UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Advanced Congestion Control Mechanisms for Internet of Things

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Luis Aldana Parra

In partial fulfilment

of the requirements for the degree of

MASTER IN TELECOMMUNICATIONS ENGINEERING

Advisor: Ilker Seyfettin Demirkol

Carles Gomez Montenegro

Barcelona, October 2019

Title of the thesis: Advanced Congestion Control Mechanisms for Internet of Things

Author: Luis Aldana Parra

Advisor: Ilker Seyfettin Demirkol and Carles Gomez Montenegro

Abstract

The number of IoT devices is growing at high speed, around 18 billion devices are forecast by 2022. Many of these devices are implemented with simple hardware, with low specifications and low resources. Taking into account the limited hardware resources and the huge network formed by IoT devices, CoAP was born as a lighter application protocol than HTTP. One important task for this scenario is the congestion control of huge networks using simple hardware devices. CoAP implements a simple congestion control solution, but many research articles show that this solution is not very efficient and it could be improved using other congestion control algorithms. CoCoA was born with the aim of being the standard congestion control algorithm for CoAP and has been proven through many studies, that it improves CoAP default performance in several scenarios. However, some research articles show that CoCoA offers low performance in bufferbloat scenarios.

This thesis evaluates CoCoA in bufferbloat scenarios and introduces changes on CoCoA algorithm, achieving an improvement on its performance.

Acknowledgements

I would like to thank Dr. Ilker Seyfettin and Dr. Carles Gomez for their expertise, supervision, assistance, guidance and patience throughout the process of writing this thesis.

I would extend my sincere gratitude to Dr. Josep Paradells for facilitate the materials needed to carry out with the thesis, and to Dr. August Betzler to transferring to me his knowledge and information needed to start this thesis.

Revision history and approval record

Revision	Date	Purpose
0	17/09/2019	Document creation
1	27/09/2019	Document revision
2	03/10/2019	Document revision

Written by:		Reviewed and approved by:	
Date	17/9/2019	Date	8/10/2019
Name	Luis Aldana Parra	Name	Ilker Seyfettin Demirkol Carles Gomez Montenegro
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

1. Introduction

This thesis evaluates CoCoA in bufferbloat scenarios and proposes changes to the algorithm in order to improve it. This necessity is born from research work that shows the CoCoA performance, in bufferbloat scenarios, may in some cases be worse than default CoAP congestion control performance[2].

1.1. Requirements and specifications

There are three main requirements for this thesis:

1. The first one is to create a network configuration scenario that may be susceptible to suffer bufferbloat.
2. The second on, is to use CoAP clients and CoAP servers in order to test CoAP and CoCoA.
3. The last one is to improve CoCoA algorithm to perform better results on bufferbloat scenarios.

The specification for the first requirement is to use Netem to add delays, bit rate and buffer dimensions to network interfaces in order to emulate specific communication links.

To face the second requirement, 400 clients are going to be emulated on a single computer using parallel threads. Each thread would be a single client that sends GET requests to a CoAP server allocated on a different computer, at the other side of the emulated link. Both, server and clients, would be developed on Java using Californium[12].

Finally, the improvements on CoCoA algorithm will be developed directly on the Californium source code used on congestion control layer for CoCoA.

1.2. Statement of purpose

The purpose of the thesis is to evaluate CoCoA in buffer bloat scenario and improve its performance.

In order to test the performance, this study will be focused on three metrics:

1. Flow completion time, FCT.
2. Medium round trip time, RTT.
3. Medium number of retransmissions per flow.

Taking into account these three metrics, improving performance implies getting lower FCT, lower RTT and less retransmissions per flow.

Improving the performance in one scenario should not affect performance in other network links or configurations. For this reason, all modifications will be tested also in lossy link scenarios and other link speed connections.

1.3. Methods and procedures

As the main goal on this thesis is to improve the results shown on FASOR's paper[2], first step would be setting the configuration as it is described on this article and check the results shown. Then, other scenarios and configurations would be tested to verify that improvements are not affecting performance on other situations.

1.3.1. Replicate FASOR's paper results for CoAP and CoCoA

Procedure 1: Replicate network and link conditions

Specific network configuration and link conditions are shown on FASOR's article. In order to replicate them, it is necessary to use Netem from Linux as it is exposed in the paper.

Procedure 2: Replicate CoAP Client and Server behaviors

Specific number of clients, packets and two ways of transmissions are given by the paper. So, it is necessary to code client and server on Californium to obtain similar behavior on packet transmission scenarios.

Procedure 3: Check similarity between results obtained and the ones exposed on the paper.

1.3.2. Implement CoCoA improvements proposals

Procedure 1: Code the proposed improvements on Californium.

Procedure 2: Check new proposal's results.

Run the same tests for CoAP and CoCoA with the congestion control modifications.

Procedure 3: Check congestion control changes in no bufferbloat scenarios

1.3.3. Compare the results and take conclusions

Procedure 1: Compare FCT, RTT and number of retransmissions.

Procedure 2: Take conclusions based on the comparison and the network conditions**.**

1.4. Work plan

1.4.1. Work Packages

Understand buffer bloat problem	1 April – 13 April
• Read FASOR's paper and understand the exposed problem.	1 day
• Read CoCoA papers and understand how CoAP and CoCoA works.	3 days
• Research about bufferbloat problems and related papers about this topic applied on CoAP.	5 days
• Propose an hypothetical solution.	3 days

Work environment set up	14 April – 21 April
• Install Linux on all computers used.	1 day
• Install Eclipse and Californium on Client and Server computers.	3 days
• Connect al computers via Ethernet to a create a LAN.	1 day
• Route all computers traffic to simulate a Client - Router - Server configuration.	1 day

Link simulation set up	22 April – 28 April
• Learn Netem.	3 day
• Apply Netem.	1 day
• Check link status.	1 day

CoAP simulation set up	29 April – 26 May
• Understand Californium code.	1 week
• Set up a CoAP server with Californium.	1 week

• Code 400 simultaneous clients using Californium clients.	2 week
• Check CoAP client-server communication	1 week

New proposed improvements set up	27 May – 13 June
• Implement algorithm improvements.	1 week
• Implement code to extract necessary metrics.	1 week
• Check new implementations.	3 days

Simulations	14 June – 14 August
• Run simulations on each proposed scenario for each congestion control mechanism.	2 weeks
• Understand the simulations results.	2 weeks
• Adjust link and client configurations to obtain more significant results.	1 week
• Run again simulations.	3 weeks

Conclusions	27 Aug – 3 Sep
• Compare the results between FASOR's paper and this results.	2 day
• Evaluate the results obtained and take conclusions.	2 day
• Suggest an improvement if it is necessary.	2 day

Write the Master Thesis	4 Sep – 19 Sep

1.5. Deviations

Unfortunately, some deviations have been found during the set up steps and simulation steps, due to a lack of information about the original configuration of [2], which was to be replicated for this thesis and time spent on getting all the material needed. Fortunately, the initial scope of the project has been fulfilled. However, because of these deviations, some interesting improvements found during the thesis have not been implemented and tested, and are only shown as proposals.

After the first configuration the results did not match those observed on FASOR's paper[2], so different configurations have been tried in order to get similar results. Finally, after contacting with the authors of the article and asking few questions about the environment, the configurations were set up as they claimed that they did.

On the other hand, the initial scenario was proposed with two computers and the final one required four computers with two network interfaces each one. So, another deviation has been getting all these computers and setting them up because, initially, this possibility was not taken into account.

2. State of the art of the technology applied in this thesis

Internet of Things (IoT) has become a huge reality in the last few years. In 2018, around seven billion of IoT devices were connected and it is foreseen, that in the year 2020, the number will increase to around 10 billion devices and around 22 billion in 2025. The forecasted value of IoT market in 2025 is set on 1,567B$[3], an important number to take into account.

Many new IoT devices differ from nowadays current connected devices in the fact of being smaller and focused on simple tasks that require minimum computation resources and a long life, talking in terms of battery life. Another handicap for these devices is the fact of being wireless and sometimes in lossy networks. Because of this, lots of IoT devices need new lighter but robust enough protocols in order to be able to adopt them with few resources and being as efficient as possible in terms of energy.

In addition, these IoT devices could be very different from each other, with different tasks, different manufacturer, etc., and they need to communicate between them. Taking into account that the current application layer protocol used on most connected devices, HTTP, use large headers that are not appropriated for constrained devices. Furthermore, lots of constrained devices are nodes like sensors that only need to send data and there is no need to establish sessions. So, the need of using TCP was also questioned.

Due to this situation, the IETF proposed a new web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks on the RFC 7252[4]. This new protocol was born on 2014 and it is named CoAP (Constrained Application Protocol). It is the standard substitute for HTTP in constrained environments.

Observing this preview, it is easy to think that networks will have to support more and more traffic as long as more devices are connected. In addition, constrained devices, can trigger problems about congestion. Therefore, it becomes necessary to have some congestion control mechanisms.

The proposed congestion control on CoAP was very simple and very inefficient so new congestion controls where born. The first one, called CoCoA (Congestion Control Advanced), was born in 2015*[5][6]*. The second one, called FASOR (Fast Slow RTO), was born in 2018*[2]*. Both are trying to become standard congestion control algorithms for CoAP.

2.1. Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) is standardized and defined by the IETF RFC 7252. This document describes this protocol as specialized web transfer protocol for use with constrained networks. CoAP is built taking into account devices with 8-bit micro controllers, with small amount of ROM and RAM, constrained networks such as IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs) with high packet error rates and low throughputs, and it is designed for machine-to-machine (M2M) applications.

CoAP is based on a request/response model between endpoints, provides built-in discovery of services and resources, and incorporates Web key concepts such as URIs or Internet media types. Other considerations are the ability to easily interface with HTTP and

meeting specialized requirements such as multicast, low overhead, and simplicity for constrained environments[4].

2.1.1. CoAP features

The main features established on the CoAP RFC are:

- Web protocol fulfilling M2M requirements in constrained environments.
- UDP binding with optional reliability supporting unicast and multicast request.
- Asynchronous message exchanges.
- Low header overhead and parsing complexity.
- URI and Content-type support.
- Simple proxy and caching capabilities.
- A stateless HTTP mapping, allowing proxies to provide access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.
- Security binding o Datagram Transport Layer Security (DTLS).

2.1.2. CoAP Working Principles

CoAP is based on an interaction model based on client/server similar to HTTP model. However, machine-to-machine interactions are usually implemented with CoAP acting as both client and server. CoAP requests are similar to HTTP requests. The request is sent by a client to request an action, defined by a Method Code, on a server resource (Identified by the uniform resource identifier, URI). Then the server sends back a response with a response code which may include a resource presentation.

One of the most important differences between CoAP and HTTP is that CoAP works with asynchronous interchanges over a datagram-oriented transport such as UDP. This is done using a messages layer that supports optional reliability. Four types of messages are defined:

- Confirmable: Requires an acknowledgement.
- Non-confirmable: Does not require an acknowledge.
- Acknowledgement: Confirms a specific received message.
- Reset: Indicates that a specific message has been received but some context is missing to properly process it.

Instead of use two-layer approach, one for UDP messaging layer, and other for request/response methods and response codes, CoAP acts like a single protocol, handling messaging and request/response methods, as features of the CoAP Header (see Fig. 1).

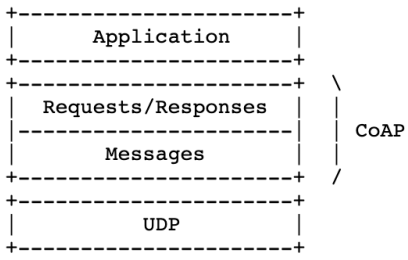

Figure 1 Abstract Layering of CoAP from IETF RFC 7252

2.1.3. Messaging Model

CoAP uses a short fixed-length binary header (4 bytes) followed by compact binary options and a payload. Messages are identified by a fixed 16-bit length Message ID that helps detecting duplicates and offers reliability.

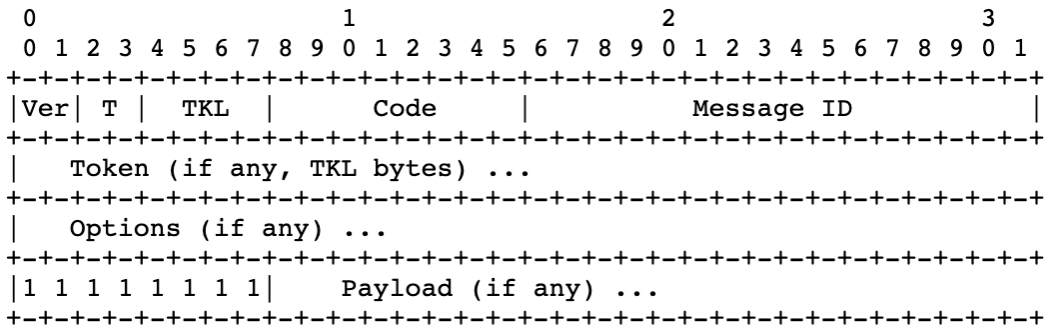

Figure 2 CoAP message format from IETF RFC 7252

If reliability is required, the message is marked as Confirmable (CON). Retransmissions are performed using a default timeout and with an exponential back-off between retransmissions until an acknowledgment (ACK) with the corresponding Message ID arrives from the corresponding endpoint. If the message has not been processed correctly a Reset message (RST) is sent back instead of the ACK.

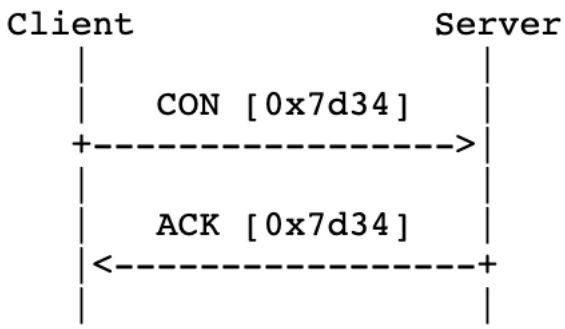

Figure 3 Reliable Message Transmission from IETF RFC 7252

If no confirmation is required, a non-confirmable message (NON) is sent. These messages are not acknowledged but also contain the Message ID in order to detect duplications and be able to send back a RST message if required.

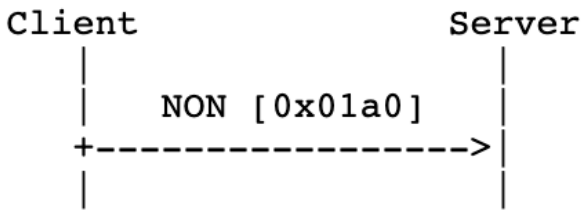

Figure 4 Unreliable Message Transmission from IETF RFC 7252

2.1.4. Request/Response Model

The Method Code and the Response code are carried in the CoAP header. The latter also contains default and optional information like the URI or media type as CoAP options and a Token used to match responses to requests independently from the underlying messages (do not confuse it with the message ID concept).

A request is carried on a CON or a NON message. If the message is a CON, the response code and the payload requested would be carried by the ACK message, proviing both functions, ACK and response, in a single message. This is called a piggybacked response. If the server receives the request but is not able at this moment to send the payload required sends and empty ACK. These methods allow the client to stop retransmitting and wait until the server could give a response.

Figure 5 Two GET requests with Piggybacked Response from IETF RFC 7252

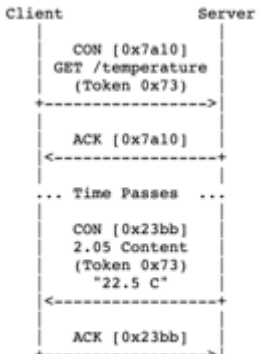

Figure 6 A GET Request with a Separate Response from IETF RFC 7252

In case of sending a NON the response would be another NON message.

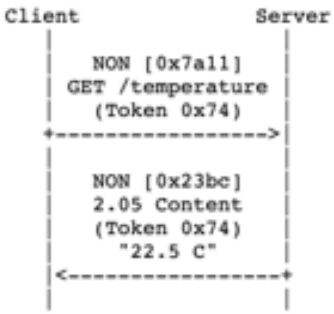

Figure 7 A request send a Response Carried in Non-confirmable messages from IETF RFC 7252

The main message methods used in CoAP are GET, POST, PUT and DELETE. These methods are similar to HTTP methods but there are some differences so it is recommended to see the specification on the RFC for more information.

2.1.4. Message transmission and congestion control

CoAP endpoints exchange asynchronously messages which transports requests and responses. This exchange is performed over UDP so messages could arrive out of order, duplicated or go missing. Because of this reason CoAP implements a lightweight reliability mechanism. The features of this mechanism are:

- Simple stop-and-wait retransmission reliability with exponential-backoff for CON messages.
- Duplicate detection.

When a message is marked as CON the endpoint sends the message and waits until an ACK arrives. If the ACK does not arrive before the initial timeout is expired, the endpoint retransmits the CON until the ACK arrives or the predefined value of maximum retransmissions allowed is reached. The timeout is set to a random duration between ACK_TIMEOUT and (ACK_TIMEOUT * ACK_RANDOM FACTOR). When the timeout expires the count of transmissions is incremented by one and the timeout is doubled.

Another important parameter which helps with congestion is the NSTART parameter. This parameter represents the number of simultaneous outstanding interactions that a client can maintain with a server at the same time. The default value of NSTART is set to 1. The client does not send a new message until the exchange of the CON is considered successful, the maximum number of retransmissions is reached or a RST message arrives.

More information and specifications about CoAP could be found in RFC7252.

2.2. CoCoA

CoCoA is a proposed advanced but simple CoRE congestion control designed to improve the default congestion control for CoAP. This mechanism is based on Retransmission Timeout (RTO) algorithm that uses Round-Trip Time(RTT) estimates[7], being the RTO the time to wait until the next retransmission, and the RTT, the time estimated from the packet transmission until the first valid ACK for this packet arrives.. The algorithm defined on the latest CoCoA draft is based on RFC6298, but introduces a number of extensions and modifications.

2.2.1 Advanced CoAP Congestion Control: RTO estimation

In the scenario where one CoAP endpoint needs to send several requests to one destination endpoint, a more appropriate RTO than the default initial time out of 2 to 3 s could be computed by basing the estimation on RTT measurements. The RTT is measured from the time when the packet is sent until a valid ACK for this packet arrives. CoCoA also defines two kinds of RTO, strong and weak. The strong RTO is the one computed with strong RTTs, defined as RTTs measured before any retransmission has been done. On the other hand, the weak RTO is the one that uses weak RTTs, defined as RTT samples from packets with one or two retransmissions[6]. Responses after the third retransmission are not used to update the RTO estimation.

- **Blind RTO Estimate:** The initial RTO is set to 2 seconds for both E_{weak} (weak estimator) and E_{strong} (strong estimator). The weak estimator is set for exchanges that have run into retransmissions, and only the first two retransmissions are considered. If there are different exchanges in parallel(NSTART=*n*), the RTO is set to *n* times 2s plus the 2s related to the actual exchange, e.g. if two exchanges are already running, the initial RTO estimate for an additional exchange is 6s.

- **Measurement-based RTO Estimate:** The overall RTO estimate is an exponentially weighted moving average computed of the strong and the weak estimator, which is evolved after each contribution to the weak estimator (1) or the strong estimator (2), from the estimator that made the latest contribution. Splitting this update in two cases avoids the RTO to grow over much in lossy networks[7].

$$RTO_{weak} = SRTT_{weak} + K_{weak} * RTTVAR_{weak}, \quad (1)$$

$$RTO_{strong} = SRTT_{strong} + K_{strong} * RTTVAR_{strong}, \quad (2)$$

SRTT and RTTVAR denote the well-known smoothed RTT and RTT variation computed as in RFC 6298, K_{strong} is 4 as in RFC 6298, and K_{weak} is 1. The newly calculated RTO contributes to an overall RTO value with a weighted average[5]:

$$RTO_{overall} = \alpha * RTO_x + (1 - \alpha) * RTO_{overall},$$

α is set to 0.25 to weak and 0.5 for strong estimators. These values were obtained through different evaluations in order to achieve the best performance.

- **Variable Backoff Factor (VBF):** CoAP uses a binary exponential backoff(BEB). This kind of backoff may not increase the RTO fast enough to allow the network to recover from congestion, keeps offering high load to the network and increasing the chance for spurious retransmissions when the initial RTO is small. Otherwise, when initial RTOs are large, it may grow too fast and produce an unnecessary large delay. Because of these reasons CoCoA applies a VBF. Several values for VBF have been evaluated. Taking into account the results of this evaluations, the values applied for the different VBF are:

Initial RTO < 1 s, VBF=3;

1s < Initial RTO < 3s, VBF=2;

3s < Initial RTO, VBF=1.5;

- **RTO Aging:** In IoT networks, conditions can change fast, so RTO values could became invalid if they are not updated for an extended period of time. To avoid this scenario CoCoA applies RTO Aging. If an RTO estimation is below 1 s or above 3 s, and no new RTT measurement is made for 16 or 4 times the current RTO, respectively, the RTO value is modified to approach the default initial value[6].

$$RTO_{overall} = (2+RTO_{overall})/2 \text{ s:}$$

- **Congestion Control for NON messages:** CoCoA introduces a rate limitation for NON messages towards a destination endpoint to one message every RTO. In order to measure RTTs and estimate RTOs, needed for the rate limitation, CoCoA forces the use of certain number of CON messages among the NON messages.

2.3. FASOR

FASOR is an alternative congestion control for CoAP defined on “Fast-Slow Retransmission and Congestion Control Algorithm for CoAP” draft on IETF website. This congestion control is based on RTTs too.

2.3.1. Algorithm

FASOR has three principal components: Fast RTO computation, Slow RTO and novel retransmission timer backoff logic[2].

- **Fast RTO:** The computation of this RTO is based on the RFC 6298 but without minimum RTO bound. Only strong RTT (unambiguous RTT) are used to update the estimator. The

initial RTO is set to 2s and RTO is initialized with the first unambiguous RTT sample. RTT variation initialization in FASOR uses the following formula:

$$RTTVAR \leftarrow R/(2K)$$

Where R is RTT and K take the value of 4.

- **Slow RTO:** This RTO is based on Karn's algorithm which does not reduce a backed off RTO until an ambiguous ACK has been received*[2]*. Slow RTO does not take into account the number of retransmissions and is measured from the initial transmission of the message until an unambiguous ACK arrives. In order to allow some growth in load but without being too aggressive, Slow RTO is multiplied by a factor (e.g. 1.5).
- **Backoff logic:** FASOR has three backoff states: FAST, FAST_SLOW_FAST and SLOW_FAST. These states differ on how conservative they are. The transitions between states are decided each time an ACK is received.

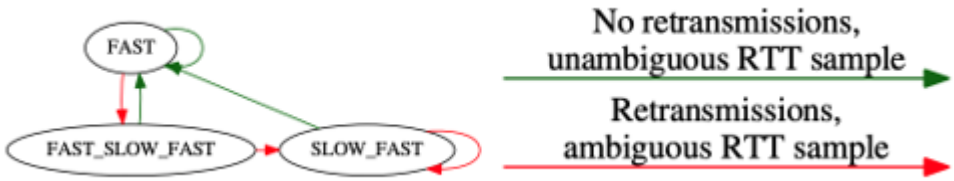

Figure 8 FASOR state diagram [2]

The FASOR senders always stay in or transit back to the FAST state when an ACK without retransmissions arrives. On the other hand, when an ACK with retransmissions is received, the state is downgraded to one more conservative. When the SLOW_FAST state is reached, the senders remains on this state until an ACK without retransmissions arrives.

Each state has its own different evolution of RTO values to back off the RTOs:

FAST: Fast, Fast x 2^1, Fast x 2^2, …
FAST SLOW FAST: Fast, max(Slow, Fast x 2), Fast x 2^1, Fast x 2^2, …
SLOW FAST: Slow, Fast, Fast x 2^1, Fast x 2^2, …

The sender always starts on the FAST state. If retransmissions are needed the state transits to FAST_SLOW_FAST. In this state, FASOR tries a Fast RTO and if the ACK arrives without retransmissions, it transits back to the fast state. Otherwise, if retransmissions are needed, the sender tries the Slow RTO and, in addition, the state transits to SLOW_FAST when an ACK arrives. This state starts using the Slow RTO for the next transmission. The intention of this logic is to allow the network to decongest from the unnecessary retransmitted messages on certain scenarios of bufferbloat.

- **FASOR + TOKEN:** This variation of FASOR introduces a modification on the token available on CoAP requests. The last bytes of the token are replaced for retransmission number of the message. The endpoint that receives the request echoes the token unmodified. This allows the original sender to compute an unambiguous RTT by taking into account the time between the retransmission, instead of the original message, and the corresponding ACK.

2.4. Bufferbloat

This congestion phenomenon occurs when large buffers receive packets faster than they can process them. This cause the packets to get stored in the buffer for a certain time increasing the latency of the network and causing jitter (packet delay variation).

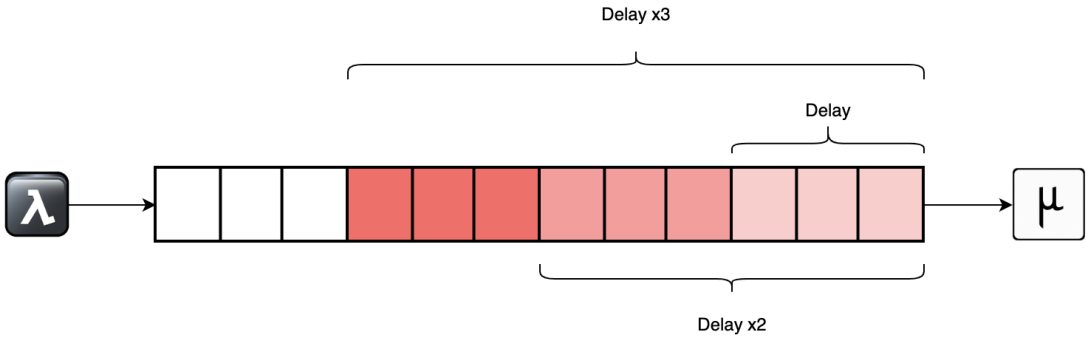

Figure 9 Delays produced by bufferbloat

This is a very well-known problem on the Internet where different types of traffic (HTTP, FTP, video, etc…) coexist heterogeneously. Most of the solutions to these problems are based on queue management algorithms which set priorities for each kind of traffic, for example, being the highest priority packet transmitted before the others. However, in the case of CoAP, this traffic suffers bufferbloat even when only CoAP packets are in the network. Because of this fact, queue management algorithms are not a complete solution in CoAP case.

As it is described in previous sections, congestion control mechanisms for CoAP are designed for being lightweight. Due to this fact, and taking into account that a CoAP packet does not indicate if it is a retransmission or the original packet, specific scenarios can suffer bufferbloat and even increase the network congestion itself.

The logic behind this statement is that, if the buffer is large enough and there is enough traffic to fill it, at certain point, the delay of the packets would be large enough to make the transmitter think that the packets are lost because the RTO would expire, so they would decide to retransmit unnecessarily causing more traffic on the network and increasing even more the congestion. In addition, the RTT can be very different from one transmission to another, so the algorithms are challenged to adapt correctly to the ongoing buffer state. For example, if the congestion control receives a low RTT sample and sets a low RTO and then, on the next transmission, the buffer is full, the transmitter would perform several unnecessary retransmissions. In the other hand, if the congestion control receives a large RTT sample and sets a large RTO and then the buffer is emptied it could cause the transmitter to wait too much to retransmit when it would be necessary in presence of packet loss.

Due to this fact, some variants of congestion control mechanisms are being studied. One example is the proposal of CoCoA++, using CAIA Delay Gradient (CDG) and Probabilistic Backoff Factor (PBF)*[8]*. This paper shows an improvement on the CoCoA behavior using

CDG, but CDG is thought for TCP and as long as CoAP runs on UDP it is complicated to adapt the CDG and becomes a complex solution.

A second example is FASOR which propose a stateful logic. The results contained on its paper, shows an improvement dealing with bufferbloat in comparison to CoAP and CoCoA.

Another example is the proposal of using retransmission match. As it is said before, CoAP does not perform a control if the packet received is the original or a retransmission. More accurate RTT can be estimated using a control of retransmission match as is described in "Round Trip Time Based Adaptive Congestion Control with CoAP for Sensor Network"[9].

The last example is pCoCoA, this proposal does not make use of the weak estimator and implements a transmission count option to match the ACKs. It initializes the variable and updates RTTVAR and the only smoothed round-trip estimator, SRTT. Finally, it employs a dynamic method to limit the minimum RTO, thus reducing spurious retransmissions[10].

2.5. Losses

CoAP is a protocol designed to work in lossy networks. Nonetheless, it was not designed taking into account mechanisms that use RTT or inform about retransmissions, so the CoAP packet does not contain any flag to differentiate if the packet received is the original one or a retransmission.

In case of congestion control mechanisms, as CoCoA or FASOR, based on RTO estimated through RTT, losses produce a miscalculation of the RTT. The RTT is calculated from the first transmission until an ACK arrives. As long as the received ACK does not contain any information about if the packet received is the original or a retransmission this RTT estimation could be larger than the real one.

For example, if the transmitter sends the original message, performs two retransmissions and the two first packets get lost, the final RTT estimated would be twice the RTO plus the real RTT. This fact can lead to delays on the communication because of large RTOs.

2.6. Netem

Netem is an enhancement of the Linux traffic control facilities that allow to add delay, packet loss, duplication and more other characteristics to packets outgoing from a selected network interface. Netem is built using the existing Quality Of Service (QOS) and Differentiated Services (diffserv) facilities in the Linux kernel*[11]*.

2.7. Californium

Californium is an open CoAP framework for Java*[12]*. This project is divided in five sub-projects:

- **Californium core:** Provides the main framework as the implementation of CoAP algorithm, CoCoA and other congestion control mechanisms and all the logic needed to create CoAP servers and clients.

- **Scandium:** The security for Californium is defined here. It implements DTLS 1.2 through ECC with pre-shared keys, certificates or raw public keys.
- **Actinium:** Is the app-server for Californium to realize IoT mashups, JavaScript apps become available as RESTful resources and can directly talk to IoT devices using the CoapRequest object API.
- **CoAP tools:** A pre-built set of tools for CoAP like predefined client and server implementations ready to use.
- **Connector:** This elements abstracts from the different transports CoAP can use like UDP, DTLS or interfaces to implement your own transport layer.

3. Methodology / project development

FASOR's paper "FASOR Retransmission Timeout and Congestion Control Mechanism for CoAP" shows a network configuration where CoCoA suffers from bufferbloat even more than default CoAP. In order to evaluate this result and find a solution for CoCoA, the first step done is try to replicate the results in the mentioned paper.

3.1. Results replication

First of all, the network and link conditions must be the same that are described in the paper. The second step is replicating the behavior between the CoAP clients and server. Finally, the results will be compared with those in the paper. The specifications of the network and link conditions correspond to a constrained wireless link defined by the rate of the uplink and downlink, delays on both directions and a buffer with three different sizes.

3.1.1. Network replication

The specifications of the network and link conditions corresponds to a constrained wireless link defined by the rate of the uplink and downlink, delays on both directions, a buffer with three different sizes and a standard Internet delay:

- **Uplink:** Rate 60 kbps, delay 200 ms
- **Downlink:** Rate 30 kbps, delay 400 ms
- **Buffer:** Sizes 2500 B, 28200 B, 1410000 B

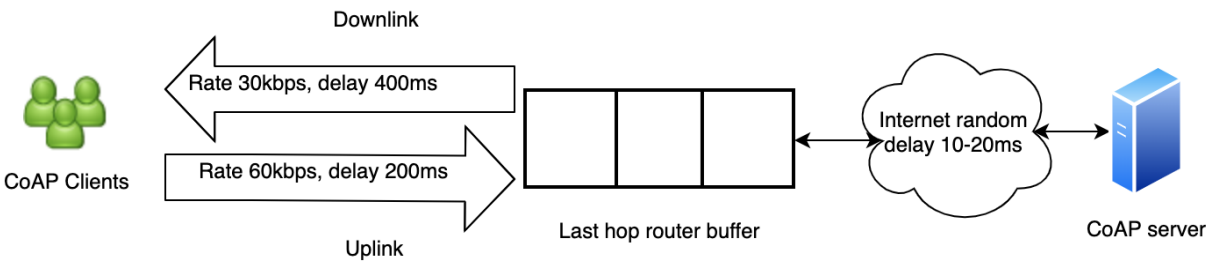

Figure 10 Network schema

- **Internet delay:** 10-20 ms random

The link conditions are emulated on Ubuntu environment and using Netem for adding rate, delay and buffer conditions to the link.

Some tests with loss are performed too. The loss model described in the paper proposes two cases, medium and high loss. The medium case corresponds to a two state model where the loss is 50% for the bad state and 0% for the good state. The high case corresponds to a two state model where the loss is 80% for the bad state and 2% for the good state. On these tests the losses are represented as a Gilbert-Elliot model.

In this model 1-k represents the probability of losses on a good state and 1-h represents the probability of losses on a bad state. Then, transition probabilities are described as p, from good to bad state and r from bad to good state. The state diagram is shown in Figures 11 and 12. the next page.

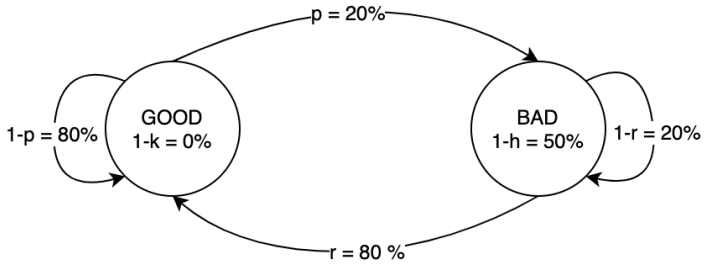

Figure 11 Medium loss states

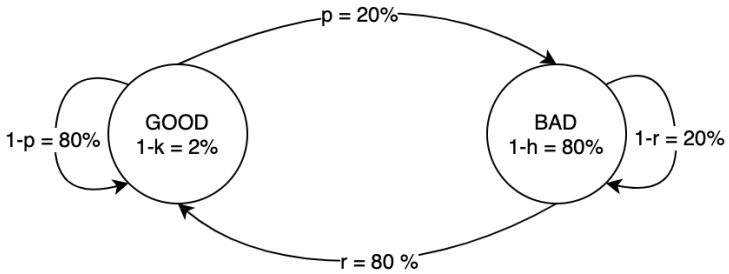

Figure 12 High loss states

The first approach in this thesis makes use of two hosts, one for CoAP clients and the other one for CoAP server. Each host has only one network interface connected directly between them via Ethernet (see Figure 13).

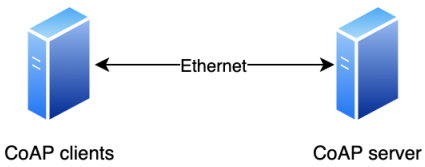

Figure 13 Physical network configuration

In order to emulate the wireless properties and the default Internet delay, two Netem statements in each direction are required. Netem only can use one statement per network interface, so the workaround has been to create a virtual network interface for the incoming packets. This way, each host disposes of a physical network interface for the outgoing packets and a virtual network interface for incoming packets.

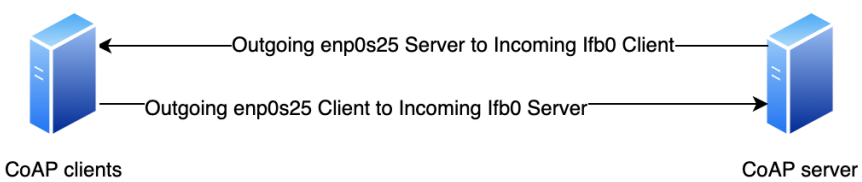

Figure 14 Virtual network configuration

The Linux commands used to create this virtual interface are the following:

 # modprobe ifb

 # ip link set dev ifb0 up

Netem commands to emulate the **client side**:

 # tc disc add dev enp0s25 root netem delay 200ms rate 60kbit limit <buffer size> (emualtes the uplink conditions)

 # tc disc add dev enp0s25 root netem delay 200ms rate 60kbit limit <buffer size> loss gemodel 80 20 50 0 (only for tests with losses).

Netem only introduces conditions on the packets processed ion the egress traffic (outgoing traffic). In order to add conditions to the ingress traffic (incoming traffic), it requires a redirection to a network interface in order to convert the ingress to egress traffic:

 # tc qdisc add dev enp0s25 ingress (set the ingress traffic)

 # tc filter add dev enp0s25 parent ffff: protocol ip u32 match ip src <server ip> flowid 1:1 action mirred egress redirect dev ifb0 (this command redirects the ingress traffic with the server's IP to the ifb0)

 # tc qdisc add dev ifb0 root netem delay 200ms (add the downlink delay)

 # tc qdisc add dev ifb0 root netem delay 200ms loss gemodel 80 20 50 0 (only on the losses case)

Netem commands to emulate the **server side**:

 # tc qdisc add dev enp0s25 root netem delay 15ms 5ms rate 30kbps limit <buffer size>(adds random internet delay 10-20ms the rate for the downlink and the buffer size)

 # tc qdisc add dev enp0s25 ingress

 # tc filter add dev enp0s25 parent ffff: protocol ip u32 match ip src 10.0.0.2/32 flowid 1:1 action mirred egress redirect dev ifb0

 # tc qdisc add dev ifb0 root netem delay 15ms 5ms (add the random internet delay)

The schema of Netem configuration:

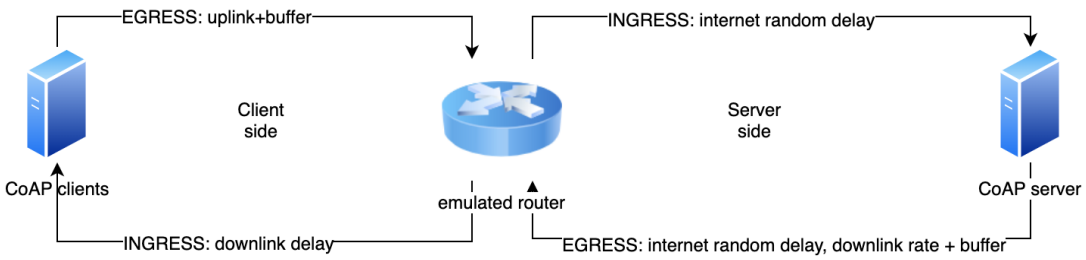

Figure 15 Netem emulated link schema

3.1.2. Client & Server behavior replication

FASOR's paper use 400 emulated clients using LibCoAP[13] implementation of CoAP. In this thesis the tests are going to be run on Californium because is where CoCoA is implemented by CoCoA authors. The paper also defines two different client behaviors or work load flows.

The first one, called continuous, consists of 400 clients, each one exchanging 50 CoAP request-response pairs until all of them have been successfully completed. On the other hand, the random flow consists of a series of short-lived CoAP clients which generates a random number, between 1 and 10, of CoAP request-response pairs until altogether 50 pairs have been successfully exchanged. The CoAP state is reset after each short-lived client.

The continuous flows allow congestion control to adapt RTOs during the 50 exchanges while in the random flows the RTOs only can be adapted between 1 or 10 exchanges. This difference challenges the congestion controls to estimate the RTO fast and precisely, otherwise the congestion control fails.

In order to emulate this work load behavior, two types of client flow and a server are implemented based on using Californium. The code can be found on the appendix.

3.1.3 Results comparison

Each test has been repeated 20 times in order to avoid random behaviors. So, all the values obtained are the average value from 20 experiment repetitions. The buffer sizes tested are 2500 bytes, 28200bytes and 1410000 bytes. The results of the first implementation, where we use only two PCs, do not match those described in FASOR's paper (Figure 17):

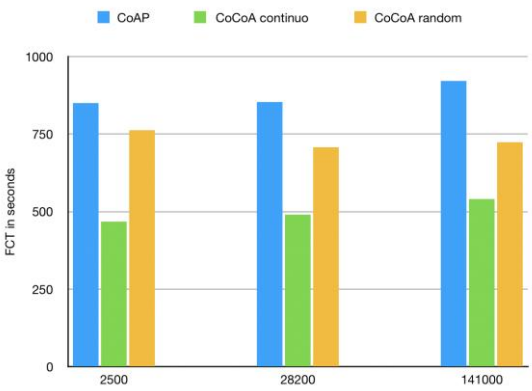

Figure 16 FCT Results of 400 flows

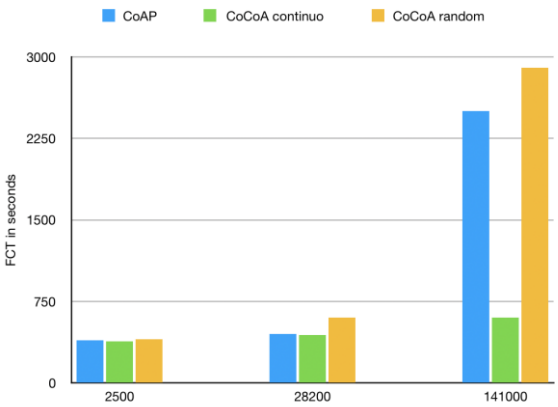

Figure 17 FASOR's paper FCT Results

As it can be seen on Figures 16 and 17 tests runs are completely different and the bufferbloat scenario does not appear. These results are because the buffers do not fill up completely and there is no extra delay corresponding to the size of the buffers.

In order to obtain bufferbloat, the tests have been reproduced increasing the payload on the downlink. On FASOR paper, the payload is not given nor the bottleneck direction. As long as the downlink is the one with the lower rate, it is the one which is going to suffer buffer bloat with minimum payload. So, the response payload is increased from 68 bytes to 298bytes on the next tests (Figure 18):

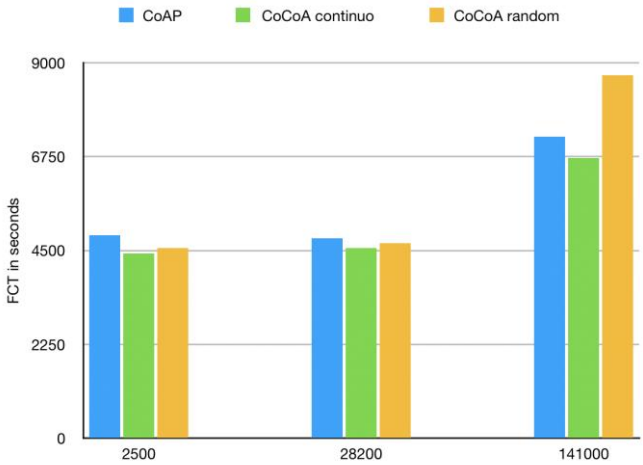

Figure 18 FCT Results for 296 bytes payload

This tests do not match the results neither, but bufferbloat and CoCoA performing worse than CoAP behaviors appear. Although the buffer bloat scenario can be achieved with this configuration, a new network configuration is also configured after requesting the original configuration from the authors of the FASOR document, taking into account the information that is missing in the document, such as the size of the payload, the bottleneck direction and the exact configuration with Netem.

The new configuration uses four computers instead of the original one, which uses only two. This is because Netem sometimes presents an incorrect behavior when delay and rate are set on the same interface.

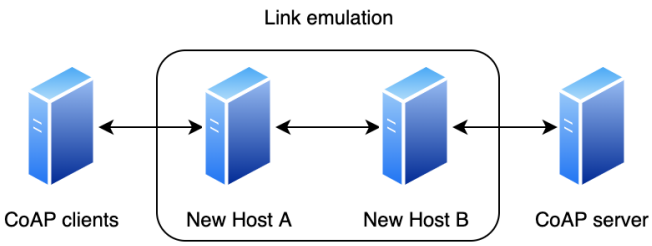

Figure 19 Physical Network Configuration Schema

In the new configuration, the client and the server hosts only perform as a client or server respectively with no Netem variations added. The new two hosts need two network interfaces each one to be able to connect via Ethernet to the other and between them.

Each interface has been applied different commands splitting the delay and rate modifications for better performance:

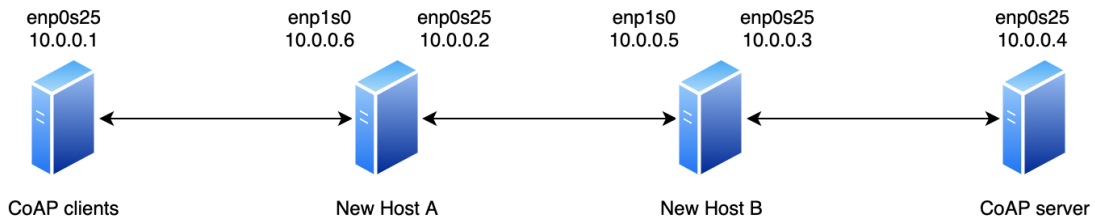

Figure 20 Virtual Network Configuration Schema

- **10.0.0.2:** On this interface, the total uplink delay is emulated, that is, 200 ms from the wireless link and 10-20 ms from the Internet random delay. Each time a Netem statement is set, a default buffer is added to the interface. The default value is 1000 bytes. For this reason, a two times buffer size is set on each interface, except on the one that implements the downlink rate, considered the bottleneck.

  # tc qdisc add dev enp0s25 root netem delay 215ms 5ms limit <buffer size x2> (loss gemodel 20 80 80 2)

- **10.0.0.3:** On this interface, the uplink rate is emulated.

  # tc qdisc add dev enp0s25 root netem rate 60kbit limit <buffer size x2>

- **10.0.0.5:** On this interface, the downlink delay is emulated.

  # tc qdisc add dev enp1s0 root netem delay 415ms 5ms limit <buffer size x2> (loss gemodel 20 80 80 2)

- **10.0.0.6:** On this interface, the downlink rate and the bottleneck is emulated.

  # tc qdisc add dev enp1s0 root netem rate 30kbit limit <buffer size>

On the other hand, the response size used in FASORs paper is 60 bytes, so setting the payload to 68 bytes on the same scenario should show the same pattern on the results. However, after setting this new network configuration the results do not match the expected results on the FASORs paper neither (Figure 21):

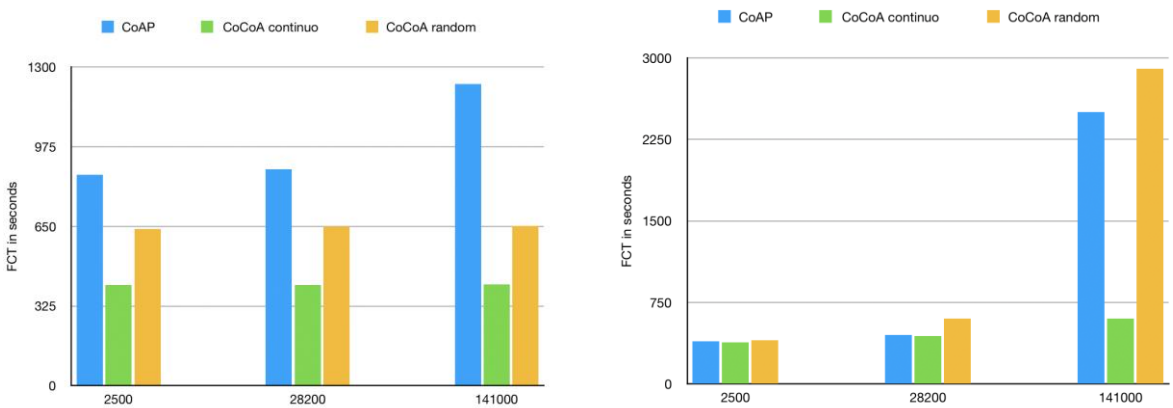

Figure 21 At left, FCT Results for 400 flows, at right FASOR's paper results[2]

And again if the payload size is increased buffer bloat and a worse performance of CoCoA appears for random flow scenario (Figure 22):

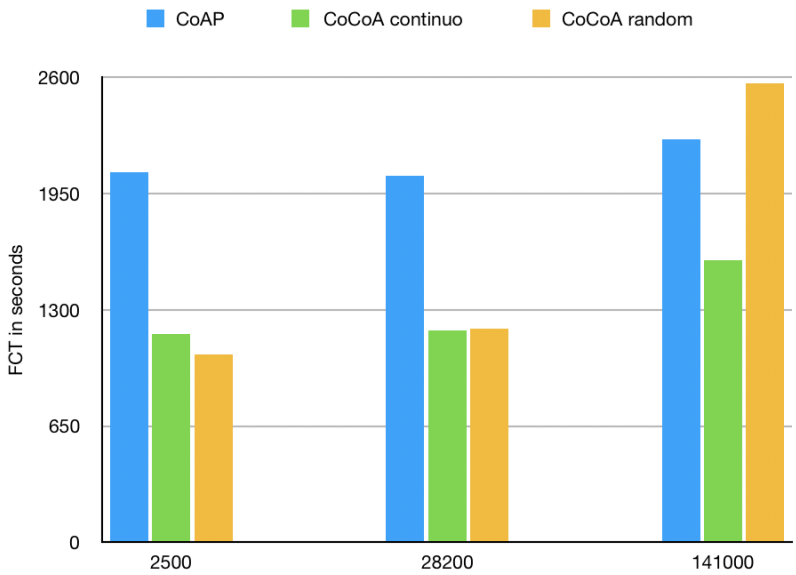

Figure 22 FCT Results for 176 bytes payload

As long as, it has not been possible to reproduce the pattern on the results on FASOR's paper, the case of a greater payload that generates bufferbloat is the one used to test the proposed improvements for CoCoA.

3.2. Implementation and tests of CoCoA proposed improvements

The improvements proposed in this thesis are based on the hypothesis that CoCoA does not respond correctly to bufferbloat scenarios because it ignores the RTT estimation when more than two retransmissions are required. CoCoA uses strong RTTs if the ACK arrives without retransmitting any time and weak RTTs if the number of retransmissions is one or two when the ACK arrives. So, the case of having more than two retransmissions is not taken into account, the congestion control does not update the RTO and keeps retransmitting.

Because of that reason, the proposed improvements are based on considering the weak RTT for more than two retransmissions. The maximum number of allowed retransmissions per exchange on these tests is set to 20. So, the number of retransmissions take it into account for these tests is also set to 20.

3.2.1. Improvements implementation

When an ACK arrives, CoCoA decides if this ACK is used to calculate a Strong RTT (ACK before any retransmission), a weak RTT (ACK when 1 or 2 retransmissions occur) and do not take into account the ACK if the retransmission is greater than 2. In Californium, "RemoteEndpoint.java" is the class in charge of setting the type of the RTT and update or

initialize the RTOs. More concretely, the function that sets the estimator type is called “setEstimatorState”. In order to implement the new Weak RTT proposal the only modification in the code that should be done is modifying the “if” statement of the failed retransmissions, as it can be observed commented on the figure below:

```
public void setEstimatorState(Exchange exchange){
		//When no CC layer is used, the entries are all null, check here if
this is the case
		if(exchangeInfoMap.get(exchange) == null){
			return;
		}
		if(exchange.getFailedTransmissionCount() == 1 ||
exchange.getFailedTransmissionCount() == 2){
//Only allow weak estimator updates from the first or second retransmission
			exchangeInfoMap.get(exchange).setTypeWeakEstimator();
		/*if(exchange.getFailedTransmissionCount() < 20) {
//Allow weak estimator until 20 retransmissions
exchangeInfoMap.get(exchange).setTypeWeakEstimator();//Modifications for the
tests
		}else{
//If more than 1 retransmission was applied to the exchange, mark this entry as
not updatable
			exchangeInfoMap.get(exchange).setTypeNoEstimator();
		}
	}
```

3.2.2. New implementation tests

All tests previously done are now repeated with the weak RTT modification. The new variation of CoCoA has a very similar performance to the original on the low payload (68 bytes) cases (Figure 23):

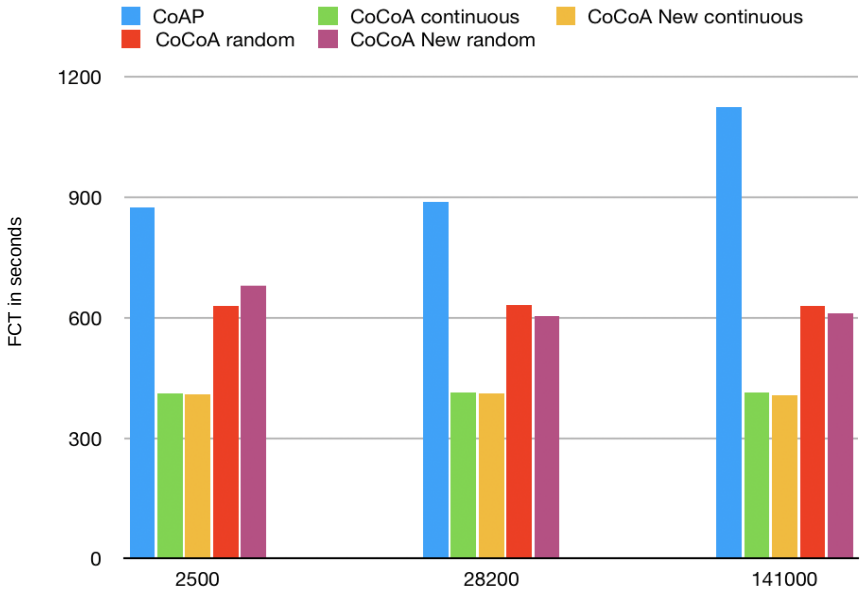

Figure 23 FCT Results for modified CoCoA

But it performs significantly better on the cases with 176 bytes payload (Figure 24):

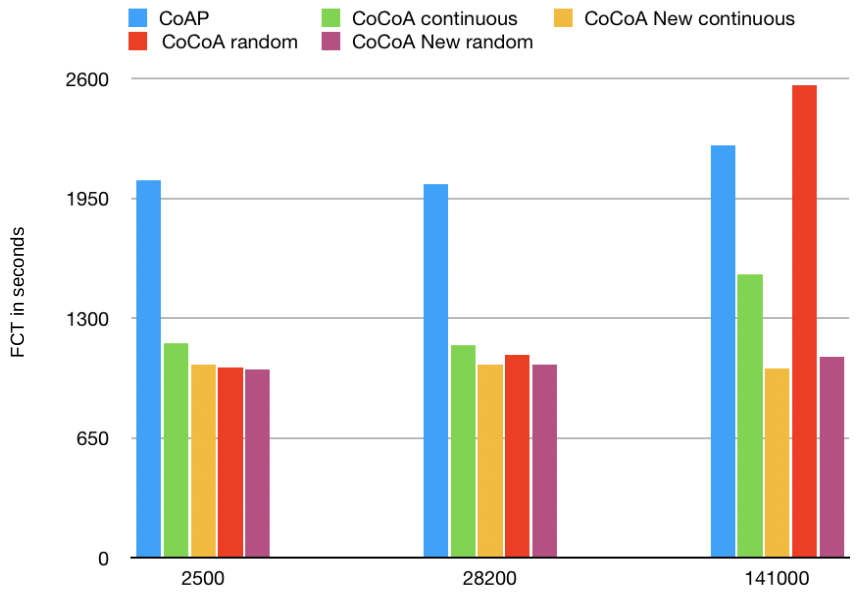

Figure 24 FCT Results for modified CoCoA with 176 bytes payload

4. Results

The results will be analyzed separately by the different network configurations implemented. Then, on the summary section, the results will be analyzed globally. In order to perform the analysis, three metrics will be taken into account. The first one is the flow completion time (FCT), this metric evaluates the time since the first packet is sent until the last ACK of all the clients arrive. The second is the average RTT, this metric is the medium RTT of all the exchanges over all the flows. Remember that the RTT is evaluated from the first packet sent until the corresponding ACK to this packet arrives without taking into account retransmissions. Finally, the last metric is the number of retransmissions performed per flow. All tests are repeated twenty times and the value shown on the graphics corresponds to the average value of all the tests, as well as the maximum and minimum values obtained.

4.1. Two hosts network configuration

In this section, the results of the network configuration using two hosts are shown and analyzed.

4.1.1. Payload 68 bytes, non-lossy link

The results for the average flow completion time running 400 clients is shown in the figure below:

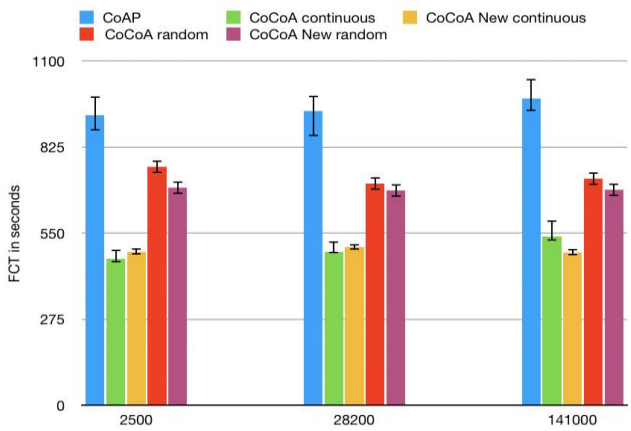

Figure 25 Average FCT for 400 clients

As it could be observed in the Figure 25, the behavior is really similar regardless of the buffer size and CoAP performs worse than CoCoA in all cases. Continuous flows using CoCoA and modified CoCoA, reduce the FCT from CoAP in 49.51% and 47.03% respectively for the low buffer size, 47.77% and 46.17% for the medium buffer size, and 44.90% and 50.10% for the large buffer size. Random flows of both, CoCoA and modified CoCoA, performs better compared to CoAP but worse compared to continuous cases. Concretely, reducing the FCT from CoAP in 17.80% and 24.91% respectively for the low buffer size, 24.68% and 27.02% for the medium buffet size, and 26.12% and 29.69% for the large buffer size, respectively.

Continuous flows perform better than random flows because the endpoint life in random flows is shorter than in the continuous flows, allowing the continuous flows endpoint to estimate the RTO better as long as it bases the estimation on fifty RTT while on random cases the estimation is based on 1 to 10 RTT.

Otherwise, the performance of modified CoCoA and CoCoA is very similar in most of the cases. This fact would be explained by observing the retransmissions graphic (Fig. 27). In addition, the little differences that may be appreciated are because of the randomness subjected to the environment. This randomness appears due to the random functions on the Java implementation. As long as each test is run with a different seed, all cases are not tested exactly on the same conditions. However, as long as the tests are repeated 20 times the average value provides results close enough to reality. Finally, it can be observed that the variance on the FCT is smaller on the modified CoCoA than on CoCoA. This is due to the fact that new modifications, generate more conservative RTT estimation taking into account more retransmissions, which leads to a better estimation of the RTO in the case of buffer congestion, avoiding random RTO initializations.

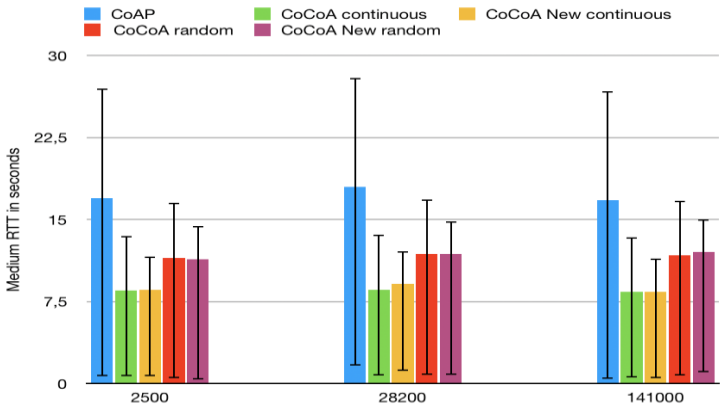

Figure 26 Average RTT for 400 clients

Again, it is possible to observe that in the case of RTT the behavior does not differs significantly depending on the buffer size. The different values on the medium RTT estimated in each case can be explained by taking into account the number of retransmissions. While more retransmissions are performed, the buffers fill up more, causing larger RTTs.

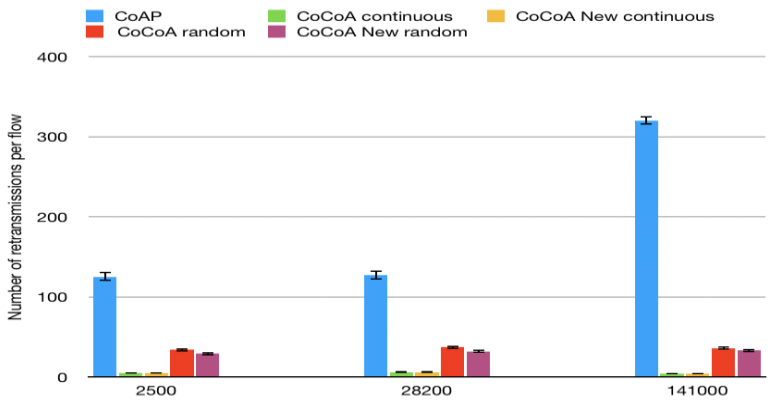

Figure 27 Average number of retransmissions per flow for 400 clients

Figure 27 provides the information needed to explain why the behavior between CoCoA and modified CoCoA is not too different, and also, why the RTTs are different on the different cases.

As long as the modifications on CoCoA only affect the use of more retransmissions, the beneficial effect only appears on those exchanges that retransmit more than two times. The number of retransmissions per flow (50 request-response pairs) that this scenario produces, leads to less than one retransmission per exchange. Because of that reason, the expected beneficial effects from the modifications do not appear.

In the case of CoAP, there are so many retransmissions that when the last ACK arrives the network remains congested for several seconds transporting lots of unnecessary retransmissions. Moreover, the congestion is increased even more because of non CoAP packets, such as ICMP packets in response of those retransmissions that do not find an available endpoint, as long as the endpoint is shut down after receiving the last ACK.

4.1.2. Payload 150 bytes, no loss

These tests are run using 200 flows instead of 400 in order to save time. After increasing the payload to 150 bytes to generate more congestion and see the beneficial effects for CoCoA modifications, the phenomena described on FASOR's paper where CoCoA performs worse than CoAP in some cases, arises.

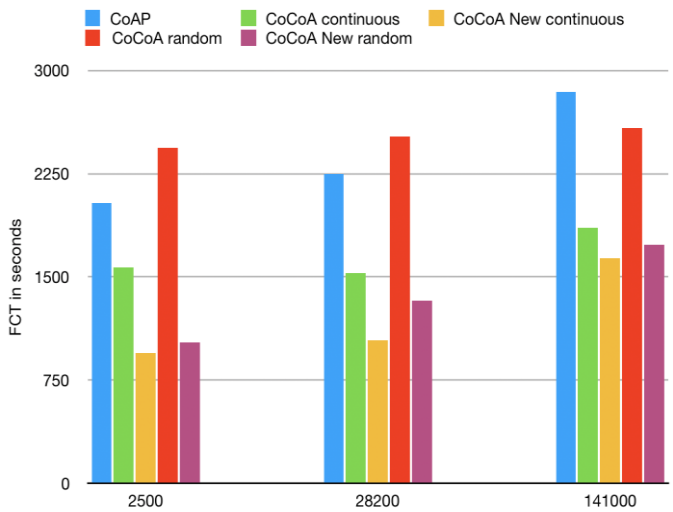

Figure 28 FCT using 150 bytes payload

On the other hand, it is possible to observe the modified CoCoA performing better, significantly reducing the FCT. Same conclusions as in the previous case could be taken from observing the RTTs and the number of retransmissions.

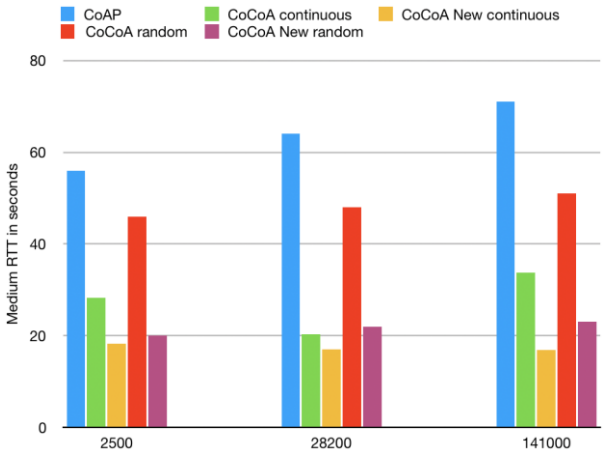

Figure 29 RTT using 150 bytes payload

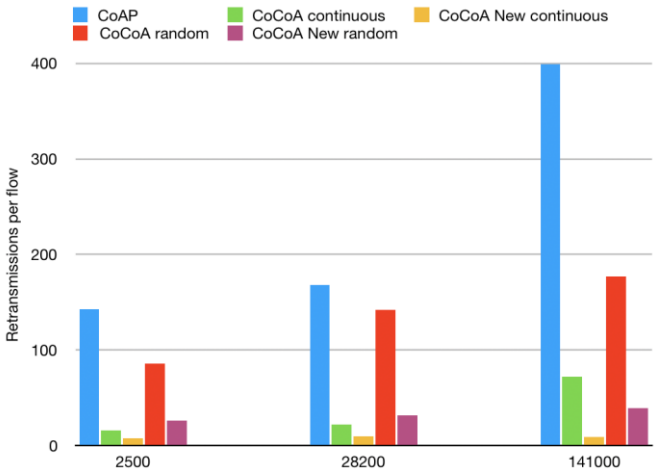

Figure 30 Retransmission per flow using 150 bytes payload

This time, the modified CoCoA also helps to reduce the number of retransmissions. As long as the RTT is estimated better, less retransmissions are performed, helping the network to decongest and improving the FCT.

4.1.3. Payload 296 bytes, no loss

The last case studied with this network configuration is the case of an even larger payload, 296 bytes. These tests are run in order to show how the difference of the performance between CoCoA and modified CoCoA grows with the payload size.

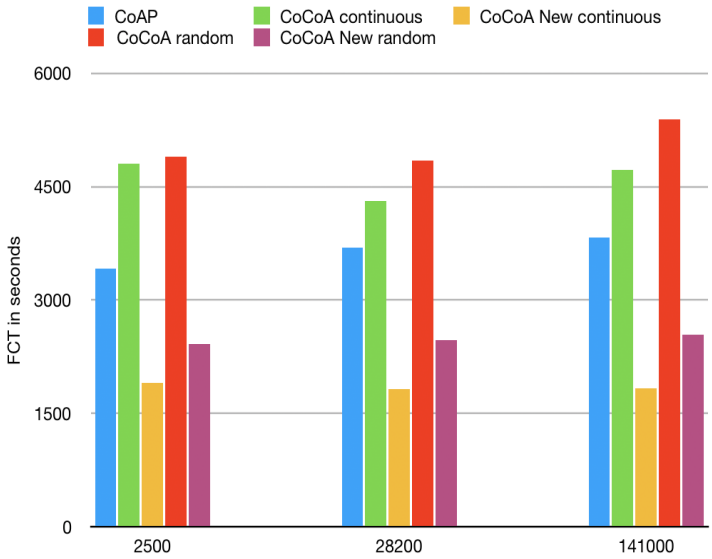

Figure 31 FCT using 296 bytes payload

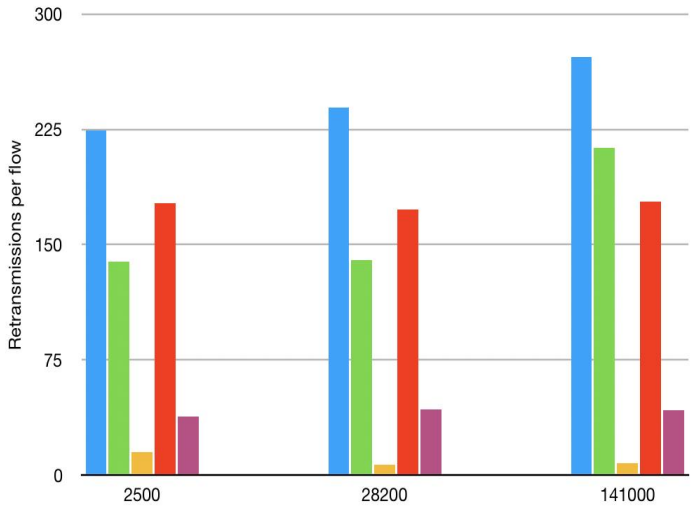

Figure 32 Retransmissions per flow using 296 bytes payload

4.2. Four hosts network configuration

In this section, the results of the network configuration using four hosts are shown and analyzed. It is also introduced the evaluation of unnecessary retransmissions. These retransmissions are those retransmissions, that are not necessary, because an ACK of a previous packet arrives after the retransmission is produced. In other words, if the retransmission is not performed, the ACK for this exchange, will also arrives.

4.2.1. Payload 68 bytes, no loss

With 68-byte responses, it can be observed that CoCoA does not suffer bufferbloat and it is stable independent of buffer size. On the other hand, CoAP performs worse when the buffer size increases.

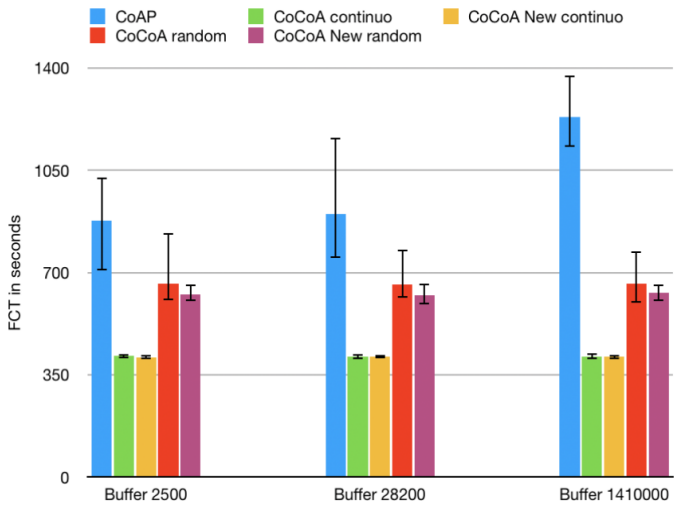

Figure 33 FCT for 68 bytes and no loss

Same pattern as in the previous configuration is reflected in these tests. In order to understand even better this behavior, in this section, the retransmissions performed by each client are compared with the unnecessary retransmissions performed by each client.

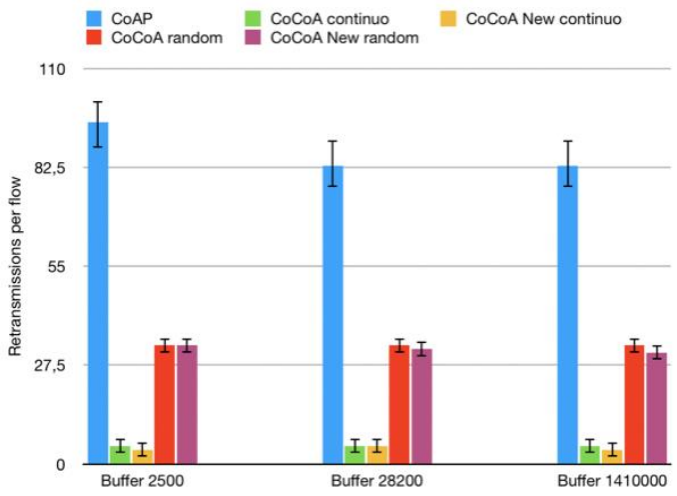

Figure 34 Retransmissions per flow 68 bytes and no loss

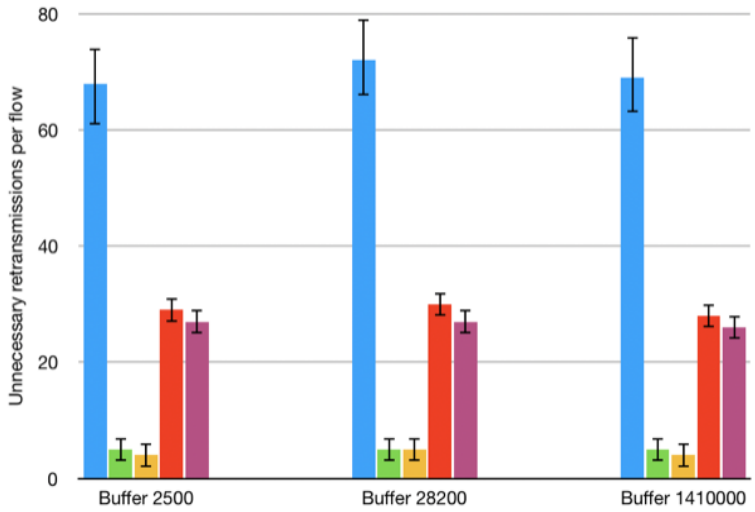

Figure 35 Unnecessary retransmissions per flow 68 bytes no loss

Comparing the retransmissions per flow with the unnecessary retransmissions, it can be observed that CoCoA and modified CoCoA perform very few retransmissions, but almost all are unnecessary. This is because no packets are lost and the retransmissions are due to the jitter on the buffers. However, both CoCoA versions adapt quickly and lead to a low number of retransmissions. On the other hand, CoAP performs more retransmissions than unnecessary retransmissions. CoAP does not adapt the RTO taking into account the RTTs, so the buffers are filled up with retransmissions and losing packets when the buffer is full.

4.2.2. Payload 68 bytes, with loss

The modifications introduced on CoCoA take into account several retransmissions in order to estimate the RTT. In large buffer cases, it helps to set larger RTOs, which match the packets behavior. On the other hand, if the retransmissions are needed because of losses, the modifications could introduce unnecessary large RTOs. Because of this reason the tests are reproduced introducing two loss models. The results of both cases are presented below running 200 flows in order to avoid too large simulation times.

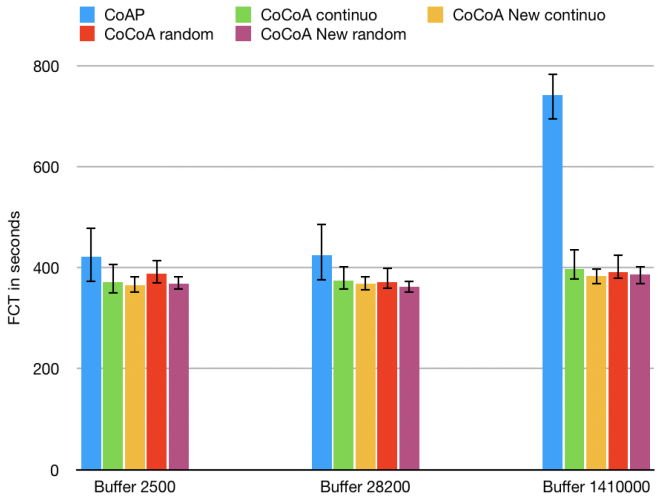

Figure 36 FCT payload 68 and medium loss

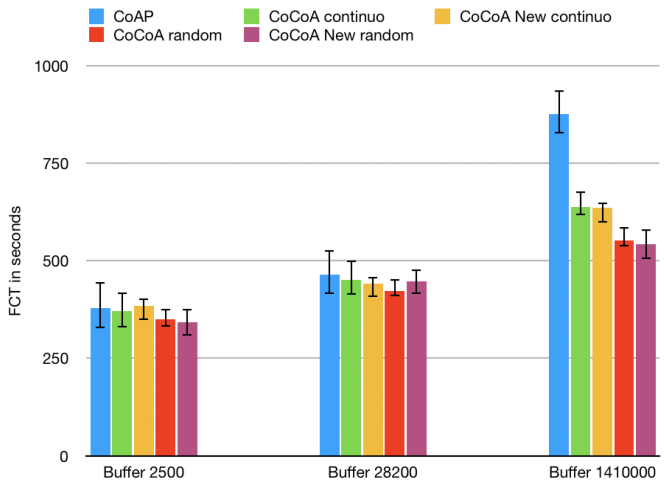

Figure 37 FCT payload 68 and high loss

As it can be observed on *Figure 36*, for the medium loss case the performance does not differs too much between CoCoA and modified CoCoA. On the other hand, CoAP suffers for the larger buffer size. In the high loss case, continuous flows perform worse than random flows. This behavior appears because, in continuous flows, it takes more RTTs

during the fifty message exchanges, in other words, the RTO has the opportunity to become larger.

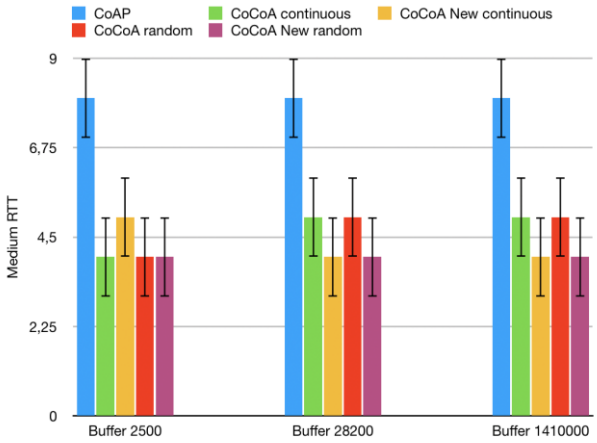

Figure 38 RTT 68 bytes medium loss

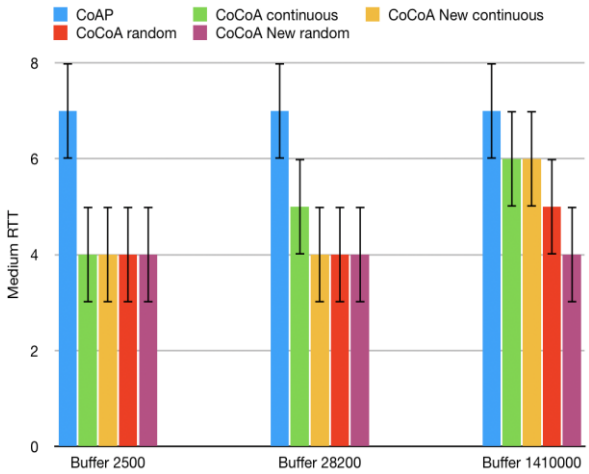

Figure 39 RTT 68 bytes and high loss

The fact of taking into account large RTTs increases unnecessarily the RTO, but as long as the time elapsed between retransmissions is larger, the buffer has time to empty and no delay, because of the buffer storage, is produced. Furthermore, less retransmissions are performed, because setting an unnecessary large RTO allows the packets that are not lost to receive the corresponding ACK before being retransmitted.

As it can be observed in Figures 41 and 43, the number of unnecessary retransmissions decreases in case of high loss because the retransmissions are due to losses and not due to low RTOs.

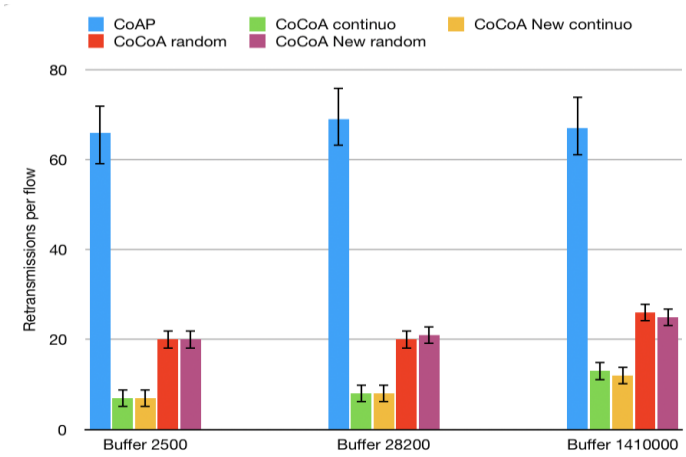

Figure 40 Retransmission 68 bytes and medium loss

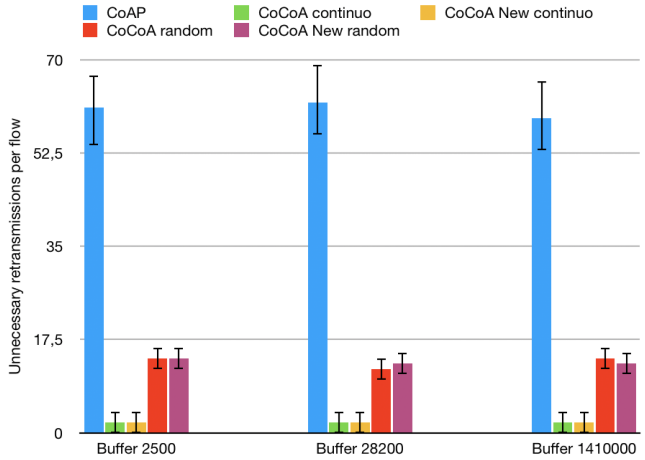

Figure 41 Unnecessary retransmissions 68 bytes and medium loss

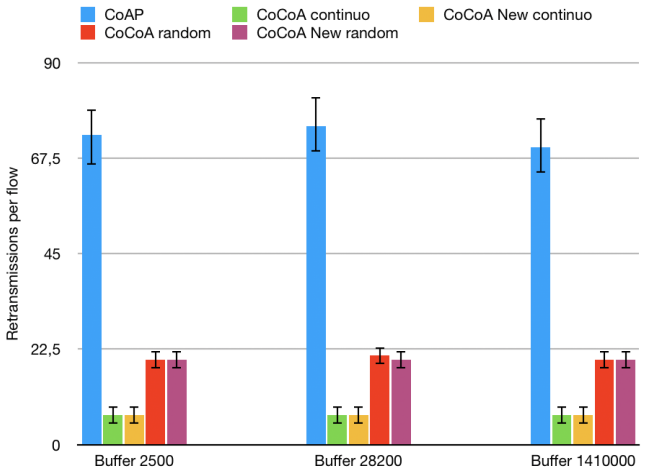

Figure 42 Retransmissions 68 bytes and high loss

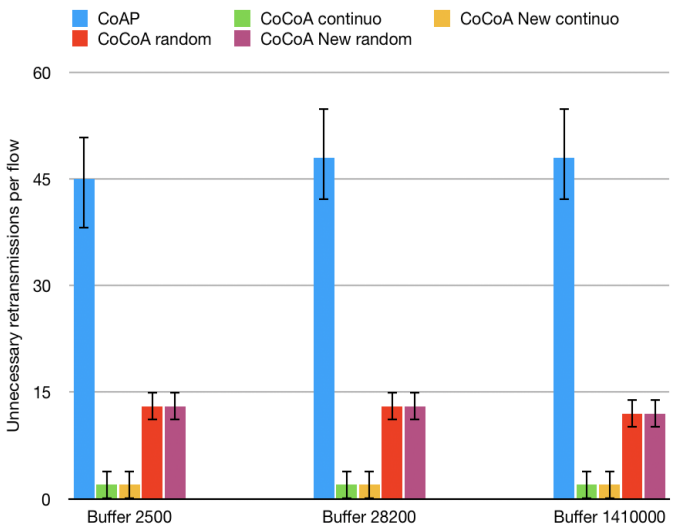

Figure 43 Unnecessary retransmissions 68 bytes and high loss

In these cases, the losses do not affect the performance of CoCoA and modified CoCoA too much, nevertheless, it has to be understood that this behavior could change in scenarios with more losses. Losses cause retransmissions and retransmissions affects directly the RTT estimation. These RTTs would be estimated from the first packet sent until the first ACK arrived. Hence, more retransmissions due to losses delay the arrival of the first ACK and makes the RTT estimated larger.

4.2.3. Payload 176 bytes, no loss

In these tests, the payload has been increased to 176 bytes in order to show that the buffer bloat appears also on this network configuration.

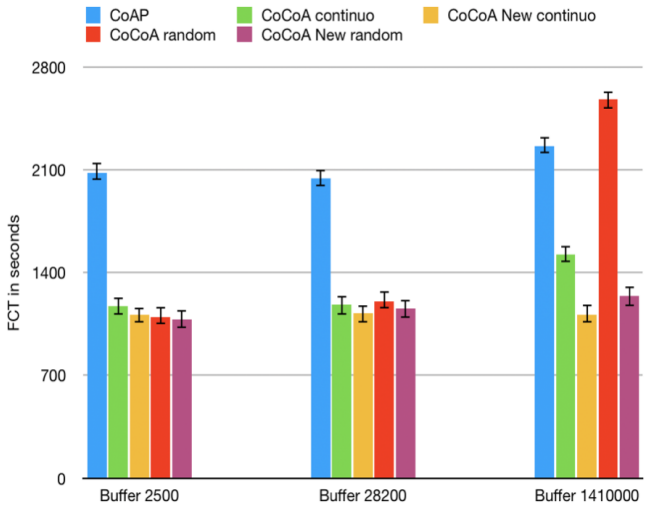

Figure 44 FCT 176 bytes and no loss

In this scenario, bufferbloat appears degrading CoCoA's performance obtaining even worse results than CoAP. However, the modified CoCoA achieves the same performance regardless of the buffer size.

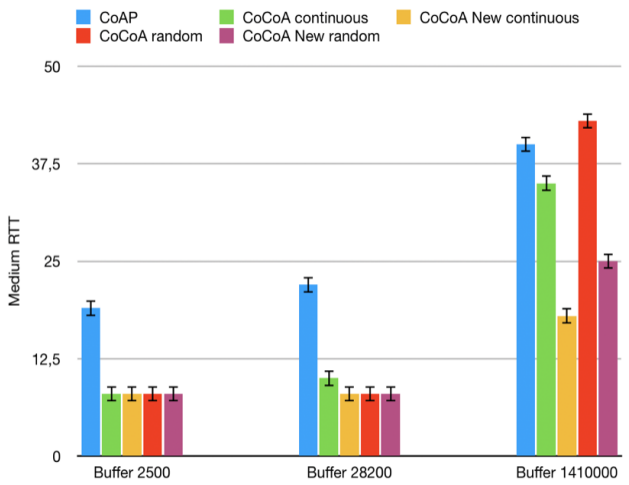

Figure 45 RTT 176 bytes and no loss

The average RTT graphic (Figure 45) shows the buffer bloat effect increasing the average RTT on CoCoA and CoAP cases. In the case of modified CoCoA, the RTT measurements also grows due to the estimation of the RTT based on more retransmissions, but not enough to cause a significant delay on the FCT. In addition, larger RTT on modified CoCoA

leads to less retransmissions in absence of loss. Hence, the delay, introduced because of unnecessary retransmissions filling up the buffer, decreases.

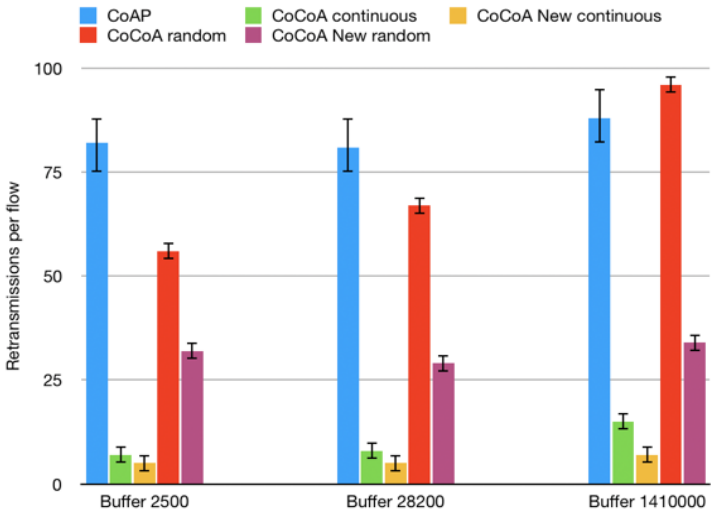

Figure 47 Retransmissions 176 bytes and no loss

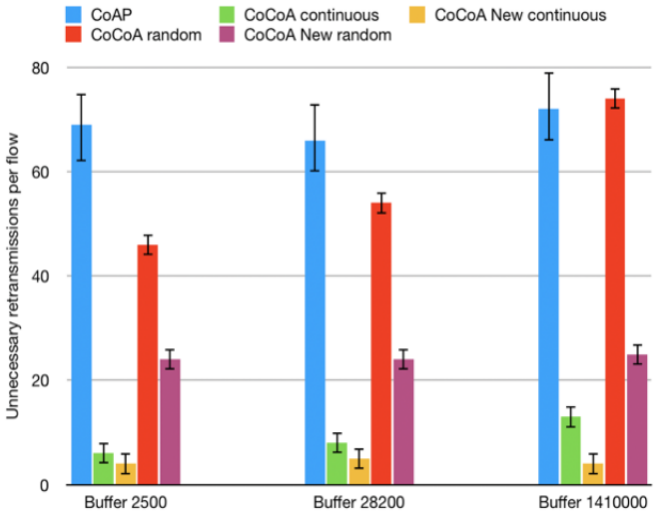

Figure 46 Unnecessary retransmissions 176 bytes and no loss

As it can be observed, the number of retransmissions always decreases in modified CoCoA cases. The number of unnecessary retransmissions for modified CoCoA almost matches the number of retransmissions because these retransmissions occur due to the jitter. However, in CoAP and CoCoA, retransmissions also appears because of the full buffers.

4.2.4. Payload 176 bytes, with loss

Losses are also tested in this scenario, again, taking into account two loss models and 200 flows.

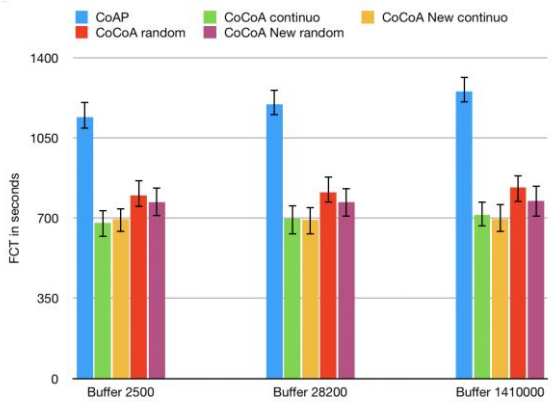

Figure 48 FCT 176 bytes and medium loss

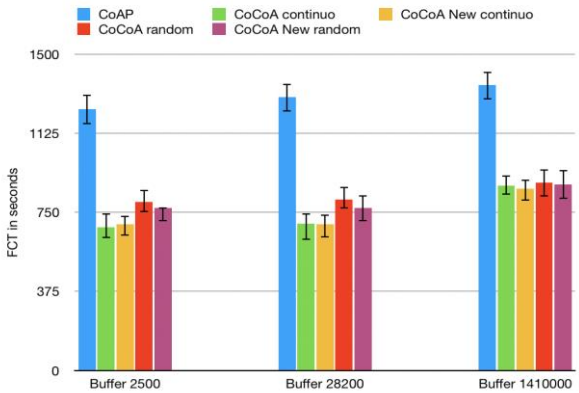

Figure 49 FCT 176 bytes and high loss

FCT increases with losses in all cases. As long as more retransmissions are performed because of losses, the RTTs estimated become greater, affecting the FCT. However, bufferbloat is not observed because the larger RTOs allow emptying the buffers.

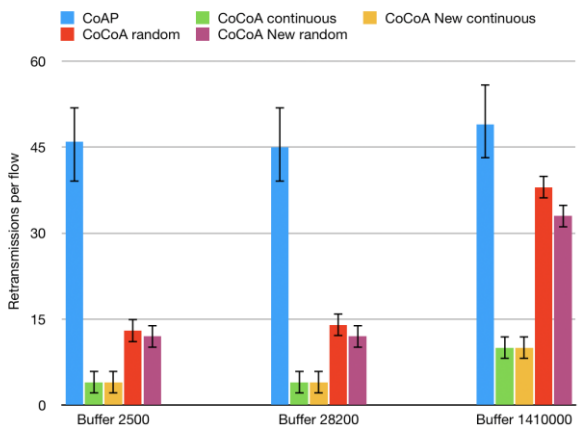

Figure 50 Retransmissions 176 byte and medium loss

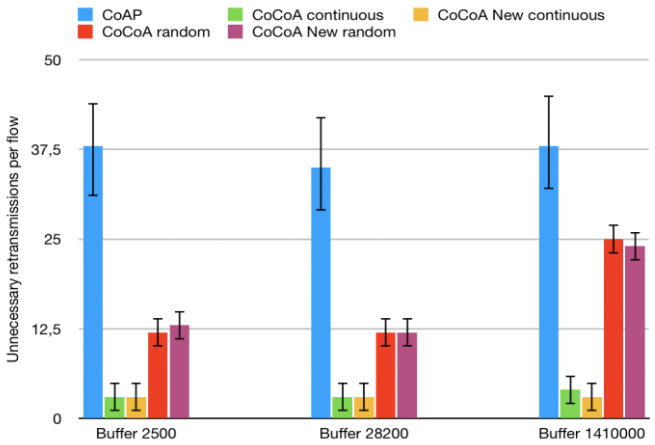

Figure 53 Unnecessary retransmissions 176 bytes and medium loss

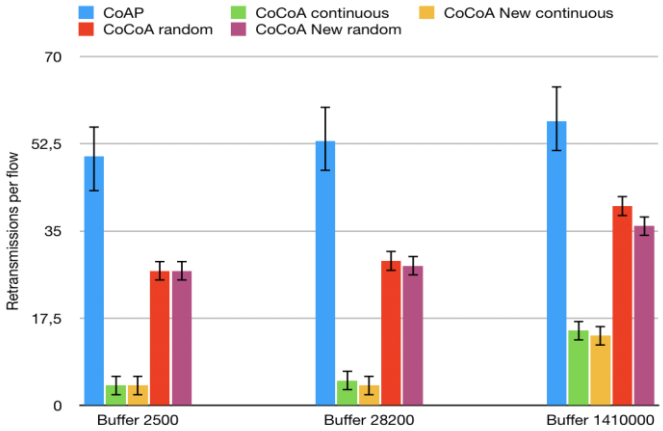

Figure 52 Retransmission 176 bytes and high loss

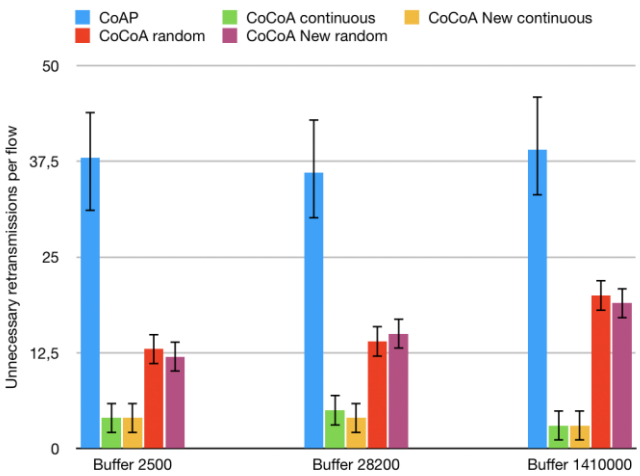

Figure 51 Unnecessary retransmissions 176 bytes and high loss

The number of unnecessary retransmissions decreases when losses increase in CoCoA cases. Because of this reason, the buffers are less congested and bufferbloat does not appear.

4.2.5. Estimating RTT for each retransmission

Finally, another modification for original CoCoA is tested. This modification consists of estimating the RTT for retransmitted packets, taking into account the time elapsed between the ACK arrival and the specific retransmitted packet transmission, instead of computing the RTT from the ACK arrival to the first packet sent. The purpose of this modification is to show how accurate RTT can reduce the FCT in lossy environments.

In order to test this hypothesis, each retransmitted packet sets the sending timestamp on its payload and then the server echoes this payload on the ACK. The client estimates the RTT as the time when the ACK arrives minus the time on the ACK payload. As long as the estimated RTT take into account the time when the retransmitted packet is sent, all RTTs are marked as strong RTTs.

This approach is experimental and only implemented to test the benefits of estimating more accurate RTTs. A useful implementation is described on future developments section.

The network links remains equal, but the loss model is changed to add more losses. In addition, two cases are tested, in order to add bufferbloat to the lossy scenario and test the worst case. So, the differences from the previous tests are:

- Request payload: 79 Bytes.
- Response payload: 69 Bytes.
- All RTTs marked as strong.
- Loss: 30%.
- Downlink rate: 30 Kbits for only loss scenario.
- Downlink rate: 10 Kbits for loss and bufferbloat scenario.

As it can be observed in *Figure 54*, the fact of adding a more accurate estimation of the RTTs, achieves a better performance in case of lossy scenario. On the other hand, for loss and bufferbloat the FCT is very similar.

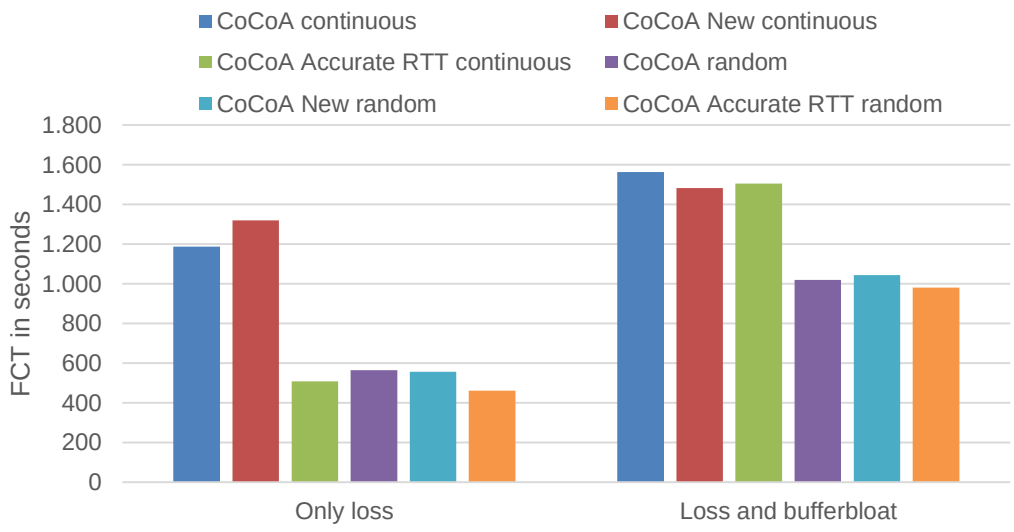

Figure 54 Comparison of CoCoA FCT using different modifications

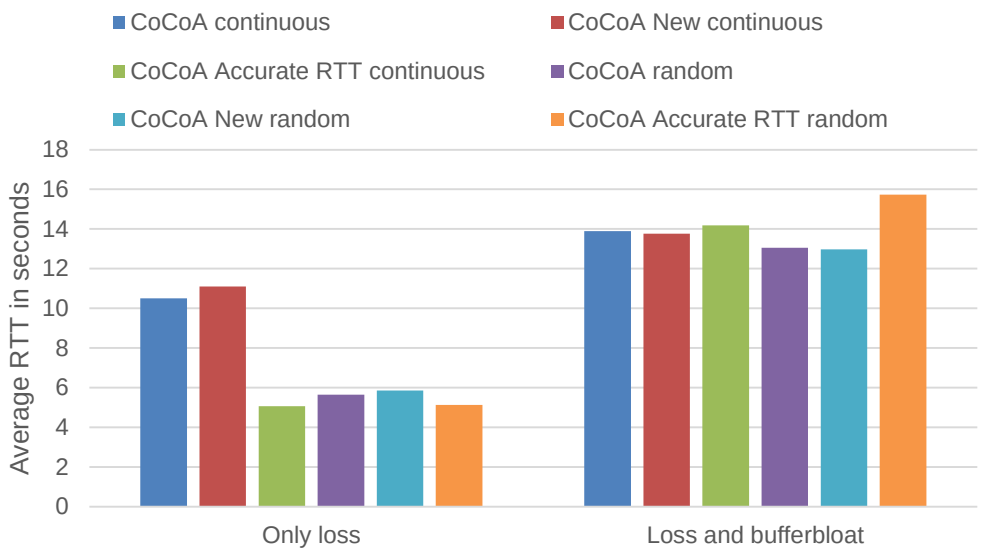

Figure 55 Comparison of CoCoA Average RTT using different modifications

On the only loss scenario, the RTTs estimated varies because of retransmissions due to loss, while in case of loss and bufferbloat, the RTTs varies also because of the different delays produced in the buffers. The drawback of estimating RTTs more accurately is that the retransmissions are performed faster due to low RTOs. Particularly, in case of bufferbloat, the first RTT estimated is lower than the second, and the second lower than the third, and so on, until the buffer is full. This cause a large number of retransmissions, and a large number of unnecessary retransmissions, in the case of CoCoA with accurate RTTs estimation.

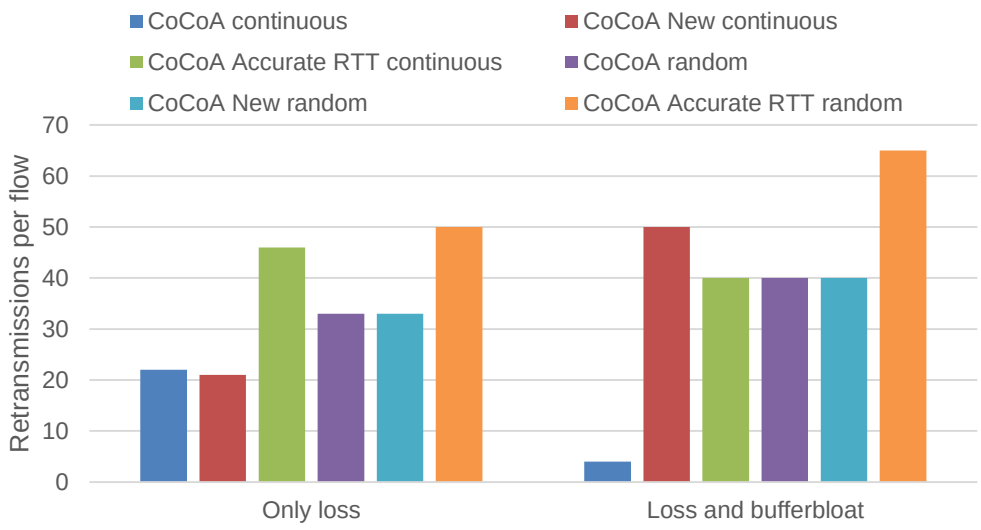

Figure 56 Comparison of CoCoA retransmissions per flow using different modifications

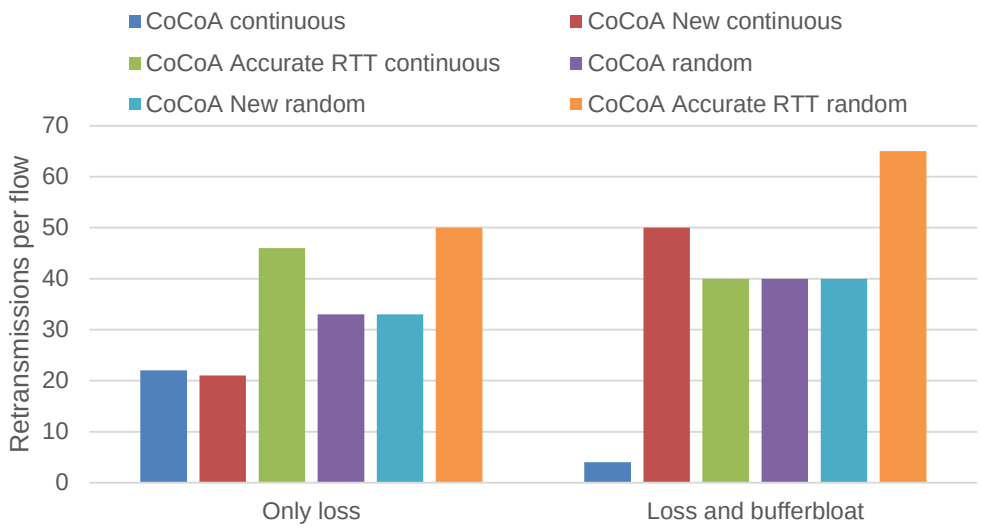

Figure 57 Comparison of CoCoA unnecessary retransmissions per flow using different modifications

In summary, the results show that taking into account more retransmissions for weak estimators, helps in bufferbloat scenarios, and obtaining more strong estimators taking into account more accurate RTTs, helps in lossy scenarios but increase significantly the number of retransmissions.

5. Budget

As long as this thesis does not create any prototype or product, and it is thought to introduced minimal changes to improve a congestion control implemented on open source Java framework, the only work required to implement and use it is a minimal change on the code by the author and update of the framework for the users. So for the author the costs will be zero and for the users the cost of updating their devices will vary depending on to the activity they perform. Because of these reasons, the cost for already used devices depends on the final purpose of this use but the cost for new devices implementing these improvements on new devices is almost zero.

As long as the cost is almost zero in time and money in most cases, it is financially viable. Furthermore, as long as these improvements reduce the number of retransmissions, it offers an improvement in terms of energy consumption. Improving the energetic efficiency of the process also means saving for the final users.

6. Environment Impact

The environment impact of this thesis depends on the devices which are going to implement the modifications proposed. As long as these modifications reduce the number of retransmissions, the energy consumed by the IoT devices which invest the most part of this energy consumption on transmission, would be significantly reduced which is always good for the environment. On the other hand, in lossy networks, reducing the number of retransmissions could lead to a greater active time for some devices. If the devices energy consumption is big enough in active mode, could suppose a greater energy consume.

7. Conclusions and future development

7.1. Conclusions

First conclusions can be taken directly by observing the results. In case of bufferbloat the new modifications on weak estimators provide CoCoA with a better performance reducing the flow completion time and reducing the number of retransmissions and, above all, reducing the number of unnecessary retransmissions. These modifications also perform significantly better than the default CoAP.

On the other hand, in case of losses, if a packet is lost and retransmitted several times, the way that Weak RTT is computed leads to a bad RTT estimation introducing larger RTOs. This fact could translate to an unnecessary retransmission wait time and a greater active time for each flow. However, this phenomenon does not appear in the specific tested scenarios. The hypothesis of why it does not occur is that the losses and the large RTOs helps the buffers to be less congested and the RTOs are not larger enough to observe a significant delay.

As long as it is a theoretical probability that this phenomenon occurs, two more variations for CoCoA are proposed taking into account the results of the accurate RTT estimation. Additionally, as long as the results of FASOR's paper differ from those tested on this thesis and FASOR is not implemented on Californium, the modifications are only compared against CoAP and CoCoA. So, another future development would be developing FASOR on Californium and compare the modifications presented on this thesis with other congestion control mechanisms.

7.2. Future developments

In order to avoid the principal drawback of weak estimators introduce to CoCoA, the congestion control mechanism should be able to understand if its retransmissions are due to congestion or losses and act accordingly in each case. The difference between these two cases are from which packet does the ACK correspond. So the intuitive solution should be adding an option to the CoAP headers where the number of the retransmission is indicated and echoed in the ACK. As long as CoAP tries to have the shortest possible header, instead of an option with the number of retransmissions it would be an option indicating if it is retransmitted or not.

7.2.1. Number of retransmissions option

The principle of this proposed modification is simple, the retransmitted packet has the number of the retransmissions on the header and the server echoes it in the ACK. This way the ACK could be associated directly to the packet it corresponds to.

In case of the ACK corresponding to the original packet (retransmission 0) and several retransmissions have been performed, the transmitter can assume that is in a congestion scenario. Otherwise, if the ACK corresponds to a retransmitted packet (retransmission > 0) the transmitter can assume that is in a lossy scenario. Moreover, the sender can estimate the RTT from the retransmission to the ACK, instead of, from the original packet to the ACK, estimating better the RTT.

Once the sender understands that is in a congested but not lossy scenario, it can decide to set the RTO to the maximum RTT computed until that point. This way, it ensures not to send unnecessary retransmissions for possible delay times.

On the other hand, if the sender decides that it is in a lossy scenario, the RTTs would be computed better so the retransmissions would be more adjusted in time. Another option is to consider if the amount of losses is high or low based on the number of retransmissions. In case of a high number of retransmissions due to loss, it could be considered to send some packets duplicated.

Two ways of estimating the RTT for retransmissions are proposed. The first one is to base the estimation on the ACK arrival time and the real retransmitted time. This one needs to store all the timestamps of each retransmission. The second one is to save the RTOs used to the retransmissions and estimate the RTT as the time from the original packet minus the sum of the RTOs used until the corresponding retransmission.

In order to compute the RTO, the results show that if the RTT is accurate it would set short RTOs, rising the number of retransmissions, and in case of bufferbloat, rising significantly the number of unnecessary retransmissions. In case of large RTOs obtained by taking into account weak estimators, the number of retransmissions decrease but increase the FCT. Due to these facts, the optimal RTO must be computed finding a balance between these two approaches.

The first proposal to compute the RTO, consist in dynamically set weights to the RTTs depending on the reason of the retransmissions. If the network presents 30% of retransmissions due to loss and 70% of retransmissions due to congestion, the weights that multiply the RTT should increase and take values over 1, in order to allow buffers to empty and to not perform unnecessary retransmissions. In the other hand, if there is a 70% due to loss and 30% due to congestion, probably the weights should tend to 1. The weights will be dynamically updated in order to obtain the best performance.

The second proposal, is to save the last n RTTs in order to understand if the congestion is growing or decreasing and adjust the weight that multiply the RTT. In example, if the first RTT is lower than the second, and the second is lower than the third it could be understood like the congestion is growing and increase the weights. In the other hand, if the third is lower than the second, the second lower than the first the weights would decrease. The more RTTs stored the more capacity to see the general trend to adjust the weights better.

7.2.2. Retransmitted state option

In order to make the header lighter, instead of adding the number of retransmissions to the header, it is enough to set a flag to 0 for non-retransmitted packets and 1 to those which are retransmitted.

The idea is the same as in the case of the number of retransmissions option but, as long as it is not possible to know exactly from which packet the ACK arrives, the RTT for the lossy case would be just the time from the original packet to the ACK minus the first RTO used. This would lead to deciding the RTT as *min(estimated RTT - RTO, Strong RTT)* for the first retransmission and *max(estimated RTT-RTO, Strong RTT)* for the following retransmissions.

8. Bibliography

[1] Ericsson, *"Internet of Thing Forecast"*. 2015 [Online] https://www.ericsson.com/en/mobility-report/internet-of-things-forecast.

[2] Ilpo Jarvinen, Iivo Raitahila, Markku Kojo from University of Helsinki and Zhen Cao from Huawei, *"FASOR Retransmission Timeout and Congestion Control Mechanism for CoAP"*. 2018 [Online] https://helda.helsinki.fi//bitstream/handle/10138/300035/paper.pdf?sequence=1.

[3] Knud Lasse, *"State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating".* August 2018 [Online] https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/.

[4] Constrained Application Protocol*, (CoAP) Specification*. IETF RFC 7252, June 2014.

[5] August Betzler from I2Cat, Carles Gomez, Ilker Demirkol, Josep Paradells from Universitat Politécnica de Catalunya, *"CoCoA+: An advanced congestion control mechanism for CoAP".* June 2015 [Online] https://www.sciencedirect.com/science/article/abs/pii/S1570870515000888.

[6] August Betzler from I2Cat, Carles Gomez, Ilker Demirkol, Josep Paradells from Universitat Politécnica de Catalunya, *"CoAP Congestion Control for the Internet of Things".* July 2016 [Online] https://core.ac.uk/download/pdf/81580479.pdf.

[7] C. Bormann Universitaet Bremen TZI, A. Betzler Fundacio i2CAT, C. Gomez and I. Demirkol Universitat Politecnica de Catalunya/Fundacio i2CAT. *"CoAP Simple Congestion Control/Advanced draft-ietf-core-cocoa-03".* August 2018 [Online] https://www.ietf.org/archive/id/draft-ietf-core-cocoa-03.txt.

[8] Vishal Rathod, Natasha Jeppu, Samanvita Sastri, Shruti Singala and Mohit P. Tahiliani, *"CoCoA++: Delay gradient based congestion control for Internet of Things"* April 2019 [Online] https://www-sciencedirect-com.recursos.biblioteca.upc.edu/science/article/pii/S0167739X18308677.

[9] Jung June Lee , Sung Min Chung , Byungjun Lee , Kyung Tae Kim , Hee Yong Youn*, "Round Trip Time Based Adaptive Congestion Control with CoAP for Sensor Network".* 2016 [Online] https://ieeexplore-ieee-org.recursos.biblioteca.upc.edu/abstract/document/7536324.

[10] Simone Bolettieri, Giacomo Tanganielli, Carlo Vallati, Enzo Mingozzi, "pCoCoA: A precise congestion control algorithm for CoAP". June 2018 [Online] https://www-sciencedirect-com.recursos.biblioteca.upc.edu/science/article/pii/S1570870518303834.

[11] Netem manual. November 2011 [Online] https://www.linux.org/docs/man8/tc-netem.html.

[12] Californium (Cf) framework. [Online] https://www.eclipse.org/californium/.

[13] LibCoAP framework [Online] https://libcoap.net/.

9. Glossary

ACK: Acknowledgement

API: application programming interface

CDG: delay-gradient congestion control / CAIA Delay-Gradient

CoAP: Constrained Application Protocol

CoCoA: Congestion Control/Advanced

CON: Confirmable message

CoRE: Constrained RESTful Environments

FASOR: Fast-slow RTO

FCT: Flow Completion Time

FTP: File Transfer Protocol

HTTP: Hypertext Transfer Protocol

IETF: Internet Engineering Task Force

IoT: Internet of Things

NON: Non confirmable messages

RAM: Random Access Memory

RFC: Request for Comments

ROM: read-only memory

RST: Reset

RTO: Retransmission timeout

RTT: round-trip time

RTTVAR: Round Trip Time Variance

SRTT: Smoothed Round Trip Time

TCP: Transmission Control Protocol

UDP: User Datagram Protocol

URI: Uniform Resource Identifier

10. Appendix

This appendix contains the code used for run CoAP clients and CoAP Server.

- **Continuous Clients:**

```
import java.util.Date;
import java.sql.Timestamp;
import java.util.concurrent.Semaphore;
import java.util.logging.Level;

import org.eclipse.californium.core.CaliforniumLogger;
import org.eclipse.californium.core.CoapClient;
import org.eclipse.californium.core.CoapHandler;
import org.eclipse.californium.core.CoapResponse;
import org.eclipse.californium.core.network.CoapEndpoint;
import org.eclipse.californium.core.network.EndpointManager;
import org.eclipse.californium.core.network.config.NetworkConfig;
import org.eclipse.californium.core.network.stack.congestioncontrol.Cocoa;

public class cocoa400clients {
	/*static {
		CaliforniumLogger.initialize();
		CaliforniumLogger.setLevel(Level.CONFIG);
	}*/
	public static Timestamp getCurrentTime() {
		Date date= new Date();
		long time = date.getTime();

		return new Timestamp(time);
	}
	public static void main(String[] args) {
		NetworkConfig config = new NetworkConfig()
		// enable congestion control (can also be done via Californium.properties)
		//.setBoolean(NetworkConfig.Keys.USE_CONGESTION_CONTROL, true)
		// see class names in org.eclipse.californium.core.network.stack.congestioncontrol
		//.setString(NetworkConfig.Keys.CONGESTION_CONTROL_ALGORITHM,
Cocoa.class.getSimpleName())
		//set MAX RETRANSMISIONS TO 20
```

```
                .setInt(NetworkConfig.Keys.MAX_RETRANSMIT, 20)
                // set NSTART to 1
                .setInt(NetworkConfig.Keys.NSTART, 1);


        Runnable runnable = () -> {
            int port = (int)Thread.currentThread().getId();
            CoapEndpoint cocoaEndpoint = new CoapEndpoint(2000 + port, config);
            CoapClient client = new
CoapClient("coap://10.0.0.4/helloWorld").setEndpoint(cocoaEndpoint);
            final int NUMBER = 50;
            final Semaphore semaphore = new Semaphore(0);
            for (int i=0; i<NUMBER; ++i) {
                client.get(new CoapHandler() {
                        @Override
                        public void onLoad(CoapResponse response) {
                            semaphore.release();
                            }
                        }
                        @Override
                        public void onError() {
                            System.out.println("Failed");
                            semaphore.release();
                        }
                    });

                }

                // wait until all requests finished
                try {
                    semaphore.acquire(NUMBER);
                } catch (InterruptedException e) {}
                System.out.println("Thread Finish Time: " + getCurrentTime());
            };

            for(int i=0; i<400; i++){
                //System.out.println("Creating Thread for a Client...");
                Thread thread = new Thread(runnable);
```

```
                        //System.out.println("Starting a Client Thread...");
                        thread.start();
                    }
                }
            }
```

- **Random Clients:**

```
import java.sql.Timestamp;
import java.util.Date;
import java.util.concurrent.Semaphore;
import java.util.logging.Level;
import org.eclipse.californium.core.CaliforniumLogger;
import org.eclipse.californium.core.CoapClient;
import org.eclipse.californium.core.CoapHandler;
import org.eclipse.californium.core.CoapResponse;
import org.eclipse.californium.core.network.CoapEndpoint;
import org.eclipse.californium.core.network.config.NetworkConfig;
import org.eclipse.californium.core.network.stack.congestioncontrol.Cocoa;

public class cocoaBurst {
	/*static {
		CaliforniumLogger.initialize();
		CaliforniumLogger.setLevel(Level.CONFIG);
	}*/
	static int numberOfBurstClients = 400;
	public static Timestamp getCurrentTime() {
		Date date= new Date();
		long time = date.getTime();
		return new Timestamp(time);
	}

	public static void main(String[] args) {
		NetworkConfig config = new NetworkConfig()
		// enable congestion control (can also be done cia Californium.properties)
		.setBoolean(NetworkConfig.Keys.USE_CONGESTION_CONTROL, true)
		// see class names in org.eclipse.californium.core.network.stack.congestioncontrol
		.setString(NetworkConfig.Keys.CONGESTION_CONTROL_ALGORITHM,
Cocoa.class.getSimpleName())
		//set MAX RETRANSMISIONS TO 20
		.setInt(NetworkConfig.Keys.MAX_RETRANSMIT, 20)
		//set EXCHANGE_LIFETIME TO 247
		//.setInt(NetworkConfig.Keys.EXCHANGE_LIFETIME, 247 * 1000)
		//set EXCHANGE_LIFETIME TO 247
```

```
                //.setInt(NetworkConfig.Keys.MAX_TRANSMIT_WAIT, 93 * 1000)
                // set NSTART to four
                .setInt(NetworkConfig.Keys.NSTART, 1);

                Runnable runnable = () -> {
                        int port = (int)Thread.currentThread().getId();
                        //CoapEndpoint cocoaEndpoint = new CoapEndpoint(2000 + port, config);
                        int currentRequests=0;
                        //System.out.println("Start Time: " + getCurrentTime());
                        CoapEndpoint cocoaEndpoint;
                        while (currentRequests < 50) {
                                cocoaEndpoint = new CoapEndpoint(2000 + port, config);
                                CoapClient client = new
CoapClient("coap://10.0.0.4/helloWorld").setEndpoint(cocoaEndpoint);
                                final Semaphore semaphore = new Semaphore(0);
                                int numRequests = (int)(Math.random() * (Math.min(9, 49 -
currentRequests))) + 1;

                                for (int i=0; i<numRequests; ++i) {
                                        client.get(new CoapHandler() {
                                                @Override
                                                public void onLoad(CoapResponse response) {
                                                        semaphore.release();
                                                }

                                                @Override
                                                public void onError() {
                                                        System.out.println("Failed");
                                                        semaphore.release();
                                                }
                                        });
                                }

                                try {
                                        semaphore.acquire(numRequests);
                                        //System.out.println("Requests : " + numRequests);
```

```
                        } catch (InterruptedException e) {
                                System.out.println("Smaphore failed");
                        }
                        cocoaEndpoint.destroy();
                        currentRequests += numRequests;
                }
                System.out.println("Thread Finish Time: " + getCurrentTime());
        };
        System.out.println("Start Time: " + getCurrentTime());
        for(int currentClient = 0; currentClient < numberOfBurstClients; currentClient++ )      {
                Thread thread = new Thread(runnable);
                //thread.setName("Thread " + currentClient);
                thread.start();
        }
    }
}
```

- **Server:** The default hello world server from Californium has been used, only changing the payload size for the response on different tests.

```
import java.net.Inet4Address;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.SocketException;

import org.eclipse.californium.core.CoapResource;
import org.eclipse.californium.core.CoapServer;
import org.eclipse.californium.core.network.CoapEndpoint;
import org.eclipse.californium.core.network.EndpointManager;
import org.eclipse.californium.core.network.config.NetworkConfig;
import org.eclipse.californium.core.server.resources.CoapExchange;

public class HelloWorldServer extends CoapServer {
        private static final int COAP_PORT =
NetworkConfig.getStandard().getInt(NetworkConfig.Keys.COAP_PORT);
  /*    Application entry point. */
  public static void main(String[] args) {
    try {
      // create server
      HelloWorldServer server = new HelloWorldServer();
      // add endpoints on all IP addresses
      server.addEndpoints();
      server.start();
    } catch (SocketException e) {
      System.err.println("Failed to initialize server: " + e.getMessage());
    }
  }
  /*Add individual endpoints listening on default CoAP port on all IPv4 addresses of all network interfaces.*/
  private void addEndpoints() {
      for (InetAddress addr : EndpointManager.getEndpointManager().getNetworkInterfaces()) {
          // only binds to IPv4 addresses and localhost
              if (addr instanceof Inet4Address || addr.isLoopbackAddress()) {
                  InetSocketAddress bindToAddress = new InetSocketAddress(addr,
COAP_PORT);
                  addEndpoint(new CoapEndpoint(bindToAddress));
```

```
                        }
                    }
    }

    /*Constructor for a new Hello-World server. Here, the resources of the server are initialized. */
    public HelloWorldServer() throws SocketException {
        // provide an instance of a Hello-World resource
        add(new HelloWorldResource());
    }
    /* Definition of the Hello-World Resource*/
    class HelloWorldResource extends CoapResource {
        public HelloWorldResource() {
            // set resource identifier
            super("helloWorld");
            // set display name
            getAttributes().setTitle("Hello-World Resource");
        }
        @Override
        public void handleGET(CoapExchange exchange) {
            // respond to the request
            exchange.respond("Hello World!");
        }
    }
}
```