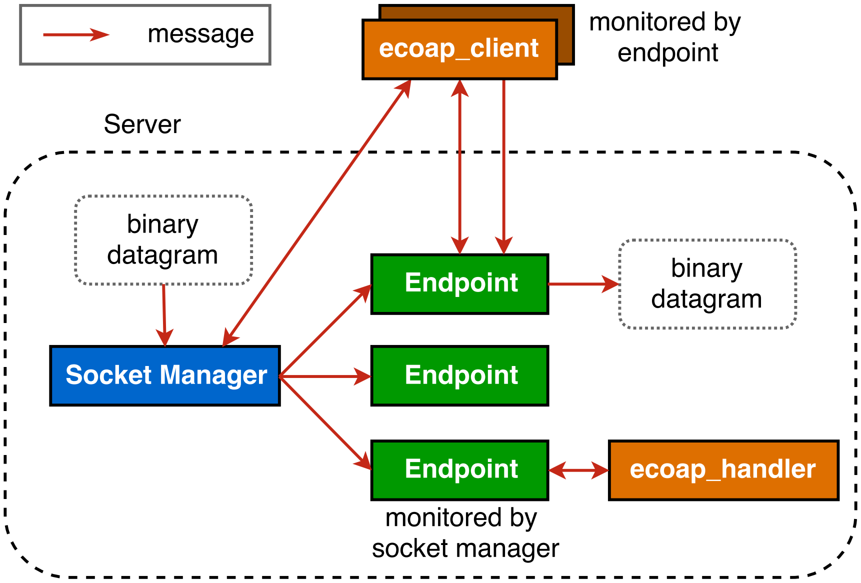


Figure 4.7: Structure of an embedded client.

ontop 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 `ecoap_endpoint` process will be connected to the supervision tree of the server, and the very `ecoap_endpoint` process will be used for both client role and server role which avoids duplication and confusion of states within the application. Last but not least, depending on the requirement of the application, the `ecoap_client` could be started either during the initialization of the server or inside a resource handler function.

Besides the state and lifespan management that will be discussed in 4.1.6, the endpoint process monitors the `ecoap_client` process for another 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 other client processes or by the server concurrently. In the standalone mode, termination of an `ecoap_client` process will shutdown 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.

The `ecoap_handler` and `ecoap_client` process both work as the handlers that provide 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 keys are the URI of resources as a list of path strings and the values are 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 hold 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. 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 time instead of writing, which renders ETS a fast and reasonable solution.

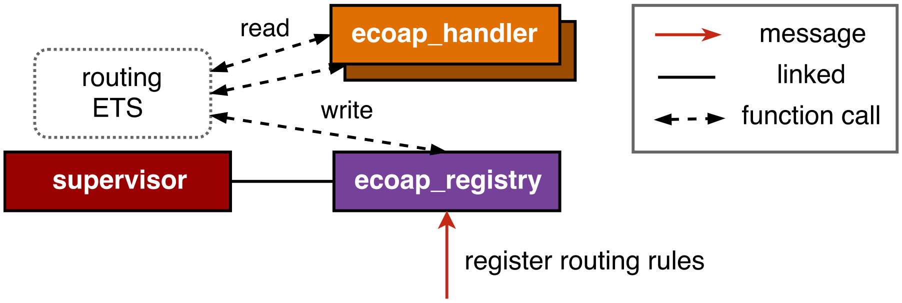


Figure 4.8: Supervision tree of the server registry. The routing ETS table is owned by the supervisor. The ecoap_registry process serialize all writes to the table including insert and delete. Many ecoap_handler processes can read from the table concurrently. Reads and writes are direct function call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

A ETS table can not be linked to or monitored as 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 deafult, 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 the only 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 only manages the `ecoap_registry` process and does not take part in any other work, 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 serve 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". If 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. It is exported as a function executed by `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 setting in Erlang ensures 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 [39]. This approach decreases the risk of core processes dying until the system can not cope with the errors properly anymore.

With the forementioned 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.

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 wether 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 above observation, a `simple_one_for_one`, `temporary` strategy is the most proper one for the `ecoap_handler` process. With this restart strategy, 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

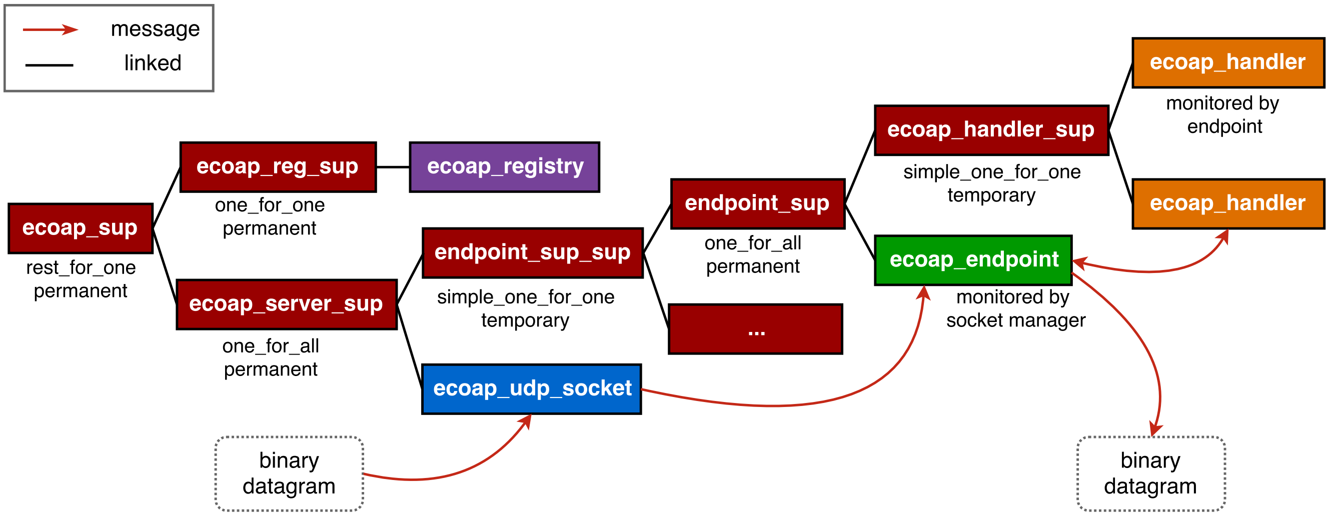


Figure 4.9: Supervision tree of *ecoap* as server

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 modelling a remote CoAP endpoint. 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 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 can not be shutdown easily, a workaround as follows is taken. The top supervisor, `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 time 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 (with related supervisors) 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*. Crash of the socket process implies the socket is closed and any reference to it becomes invalid. In such a case, all endpoint processes can not 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 serialize 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 CoAP Endpoints 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 risk of race condition and is not considered.

As a result of that, the supervisor of `ecoap_udp_socket` and `endpoint_sup_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 sub tree under this layer are 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. 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.

On the other hand, within the server registry, the supervisor sets a `one_for_one`, `permanent` strategy since the `ecoap_registry` process is its only child and should always be restarted after a crash.

As shown in figure 4.9, the supervision tree completes when the `ecoap_server_sup` process and the `ecoap_reg_sup` process both connect to the root supervisor of the application. The root supervisor is then linked to the application controller of the Erlang VM. Crash of the root supervisor implies failure of the whole application thus no further action is taken other than completely shutdown *ecoap*. It is desired that crash happens to `ecoap_server_sup` should bring down the tree under it but leave `ecoap_reg_sup` alone. While the crash of the `ecoap_reg_sup` process should terminate both of them because the routing table has gone with the `ecoap_reg_sup` process and all subsequent requests will result in a 4.04 (Not Found) response. There is no point to continue service unless one manually initialize the routing table again. For this reason, the start order of `ecoap_server_sup` and `ecoap_reg_sup` is important. If the supervisor starts `ecoap_reg_sup` first and then `ecoap_reg_sup`, and applies a `rest_for_one`, `permanent` strategy to them, it is guaranteed that crash of the later started one will not affect the first started one, but crash of the first started one will bring down both of them.

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 leading too many restarts that exceeds 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 clarifies the dependency of worker processes, making start and shut down of every component in a deterministic order. The worker processes are always restarted in the pre-defined order so that none of them will be left in unknown state.

4.1.6 State Management

Deduplication States

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 [48]. It is required that a CoAP endpoint remembers confirmable messages for a EXCHANGE LIFETIME (247 seconds) and non-confirmable messages for an NON LIFETIME (147 seconds) [60]. 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 have to cache the response for transmission, along with the request information for filtering. The potential 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 map 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 map and checks their age. Any message whose lifetime has expired is removed from the map. It is proved that the above method is more efficient than creating a timer for every message. As timer is also implemented as 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.

It should be noticed that CoAP server can further relax deduplication [60]. NON requests can be designed with semantics that no response needs to be cached, thus only leaving duplicate filtering, which is less critical since their lifetime is shorter. When processing only idempotent requests such as GET, PUT, and DELETE, no responses have to be cached and *ecoap* can be configured with no 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 and no handler process is running. The conditions have to be both met because it is possible that the exchange map is empty while one or more handler processes are still running. For instance, when the client is observing a resource that is seldom updated or waiting for a separate response of a time-consuming task, there is a chance that no new messages arrive during the wait and all exchanges have expired. And the endpoint should not terminate in such a case. The endpoint process therefore monitors all alive handler processes and maintains a counter which records the number of them. It could then directly check whether the exchange map is empty and the counter is set back to zero right after each exchange cleanup. However, it is not enough to examine only the above when no deduplication is enabled or when message lifetime is configured to be strongly reduced. The endpoint process might falsely consider it is to terminate while there is still ongoing traffic on the way because the exchange map is emptied too fast. In order 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 set after receiving a message. As a result, an endpoint process could only safely terminate after a cleanup when the following items are all true,

- The last message exchange has expired.

- The handler process counter is zero.

- No message has arrived between two cleanups.

The three conditions are not valid in all cases, but any violation of them implies the remote endpoint may further communicate with the application and corresponding processes should be ready for that. The socket manager will be notified on termination of the endpoint process and remove its record from the dispatching table. The sub supervision tree where the endpoint process resides is automatically shut down as described in 4.1.5.

Observe States

The basic observe workflow is introduced in 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 [44]. 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 arrive the message queue of the process. There is no chance that a first produced notification gets a higher sequence number than the actual latest one does. This method compliances with RFC 7641 [38], 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 modelled 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 export them through Erlang behaviours. 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 cusomized callback module which exports a set of pre-defined callback functions, much like an interface implementation in a OOP context. The OTP and behaviours reduce the complexity in terms of code, testing, maintenance and understanding. *ecoap* is built all based on OTP behaviours. Besides all supervisor processes, which are obviously implementation 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 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 4.2.4.

The socket manager, endpoint and handler processes make most communications in an asynchronous manner using one way `gen_server` cast or direct message passing in order to avoid unnecessary blocking operations. 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. The client functions 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 client functions though, so that further changes that are internal to the process can be made smoothly.

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 comma, pattern matched respectively and bound to corresponding variables. Fields begin with 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 two bits of the second segment to variable `Type` and `Token:TKL/bytes` means setting variable `Token` to `TKL` bytes start right after the 16 bits message identifier (`TKL` is bound earlier in the match). On the other hand, the `?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. Assume `BinMessage` is a datagram received from 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 1: 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, CoAP messages are further decoded and represented as Erlang record, which is a syntax sugar of tuple where attributes can be directly accessed by name. Listing 2 and listing 3 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 of a record can be done in 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 2: 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.

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

Listing 3: 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].

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. 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 machines, the state transition of the message exchange is implemented using plain Erlang functions. Similar to 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, cached result for deduplication, timer for triggering any time out 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 a map 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 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 time out message caused by a timer, it searches the map and invoke corresponding state transition function as specified by the stage stored in the exchange record. The full state transition flowchart is shown in figure 4.10.

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 has 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. One can pattern match against the prefix and suffix to get dynamic URIs. For example, the handle function can be written to only respond requests that hit exactly the prefix while rejecting any other requests. The `Request` is a CoAP message record underneath and can be used to obtain options and payload via helper functions. The `coap_discover/2` 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 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 an unique reference here if the notification is to be sent as a confirmable message. The reference can then be used to verify the delivery in `coap_ack/2`. If no reference is explicitly specified, a dummy value is used.

In the example server, the GET handle function responds with the text ”HelloWorld”. The PUT handle function invokes `ecoap_handler:notify/2` to send a new notification (the same as the content being uploaded in the example). The DELETE handle function sends a 4.04 (Not found) response to all observing clients.

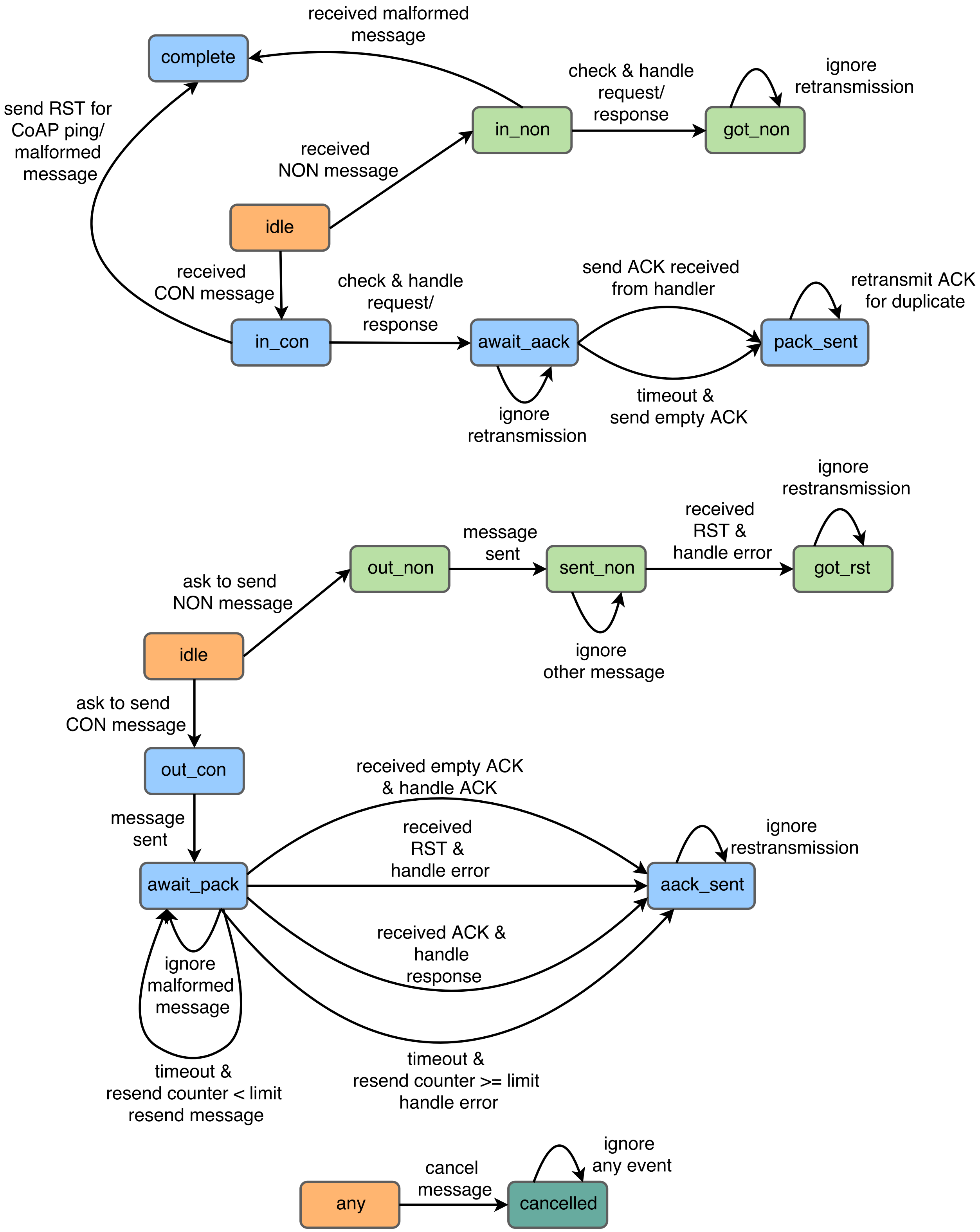


Figure 4.10: Exchange state diagram. All exchange starts from idle state. The complete state means the exchange is immediately removed.

```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() ->
    {ok, _} = application:ensure_all_started(ecoap),
    % maps /HelloWorld/* to the module that implements ecoap_handler behaviour
    ok = ecoap_registry:register_handler([
            {[<<"HelloWorld">>], ?MODULE}  % ?MODULE refers to current module
    ]).

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, [<<"HelloWorld">>], Suffix, _Request) ->
    % Payload would be <<"HelloWorld/Suffix1/Suffix2/..">> or <<"HelloWorld">> with no suffix
    Payload = lists:foldl(fun(Seg, Acc) -> <<Acc/bytes, "/", Seg/bytes>> end, <<"HelloWorld">>, Suffix),
    % reply with 2.05 payload (text/plain)
    {ok, coap_content:new(Payload, #{'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, Name, _Request) ->
    % do something
    {ok, {state, Prefix, Name}}. % {ok, SomeState} where SomeState could be anything

coap_unobserve({state, _Prefix, _Name}) ->
    % do something
    ok.

% observe notifications from ecoap_handler:notify/2
handle_info({coap_notify, Msg}, _ObsReq, State) ->
    {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 4: ecoap server API

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 and only required callbacks are included.

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/{0,1}` to spawn and links to a `ecoap_client` process and `ecoap_client:close/1` to stop it. The optional parameter of the open function can be used to specify the port to be bound or the socket manager to be associated with as an embedded client. Synchronous requests can be sent through `ecoap_client:request/3,4,5,6` and asynchronous requests via `ecoap_client:request_aysnc/3,4,5`. Similarly, there is `ecoap_client:observe/2,3` and `ecoap_client:observe_and_wait_response/2,3` as well as their unobserve counterparts. The only difference between the two is that the latter will block until the first notification or another valid response is returned from the server while the former is completely asynchronous. The number of parameters varies among the functions so that one can have more refined modifications against 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.

```
{ok, Client} = ecoap_client:open().  % use a random port by default
Response = ecoap_client:get(Client, "coap://example.com:5683/HelloWorld").
case Response of
    {ok, {ok, Code}, Content} -> io:format("Success: ~p ~p~n", [Code, Content]);
    {ok, {error, Code}, Content} -> io:format("Error: ~p ~p~n", [Code, Content]);
    {error, Reason} -> io:format("Fail: ~p~n", [Reason]);
end.

catch ecoap_client:put(Client, "coap://example.com:5683/HelloWorld", <<"some payload">>,
#{'Content-Format' => <<"text/plain">>}, 5000) of
    {'EXIT', {timeout, _}} -> io:format("Request time out after 5 seconds~n");
    {'EXIT', Reason} -> exit(Reason);
    Response -> Response
end.
ok = ecoap_client:close(Client).
```

Listing 5: Synchronous client API

A synchronous call blocks the caller until a valid response is delivered. If the server is unreachable and the request timeouts, `{error, timeout}` will be returned. An optional timeout can be set, which will crash the caller if no response arrives within this time. For asynchronous calls, since the response are delivered as message, one can use either the `receive` primitive in Erlang or let an OTP process handle it. In the observe example, 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 cancelled. This is not the only way and unnecessary if being used within an OTP process where messages are processed one after another.

```
{ok, Client} = ecoap_client:open().  % use a random port by default
{ok, Ref} = ecoap_client:get_async(Client, "coap://example.com:5683/HelloWorld").
receive
    {coap_response, Ref, Client, Response} ->
        % do something
after 5000 ->
        % do something
end.
{ok, Ref2} = ecoap_client:get_async(Client, "coap://example.com:5683/HelloWorld").
ok = ecoap_client:cancel_request(Client, Ref2).  % cancel the asynchronous request
ok = ecoap_client:flush(Ref2).  % flush in case the response is delivered already
ok = ecoap_client:close(Client).
```

Listing 6: Asynchronous client API

```
{ok, Client} = ecoap_client:open().  % use a random port by default
{ok, Ref} = ecoap_client:observe(Client,  "coap://example.com:5683/observable").
Pid = spawn(fun() -> get_notification(Client, Ref) end).  % receive notifications in another process
% ...
Pid ! proactive_cancel.  % send the cancel command
% ...
ecoap_client:close(Client).

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 not observable/deleted
            % 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 7: Observe API

Chapter 5

Evaluation

5.1 Experiment Setup

5.1.1 Benchmark Tool

5.1.2 Setup

5.2 Muti-core Scalability

5.3 Throughput Verification

5.4 Fault-Tolerance Test

5.5 Discussion

Chapter 6

Conclusion

6.1 Summary

6.2 Limitations and Future Work

References

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

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

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

[4] Joe Armstrong. Concurrency oriented programming in erlang. *Invited talk, FFG*, 2003.

[5] Luigi Atzori, Antonio Iera, and Giacomo 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`.

[6] S. Bandyopadhyay and A. Bhattacharyya. Lightweight Internet protocols for web enablement of sensors using constrained gateway devices. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 334–340, January 2013. DOI: `10.1109/ICCNC.2013.6504105`.

[7] Andrew Banks and Rahul 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.

[8] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh 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`.

[9] 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`.

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

[11] 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.

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

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

[14] George Candea and Armando Fox. Crash-Only Software. In *HotOS*, volume 3, pages 67–72, 2003.

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

[16] 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).

[17] 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`.

[18] Simone Cirani, Marco Picone, and Luca 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.

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

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

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

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

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

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

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

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

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

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

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

[30] C. Esposito, S. Russo, and D. Di 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`.

[31] Christian 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.

[32] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[33] 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`.

[34] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007:1–16, 2012.

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

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

[37] Jim 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.

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

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

[40] 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`.

[41] 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`.

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

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

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

[45] Matthias Kovatsch, Simon Duquennoy, and Adam 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.

[46] Matthias Kovatsch, Martin Lanter, and Zach 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.

[47] Koojana Kuladinithi, Olaf Bergmann, Thomas Pötsch, Markus Becker, and Carmelita Görg. Implementation of CoAP and its application in transport logistics. *Proc. IP+ SN, Chicago, IL, USA*, 2011.

[48] Martin Lanter. Scalability for IoT Cloud Services, 2013.

[49] Edward 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`.

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

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

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

[53] Gerardo Pardo-Castellote, Bert Farabaugh, and Rick Warren. An introduction to DDS and data-centric communications. *Real-Time Innovations. OpenURL*, 2005.

[54] 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`.

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

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

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

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

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

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

[61] 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`.

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

[63] Alessandro Sivieri, Luca Mottola, and Gianpaolo 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.

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

[65] Sriram 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.

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

[67] 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.

[68] 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.

[69] Héctor Pérez Tijero and J Javier Gutiérrez. On the schedulability of a data-centric real-time distribution middleware. *Computer Standards & Interfaces*, 34(1):203–211, 2012.

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

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

[72] 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.

Appendix A

Sample Appendix

Stuff for this appendix goes here.

Appendix B

Another Sample Appendix

Stuff for this appendix goes here.