ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ
ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

ΣΧΟΛΗ
ΕΠΙΣΤΗΜΩΝ &
ΤΕΧΝΟΛΟΓΙΑΣ
ΤΗΣ
ΠΛΗΡΟΦΟΡΙΑΣ
SCHOOL OF
INFORMATION
SCIENCES &
TECHNOLOGY
ΜΕΤΑΠΤΥΧΙΑΚΟ
ΕΠΙΣΤΗΜΗ ΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
MSc IN COMPUTER SCIENCE

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΜΣ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία
Μεταπτυχιακού Διπλώματος Ειδίκευσης

«Simulation of the RACH Procedure to Investigate Flow Control Mechanisms in the Application Layer»

ΓΙΩΡΓΟΣ ΠΑΝΤΕΛΗΣ

f3321910
Επιβλέπων: ΒΑΣΙΛΕΙΟΣ Α. ΣΥΡΗΣ

ΑΘΗΝΑ, ΝΟΕΜΒΡΙΟΣ 2020

Simulation of the RACH Procedure to Investigate Flow Control Mechanisms in the Application Layer

---

George Pantelis

November 2020

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Athens University of Economics and Business

ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

School of Information Sciences and Technology
Department of Informatics
Athens, Greece

Master Thesis
in
Computer Science

Simulation of the RACH Procedure to Investigate Flow Control Mechanisms in the Application Layer

George Pantelis

Committee: Associate Professor Vasilios A. Siris (Supervisor)

Professor George C. Polyzos

Professor George Xylomenos

November 2020

George Pantelis
Simulation of the RACH Procedure to Investigate Flow Control Mechanisms in the Application Layer
November 2020
Supervisor: Prof. Vasilios A. Siris

Athens University of Economics and Business
School of Information Sciences and Technology
Department of Informatics
Mobile Multimedia Laboratory
Athens, Greece

Abstract

In cellular networks, whenever a mobile phone or a sensor needs to connect to the wireless channel, the Random Access Channel Procedure (RACH) takes place. The first step in the RACH, to establish a connection with the Base Station, involves a random access process. As the number of devices is increasing, the number of collisions in the first step of RACH increases. In the Next Generation Internet, a large number of IoT sensors will be connected to the cellular network, leading to RACH's possible congestion. In this Thesis we developed a Simulator to investigate the performance of the RACH procedure, under different parameters, loads, data rates, and traffic mixes. Moreover, we examine two scenarios, where we seek to reduce the data flow rate at the application level, without requiring changes to the low layers.

Περίληψη

Το πρώτο βήμα σύνδεσης ενός κινητού ή ενός αισθητήρα στο σταθμό βάσης ενός δικτύου κινητής επικοινωνίας είναι το πρωτόκολλο RACH, το οποίο περιλαμβάνει μια παράμετρο, η οποία επιλέγεται τυχαία απο ένα σύνολο τιμών. Αν δυο κινητές συσκεύες επιλέξουν την ίδια τυχαία τιμή τα δεδομένα που στέλνουν θα συγκρουστούν. Στο μέλλον πολύ μεγάλος αριθμός συσκεύων του Διαδικτύου των Πραγμάτων (Internet of Things), μαζί με τα έξυπνα κινητά τηλέφωνα, θα συνδέονται στο ίδιο ασύρματο δίκτυο, με αποτέλεσμα να υπάρχει επιπλέον επιβάρυνση. Σκοπός αυτής της εργασίας είναι να μελετήσει πως επηρεάζεται το ασύρματο κανάλι, όταν μεταβάλλονται ο αριθμός των συσκευών, η περιόδος που παράγουν τις μετρήσεις τους, και άλλα, καθώς και να εξετάσει πιθανές λύσεις που δρούν σε επίπεδο εφαρμογής, χωρίς να αλλάζει κάτι στα χαμηλότερα επίπεδα. Στην αρχή, εξηγούμε πως λειτουργεί η RACH διαδικασία και στη συνέχεια αναφέρουμε δυο πιθανά σενάρια, τα οποία βασίζονται σε ιδέες που έχουν προταθεί στην βιβλιογραφία. Το πρώτο σενάριο εφαρμόζει μια λογική παρόμοια με τη λογική του αλγορίθμου του TCP για έλεγχο συμφόρησης και το δευτερο χωρίζει τις συσκεύες σε ομάδες με την κάθε μια να έχει έναν αρχηγό που αναλαμβάνει την αποστολή των μετρήσεων τους στο σταθμό βάσης. Η διερεύνηση της απόδοσης έγινε σε προσομοιωτή του πρωτοκόλλου RACH που υλοποιήθηκε στα πλαίσια της εργασίας.

Acknowledgement

I feel obligated to thank everyone who made this thesis achievable and supported me during my post-graduate studies. To begin with, I would like to give thanks to my supervisor, Associate Professor Vasilio A. Siri, whose valuable and concentrated assistance along with our excellent cooperation and communication, led to the completion of this thesis. To continue, I would especially thank Professor George C. Polyzos and Professor George Xylomenos for taking part in my thesis committee and for their time. Last but not least, I would like to recognize the help and the inconsistent support of my family and friends through this challenging period, as well as all of my university colleagues, who tolerated with me until the end.

Contents

1

Introduction

This thesis aims to analyze the Random Access Channel procedure (RACH) of LTE, determine an issue that may show up in the Next Generation Internet, and more specifically, in a Massive IoT environment and examine some possible solutions to overcome this problem.

1.1 Motivation and Problem Statement

In a few years, the number of sensors connected to the internet will be much larger than today. These sensors devices (IoT) will connect to the internet through the wireless channel. Smart traffic lights, temperature sensors, and more, have to send their data now and then to the respective server. As the sensors will share the same wireless network channel with mobile phones, they will use the same protocols, like LTE. A possible issue is located in the LTE's RACH procedure, a random access protocol to control the channel access requests that the end-devices generate. If we increase the number of requests, we respectively increase the probability of collision. So, as we mentioned before,the number of IoT devices is rising, creating a Massive IoT environment, which will probably cause an overload issue in the existing mobile network.

In the RACH procedure, while the end-devices are increasing, the probability of a collision increases respectively, as we will see later. Thus, it is essential to overcome this issue not by reducing the number of end-devices, but by decreasing the amount of data they need to send. Specifically, we examine how we can reduce a sensor's data flow in the application level without evolving the low layers. Moreover, we simulate the possible solutions, and we compare them. The comparison is implemented in a simulator that we have built from scratch, and it aims to emerge metrics crucial for the performance evaluation of a network, such as the number of collisions and the delay.

1.2 Thesis Structure

Chapter 2

In chapter 2, we present the theoretical approach, which is necessary to understand the source of the problem we aim to overcome. Specifically, we explain what the RACH procedure is, how it works, and why it is essential in the wireless networks, even at 5G. Also, we present some other related work briefly and introduce the technical characteristics of the simulator.

Chapter 3

Firstly, In chapter 3, we explain how the sensors send their data, at which rate and we denote the basic scenarios that we are going to implement. Then, we present the architecture of the simulator and we describe how we implement the RACH procedure and the scenarios that we denote in the beginning of chapter 3.

Chapter 4

In chapter 4, we present the experimental results. In particular, we explain what parameters we chose to tune and why, we demonstrate the results for every scenario that we denote in chapter 3 and we compare them.

Chapter 5

In chapter 5, we restate the possible weakness of the RACH procedure, we explain how we can overcome this, and we summarize the results from our experiments.

Background and Related Work

2

2.1 Background

2.1.1 Description of RACH procedure

The RACH procedure, as [Erion] describes, is a protocol of LTE for the communication of mobiles with the cell towers. Specifically, the main purpose of RACH is to control the channel access requests that end-devices generate. We will use the User Equipment (UE) annotation to describe the end-devices (cell phones etc.) and eNodeB to describe the antenna.
The RACH is initiated for many reasons, like when:

- a cell phone wants to establish a connection with the eNodeB
- an end-device changes to a new eNodeB
- a radio-link failure occurs
- a cell phone wants to synchronize with the eNodeB

In the RACH procedure, the time is divided into slots, but these are further divided in the frequency domain. For that reason, the Random Access (RA) slots are created in the time-frequency domain. Every UE is allowed to transmit in an RA slot with a specific signature, which is called "preamble". In each LTE cell there are 64 different preambles. Also, the RACH can be either contention-free or contention-based. The first one is used to manage delayed-constrained access requests with high success requirements, which is now examined in this research, and the second one is what will be analyzed below. The flow of the procedure is the following:

- **Message 1**:
  In the first step, when a UE wants to connect to the eNodeB, it chooses randomly one of the 64 preambles and transmits it in the first available RA slot. It is possible for two or more UE to choose (randomly) the same preamble, occurring to a collision.
- **Message 2**:
  After a successful preamble transmission, the eNodeB sends back to the UE a message, which includes 3 things. The first is a Timing Alignment, which is a parameter for the synchronization. The second one is the UpLink resources that the UE must use

to communicate in the next steps and the third one is a backoff indicator, which helps to reduce the collision probability.

- **Message 3**:
  In the third step, when the UE receives the proper message from the eNodeB, it sends a connection request. It is possible for different UEs to transmit in the same preamble, causing a collision.
- **Message 4**:
  In the last step, if the eNodeB successfully detects the connection request, it responds to the UE with a content-resolution message that contains a mobile id. In the case that UE doesn't receive this message, it will start the procedure from the beginning.

In figure 2.1 we illustrate the above procedure.

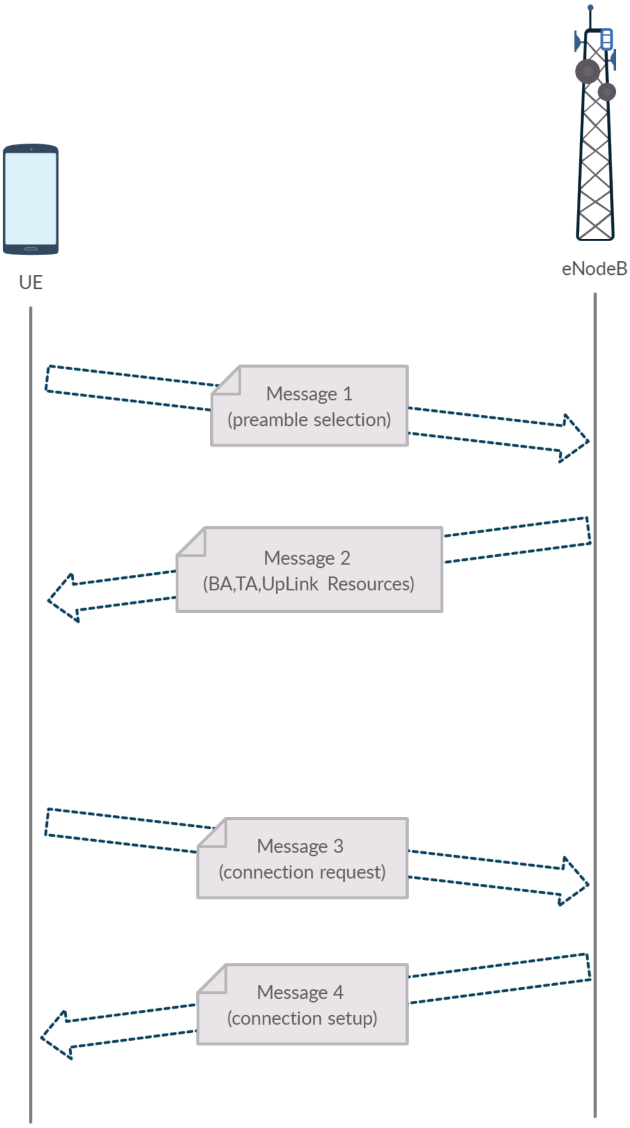
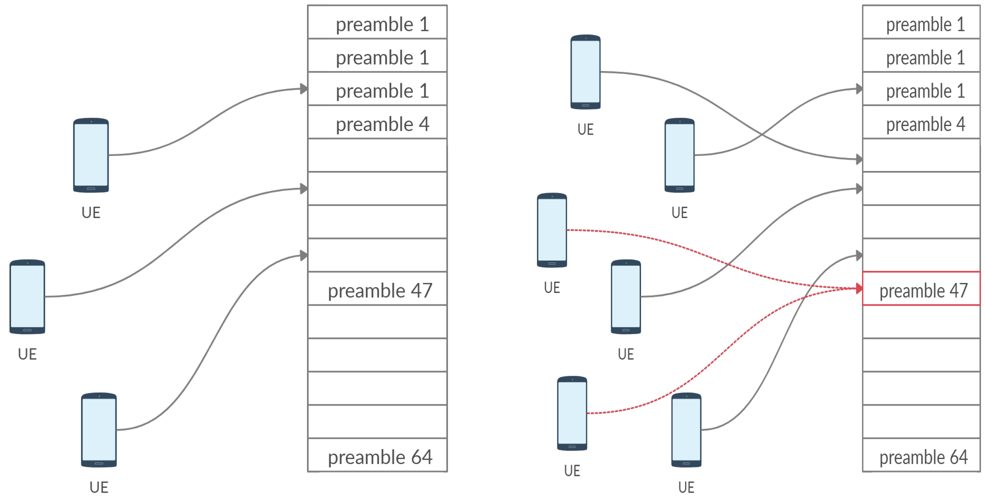


Fig. 2.1: RACH procedure.

2.1.2 Weakness of RACH

In figure 2.2, we illustrate the weakness of RACH. Because the number of available preambles is fixed, as the number of UEs is increasing the probability of two different UEs to choose the same preamble is increasing respectively, as we see in figure 2.2.



(a) Small number of UEs

(b) High number of UEs

Fig. 2.2: This figure shows that as the number of UEs is increasing, the number of available preambles is remain the same, leading to collisions.

2.1.3 Technical Characteristics of the Simulator

The first step to the construction of the simulator was to choose the appropriate framework to implement the RACH procedure. We chose Java, due to its object oriented architecture. Especially, we built five classes, four of them are representing the objects that take place in the procedure (eNodeB, User Equipment, Measurement, Gateway) and in the fifth one, we implement the main flow. Also, we use the following libraries to create charts:

- Simple Java Plot [Yur]
- JFreeChart [Gila]

and we use WPS office to store our experimental results and create more complicated charts.

The basic system parameters that are the same for all the experiments are:

- the time of RA slot is 10 Milliseconds
- the simulation time is 120 second per experiment (or 12.000 slots)

Later, we will describe in more detail how we implemented every object and the main flow.

2.2 Related Work

There are a few researches that aim to analyze the performance of RACH procedure, and all of them are focusing on bringing out the weakness that we state in the previous subsection, the overload issue when high number of UEs are simultaneously attempting to connect to the eNodeB. In researches [AK16], [WCT13] and [WCT12] analytical models are proposed based on existing schemes and specific solutions. In [YEZ15] they present a detailed simulation of the RACH based on the 3GPP release 9 standards. In [PP14], they examine in various scenarios two different methods of allocating preambles (Disjoint and Joint allocation) based on the origin of the call request, M2M or H2H.

In [Bir+15],the authors present the functionality of the RACH procedure and they highlight the overload issue,called "the threat of PRACH overload" and they detail one by one some schemes to alleviate the overload problem. They separate those schemes in two categories. The first is isolating the Human-to-Human (H2H) from the Machine-to-Machine (M2M) communication, while in the second one this separation does not take place, in contrast they allow to the devices to access the same resources but with different probabilities.

There is another research [AKT16], where they are trying to propose a new model for the RACH procedure. They denote that all the previous works that have been done, do not take consideration all the constraints leading to miscalculations. Thus, they build a custom C++ simulator to evaluate their proposed model and compare with the existing models.

One more research that is worth mentioning is [Xie+19].The researchers do not investigate the RACH procedure, but the flow control in the application level at the RPC applications. Specifically, in RPC applications the server has a threshold, that determines the number of access requests that it can process. The researchers proposed an interesting algorithm to set the threshold, based on the status of the server (congested or not). Influenced by this approach, we create a similar scheme (case 1 as we will see later), where we set a threshold, based on the delay of the packets that are sent.

Finally, in [Pol+16] the researchers are extending the NS3 network simulator in order to improve the existing simulation of the RACH procedure building the LENA+ module[Gilb].

Taking into consideration all the related work, we decide to build a custom RACH simulator to test our scenarios in the application level. We choose to build our simulator from scratch, because the technical characteristics of the scenarios that we are trying to investigate are not easily applicable in the existing simulators.

3

System Design and Implementation

3.1 Design

In this section, we denote the data traffic, and we lay out some Cases, which we will simulate later. Specially we specify the traffic that different sensors can produce, and we build two different architectures to investigate the performance of the network under each architecture.

3.1.1 Traffic Modeling

It is crucial to consider the kind of traffic that sensors generate because it affects the efficiency of the network. To simulate as much better the different Cases, we need to include the proper traffic load in our system. The traffic can be either:

- **periodic**: temperature or traffic camera sensors, which send data in a specific period (for example every 5 seconds) ,
- **random**: traffic sensors, smart cars, or sensors that need to send their data if a measurement is above a threshold and the most important because we are in the wireless channel, are the smartphones.

Thus, we will have two different kinds of sensors. The first one will produce periodic data, and the second one will produce data in a random period. The number of sensors at each kind will be a parameter of the system that we will examine in the simulations.

3.1.2 Case 1 - AIMD

We will first denote the most classic Case: one eNodeB and N-number of UEs that want to connect. This Case's flow begins with some of the UEs that want to send their message to the network. They need to select one of the 64 available preambles and send the Message 1 to the eNodeB, under the selected specific preamble. If no collision occurs, we can assume that the sensor will later send his message after the RACH procedure. As we discussed, while the number of UEs increases, the possibility of having a collision is increasing, respectively.

An interest approach is proposed in [vsi19]. In this research, they are trying to reduce the data flow in IoT under an AIMD (additive increase and multiplicative decrease) scheme. Specifically, they implement an algorithm, similar to the AIMD of TCP's congestion window, where they adjust the sensors' sending period time, based on some strategies. Moreover, they define a new strategy layer, with four different schemes (data accuracy, response time, energy, and privacy protection). Depending on the application requirements, they change the period between consecutive measurement requests, resulting in a reduced data flow.

In our Case, we can illustrate this approach as follows. We first need to have a metric to determine when to apply the increase of the sending period. This metric can be the segment of time between the moment the sensor gives the data to the low layer to be sent, and the moment the data are actually sent. Precisely, we can denote as t_{start} the time the data generated from the sensor and t_{send} when the data are sent to the wireless channel.Also we define as t_{delay} the difference between t_{send} and t_{start}. This fraction of time includes the following actions:

- the transportation of the data to the lower layers
- the implementation of the RACH procedure

The second action includes the delay that is caused by a wireless overload network. If a collision occurs, then the second action (the RACH procedure) will take more time. The cause is that after the stage of selecting a preamble, it is possible a collision to occur between two different UEs that choose the same one in an overloaded network. When this happens, the RACH will wait a short amount of time, and then it will be initiated again. This extra time is the delay that is caused by the congested wireless channel. Thus, the $t_{delay} = t_{send} - t_{start}$, it will be longer and can determine if the channel is overloaded, as we see in the figure 3.1.

Now that we have the metric, we can set one more parameter, the threshold, that if the metric we define later is above it, we will double the sending period. The value of the threshold will be defined in the next chapter. It is essential to mention that, in order to double the sending period in a sensor, the application that this sensor belongs must be tolerant of this increase of the period. Indeed, many applications can achieve this, like temperature monitoring apps, apps that monitor the concentration of heavy metals or the wind's speed, and more. Another approach is that even if the application cannot be tolerant in doubling the sending period, we can still double it. Let us assume the following scenario, in a heavy metal monitoring app, the sensors send every 10 seconds their measurements.

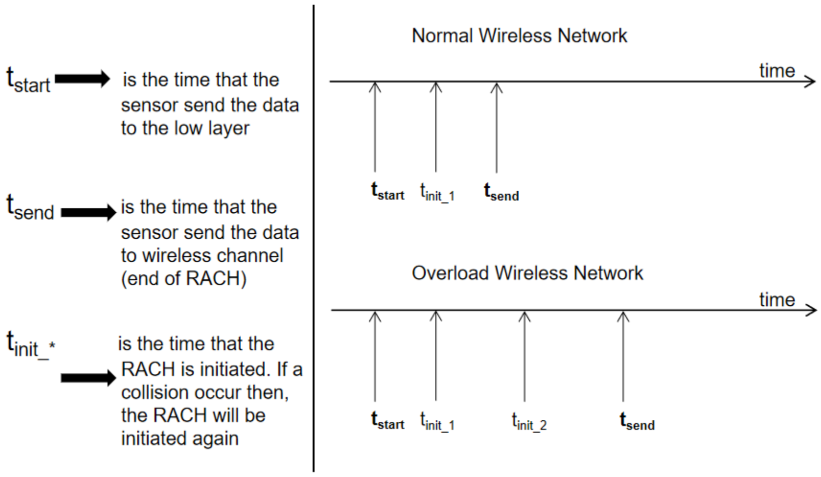

t_{start} is the time that the sensor send the data to the low layer
t_{send} is the time that the sensor send the data to wireless channel (end of RACH)
t_{init_*} is the time that the RACH is initiated. If a collision occur then, the RACH will be initiated again
Normal Wireless Network
time
t_{start}
t_{init_1}
t_{send}
Overload Wireless Network
time
t_{start}
t_{init_1}
t_{init_2}
t_{send}

Fig. 3.1: In the figure above, we see that the time between t_{start} and t_{send} is longer in an overloaded network. Also, t_{init_1} is when the RACH is initiated for the first time and t_{init_2} is when start again, because a collision occurs in the first.

Due to the congestion, one sensor doubles his period, so when 10 seconds pass through the last sending, the application will expect a measurement from the sensor. However, because he will have increased his period, no measurement will be received. In this Case, the application could create the expected measurement based on the last measurements to maintain proper functioning. So the increase in the sending period is a feasible solution.

The final step is to define when the sending period returns to normal. We will use the same metric and the same threshold again. After each sending of measurement, we will compare the delay (t_{delay}) with the threshold, and only when the delay is under the threshold will we set the period to normal.

3.1.3 Case 2 - Gateways

In many IoT environments, a device is used as an intermediate to connect some sensors to the internet. This device is called gateway, and it is beneficial for many reasons like:

- in some IoT environments, a sensor may not be able to connect to the network due to the low range of his transmitting signal
- we want to execute specific procedures to the data before we push them to the internet

We can use this existing architecture of gateways in order to achieve the smallest collision probability. Specifically, under the gateways' scope, we can cluster the sensors in small groups (N-number of UE per cluster) to create small teams with one leader (the gateway) per team. In this architecture, the team members send their data to the leader, and he is responsible for forwarding them to the eNodeB, as we observe in figure 3.2. The number of UEs at each team is a system parameter that we tune later in the simulations. Thus, we may overcome collisions because we reduce the number of requests from the UEs to the wireless channel.

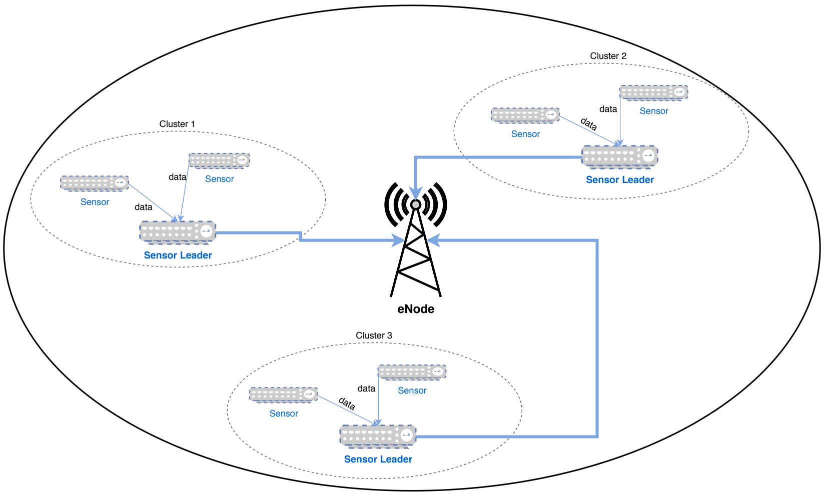


Fig. 3.2: This figure shows the basic architecture of Case 2. There are 2 sensors at each cluster. Sensor Leaders are the gateways.

Moreover, those leaders can be either classic gateways devices installed in the IoT network infrastructure, either regular IoT devices (sensors). The difference between them is that a classic gateway only forwards the data that is receiving to the eNodeB, while a regular IoT device it should forward the data from the team members and, at the same time, should handle his own data.

So, in the first example, we have the advantage that the gateway does not generate data; thus, it can forward the receiving data directly, resulting in higher efficiency, but there is an extra cost of purchase and installation of those gateways. In the second Case, we do not have additional costs because the gateways are the current sensors, but a notable drawback is that the leader has to forward the team's data and, at the same time, to control his data, leading to a possible efficiency problem. A tactful compromise is to substitute the sensors that will be leaders with better devices, lessening the cost from the first Case and the overload from the second one.

After the description of the architecture, we have to determine how the gateway will operate. The gateway will wait a short amount of time to collect data from the UEs and operate as a regular device. We define as t_{period} the period that a gateway gathers the team members' data. Precisely, let us assume that there is a cluster of N UEs that can be connected to the team leader (the gateway), under a wireless protocol (WiFi) and send their measurements to him. The gateway gathers those data for t_{period} time and then initiates the necessary procedures to send them to the network, like the RACH. While the gateway waits to establish a connection with eNodeB, the data that are going to send are stored to a buffer, and a new period (t_{period}) begins. This is a repeating pattern that it can relieve a wireless overload channel because instead of having N different UEs trying to connect with the eNodeB, we have one gateway to take over the jobs that include the communication with the eNodeB, as we can see in the figure 3.3. The variable t_{period} it will be tuned in the simulations.

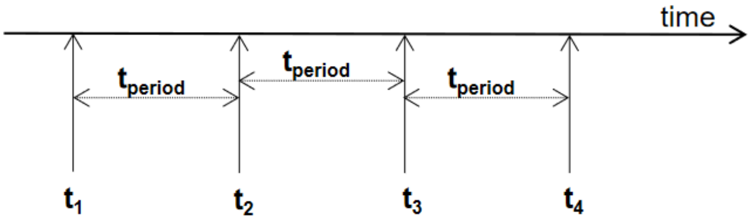


Fig. 3.3: Between t_1 and t_2 the gateway will gather all the data that the sensor will send. At t_2 the gateway will initiate all the necessary procedures to establish connection with eNodeB and it will start listening for new data from the sensors until t_3. At t_3 it will again establish a connection with eNodeB to send the data that receive in the time between t_2 and t_3

A challenging issue that is out of this thesis's scope is to create the clusters dynamically. There are different kinds of applications with different requirements that need higher availability. For instance, some apps may be vital to have their measurements (from the respective sensors) available under a specific time, like traffic control apps. For others, it may not be so important, like temperature monitoring apps. Therefore, sensors that belong to different applications create different criteria for how long the t_{period} of the gateway can be. If in the same cluster is a sensor of traffic control and a sensor of monitoring temperature app , the first will require short t_{period} to maintain his availability. In contrast, the second one can support longer t_{period}. Thus, an interesting approach is to form the clusters based on the application's strategy that the sensors belong.

Additionally to the above approach, an impressive strategy is to fluctuate the number of cluster members depending on the t_{period}. In a gateway with short t_{period}, we may have many team members, because the collected data are sent frequently, instead of a gateway with a long t_{period} that it is better to be a few members in the team because the data are sent more sparsely in time. More specifically, the more connected devices to a gateway, the more data is gathered, so it will be more efficient to have a smaller t_{period} to forward the data faster in order the gateway be able to receive the new incoming data.

3.2 Implementation

In this section we explain how we implement the simulator to reproduce the RACH procedure, the AIMD and the Gateway scenarios.

3.2.1 Architecture

As we described in the introduction, we build five objects:

- Main
- eNodeB
- UE
- Measurement
- Gateway

In figure 3.4 we illustrate the objects with their basic attributes and their relation.

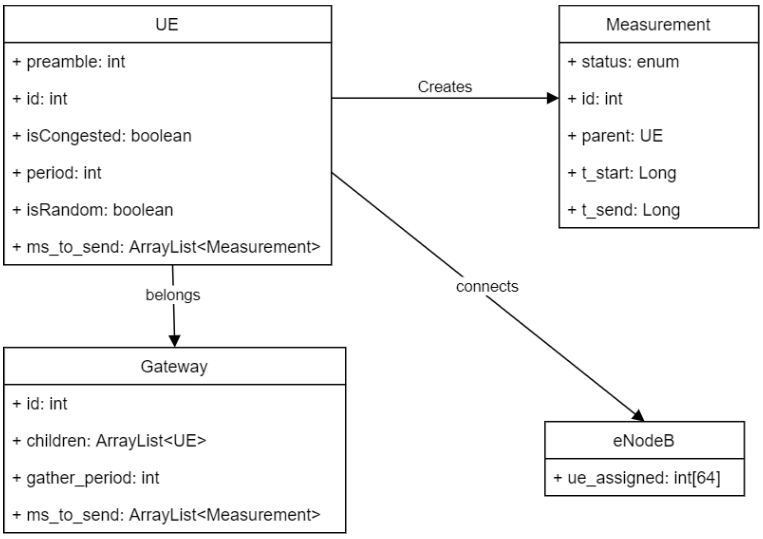


Fig. 3.4: This figure shows the architecture of the simulator.

UE Class

In this class, we implement the behavior of the UE. The important variables are shown in figure 3.4. *"Preamble"* is an integer, where we store the value of the preamble (1-64) that the UE will select in the initiation of RACH procedure. The variable *"id"* is a serial number to identify the UEs, *"isCongested"* is a Boolean variable, which is used in Case 1 (AIMD) as we will see in next sections. In addition, *"period"* is an integer which defines the period of the UE (in seconds), *"isRandom"* is a Boolean variable which is true if the UE is random and false if it is periodic, and finally *"ms_to_send"* is an ArrayList which stores the measurements of the UE that have not been sent yet. We designate that a random UE is generating measurements in random time. Later we will see how exactly we implement this.

The two basic methods of the UE class are those that generate the measurements. Specifically, we create one method for the periodic UEs ($periodic\_traffic\_generator(period)$) and another for the random UEs($random\_traffic\_generator()$). In both methods, in order to simulate the behavior of a real device (sensor or mobile phone), we use a Java Class, ScheduledExecutorService, which offers methods to run a piece of code periodically. This methods create an executor (in Java terms), which runs periodically. Specifically, for each method :

- $periodic\_traffic\_generator(period)$: We create a Measurement object and we connect it to the UE that generates it (the connection is implemented as follows: We have created a List in the UE, that we store the Measurement objects that have not been sent yet. So, we add the Measurement object that was created in this List). Also, in Case 1, we check if the variable "isCongested" is true and if it is we change the period of the UE accordingly, as we will see later. This functionality is repeated based on the UE period.

- $random\_traffic\_generator()$: First, we define the probability of a (random) UE to generate a measurement, which is 0.0025. In order to achieve this, every time that the executor is running, we use the Java Random Class to generate a random value between 0-399, then only if this value is 399, we create a Measurement object and we connect it to the UE. Next, we have set the period of this executor to be 10 ms, thus the probability of a random UE generating a measurement in 1 second is 0.25.

Finally, in both methods, because we simulate real-world processes, we add randomness characteristics, so that the simulation will be more precise to them. Specifically, we slightly modify the period of each periodic UE. This modification is implemented with the Java Random Class as follows: we generate a random value between [-250,250] and we add it to the period (in Milliseconds) of the UE. Another random characteristic is the modification of the time that a UE (random and periodic) starts. This can be implemented by setting an initial delay in the Executor. We set this initial delay (in Milliseconds) to be a random value between [95-135].

eNodeB Class

The eNodeB class basically simulates the cell tower. We have created an array, "*ue_assigned*", that keeps the UE that have chosen different preambles, so that no collision occurs at Message 1.

Measurement Class

In this class, we implement the Measurement. A Measurement can be data, produced by a sensor, or a cell phone attempt that connects to the eNodeB. Thus, we define a Measurement to be something that initiates the RACH procedure, resulting in the transmission of Message 1 (as we see in chapter 2).

The important variables are shown in figure 3.4. *"Status"* is an enum variable (values: "PENDING", "COMPLETED"), which shows the state of a Measurement. "PENDING" if it is not sent and "COMPLETED" if it has been sent.

We restate that a Measurement is sent if it has transmitted the Message 1, which means that no collision has occured, as we in figure 3.1.

The variable *"id"* is a serial number to identify the Measurement, *"parent"* stores the UE that generates this Measurement. In addition, *"t_start"* is a Long variable which defines the time (in UNIX time) that the Measurement was generated and *"t_send"* is a Long variable which defines the time (in UNIX time) that the Measurement was sent. If the *"t_send"* is null, it means that the Measurements has not been sent.

Gateway Class

This Class implements the functionality of the gateway. The variable *"id"* is a serial number that identifies the Gateway, *"children"* is a List where we store the UEs that are assigned to the Gateway and *"gather_period"* is the time period that the gateway is gathering Measurements from the assigned UEs. Lastly, *ms_to_send*, is an ArrayList which stores the measurements of the UEs that have not been sent yet.

The important method of this class is the method $start()$, which is starting the Gateway. As in the previously mentioned methods of the UE, (*periodic_traffic_generator(period)* and *random_traffic_generator()*), the $start()$ method uses an Executor with a period equal to gathering period. This Executor includes the piece of code that implements the gathering of Measurements that the child UEs have generated.

3.2.2 Implementation of RACH

The only class that we did not analyze is the Main. In this class we have implemented the RACH procedure. To be more precise, we have not implemented the whole RACH procedure, but only the part that is related to the efficiency of it. This part includes the procedures until the Message 1 (as we see in chapter 2). The reason is, that the collisions, that lead to the decrease of the RACH efficiency are occurring right after the the transmission of Message 1, thus we simulate up until to this point.

We implement this part of RACH as follows: we use the ScheduledExecutorService Class (as in the UE and Gateway Class) to create an Executor with period of 10 ms. In the body of the Executor we call the $slot()$ method (algorithm 1). So, every 10 ms the algorithm 1 is running. We choose 10 ms because in the RACH procedure the slot has a period of 10 ms.

Algorithm 1 slot

```
1: procedure SLOT(UEs)
2:     UE ← check_UE(UEs)            ▷ returns the UEs that has measurements to send
3:     for ue ∈ UE do
4:         ue.preamble ← random(1,64)
5:         if check_collision(preamble) then      ▷ returns true if no collision occurs
6:             ue.send_measurements()
7:         else
8:             number_of_collisions++
```

In line 2, we choose the UEs that have Measurements to send (which means that we take the measurements that have the t_send value to null). Then, we iterate these UEs and assign to them a random preamble. In line 5, we check if another UE has chosen the same preamble, leading to a collision. If no collisions have occured, in line 6, we send the measurements of this UE by calling the $send\_measurements()$ method, which changes the t_send variable of each Measurement to the current time in UNIX format. Thus, we calculate the delay for each measurement that has been sent, by subtracting the t_start from the t_send.

3.2.3 Implementation of Case 1 - AIMD

We implement Case 1 as follows: first, we create the Boolean variable *"isCongested"*, that defines if the network is congested (True) or not (False). If it is congested, we modify the method of the UE $periodic\_traffic\_generator(period)$, by doubling the period of this specific UE.*"isCongested"* changes in the slot() procedure, as we see in algorithm 2.

Algorithm 2 Case1

```
 1: procedure SLOT(UEs)
 2:     UE ← check_UE(UEs)              ▷ returns the UEs that has measurements to send
 3:     for ue ∈ UE do
 4:         ue.preamble ← random(1,64)
 5:         if check_collision(preamble) then          ▷ returns true if no collision occurs
 6:             measurements ← ue.send_measurements()
 7:             for ms ∈ measurements do
 8:                 if ms.delay() > threshold then        ▷ threshold is system parameter
 9:                     ue.isCongested ← True
10:                 else
11:                     ue.isCongested ← False
12:         else
13:             number_of_collisions++
```

In lines 7-11, we iterate the measurements that have been sent in this slot and we check the delay that they have (line 8). If this delay is above the threshold, which is a system parameter, we set the *"isCongested"* to True, which means that this UE will generate a measurement in double time of the period, with s purpose of reducing the amount of requests of the connection (Message 1) to the eNodeB. Also, we check if the delay is under the threshold that we have set, and if it is we the *"isCongested"* to False, in order to reset the period of the UE back to normal.

3.2.4 Implementation of Case 2 - Gateways

To implement Case 2 we create the Gateway Class as we described later. Then, in the initiation of the simulator, we separate the total UEs into teams. Each team has a size equal to the value of the system variable *"number_in_each_team"*, which defines the number of child's of the gateways. Then, we follow the same procedure similarly as in the Normal scenario(algorithm 1), as we see in algorithm 3.

It is important to explain extensively how the start() method works. In algorithm4 we show the basic logic.

Algorithm 3 Case2

```
procedure SLOT_CASE_2(Gateways)
    GT ← check_gt(Gateways)    ▷ returns Gateways that has measurements to send
    for gt ∈ GT do
        gt.preamble ← random(1,64)
        if check_collision(preamble) then    ▷ returns true if no collision occurs
            gt.send_measurements()
        else
            number_of_collisions++
```

Algorithm 4 Executor in start

```
1: repeat
2:     for ue ∈ children do
3:         for ms ∈ ue.measurements do
4:             measurements_gt.add(ms)
5: until
```

The method $start()$ is running inside a Gateway object. This objects has children, as we define later, which are the UEs that are assigned to this gateway. Thus, in method $start()$ we iterate the children (line 2) and for every child we take the measurements that they generate and have not been sent to the eNodeB successfully yet (line 3). Then, we add those measurements to the gateway's measurements, that need to be sent(line 4). This method is running with a period equal to the *"gather_period"*, which means that each gateway gathers the measurements of the respective UEs every *"gather_period"* time.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

4

Evaluation

4.1 Experimental setup

In this section, we will describe the setup of our Experiments. We run the simulator for each of the scenarios that we describe in chapter 3. Specially we tune the following parameters:

- the total number of User Equipment
- the percentage of random and periodic User Equipment
- the threshold (Case 1)
- the number of UEs that are assign to one Gateway (Case 2)
- the gathering period of the Gateways (Case 2)

Then, we measure the average delay, the standard deviation of the delays, the number of collisions, and the total attempts of requests (an UE, before send one or more measurement, sends the Message 1 to the eNodeB) , in order to investigate how each scenario behaves and to compare them.

In more detail, after the end of the simulation, we have gathered all the measurements and their delay. The delay is calculated as follows. We take the time that the measurement is sent, and we subtract the time that the measurement is generated. Moreover, the number of total attempts refers to the times that a sensor is sending the Message 1 (preamble selection), while the number of collisions refers to the times that the sensor did not get a response from eNodeB, after sending the Message 1. So we can calculate the collision rate($\frac{collisions}{totalattempts}$), which shows the percentage of collisions in the total attempts.

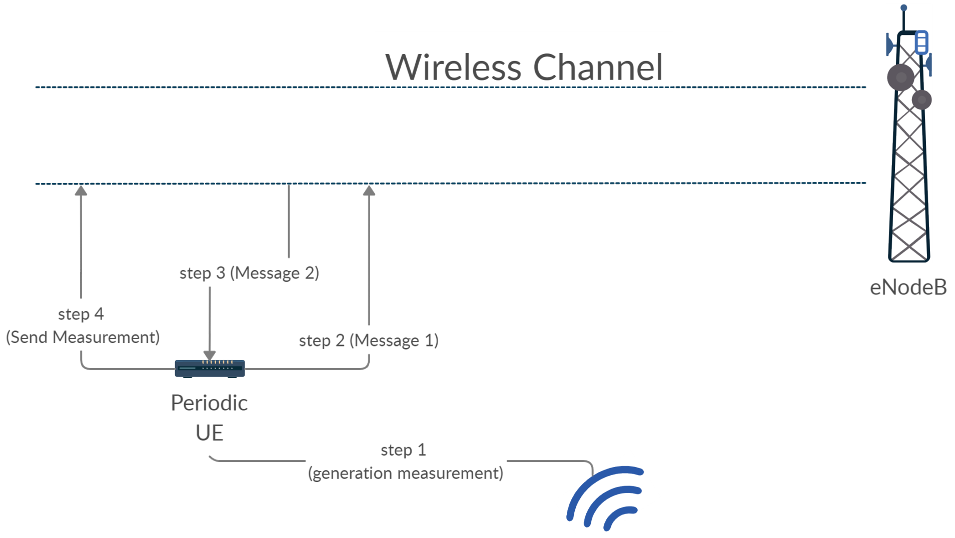


Fig. 4.1: Experimental Setup - no collision

In figure 4.1 we illustrate how we measure the delay. In the first step, the periodic UE generates a measurement and initiates the RACH procedure in order to get up-Link Resources and send the measurement to the network. This initiation, starts in step 2, where the periodic UE, choose a preamble and send it to the wireless channel. If collision does not occur, the eNodeB will respond with the Message 2 (in step 3) and then the UE will send the Measurement under the Up-Link resources that the eNodeB has assign to it. The delay that we measure is the time difference between step 1 and step 4. As we see in figure 4.2 if collision occurs in step 2, then the UE will have to send again the Message 1, resulting to a higher delay (time difference between step 1 and step 5).

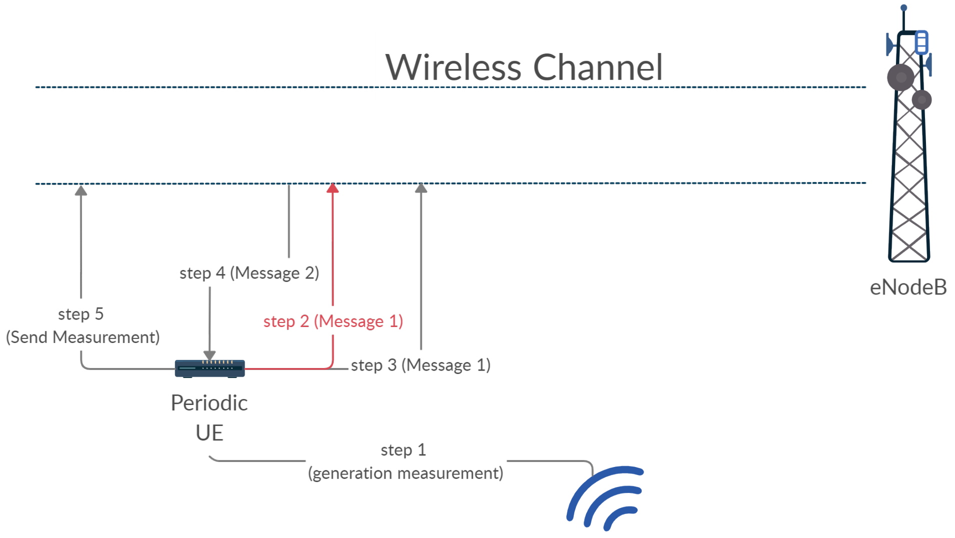


Fig. 4.2: Experimental Setup - collision

In figure 4.3 we show the difference between the periodic and the random UE. We observe that a random UE generates measurements in non periodic time, it produces in slot 1 then in slot 4 and finally in slot 27. In contrast, a periodic UE generates measurements in periodic time, as we see, it first produces in slot 5, then in slot 10 and so forth. Moreover, the system parameter "percentage of random and periodic UE", is referring to the number of each kind of UE.

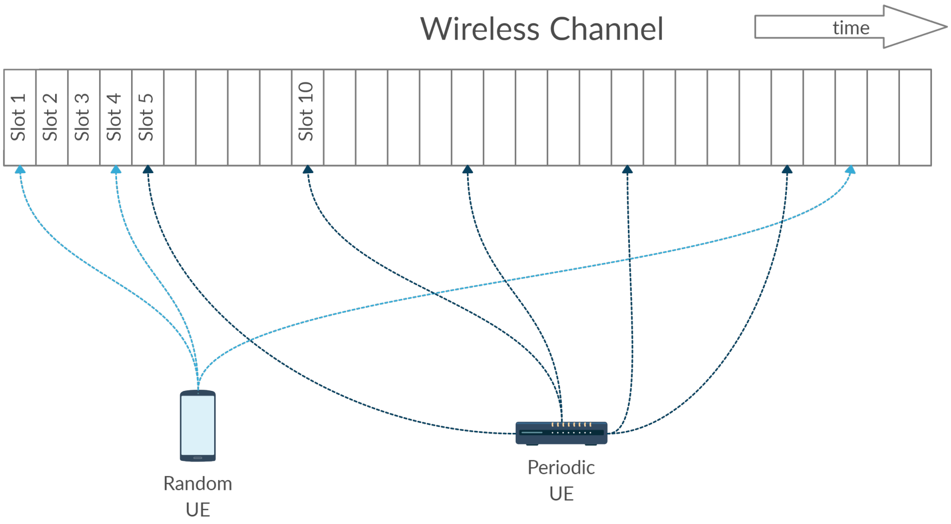


Fig. 4.3: This figure shows the difference between periodic and random UE. The first one, is requesting in random time, while the second one is requesting in periodic time (every 5 slots).

In figure 4.4 we show what happens when a collision occurs. The UEs that were involved in this collision are sending again a request, in the next slot. This extra time of the new requests leads to a higher delay.

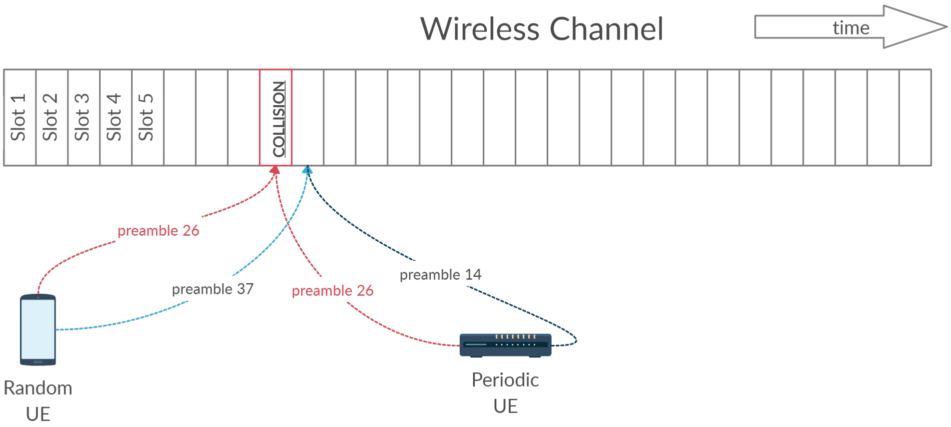


Fig. 4.4: This figure shows what happens when a collision occurs. The involved UEs are requesting again to the next available slot.

4.2 System evaluation

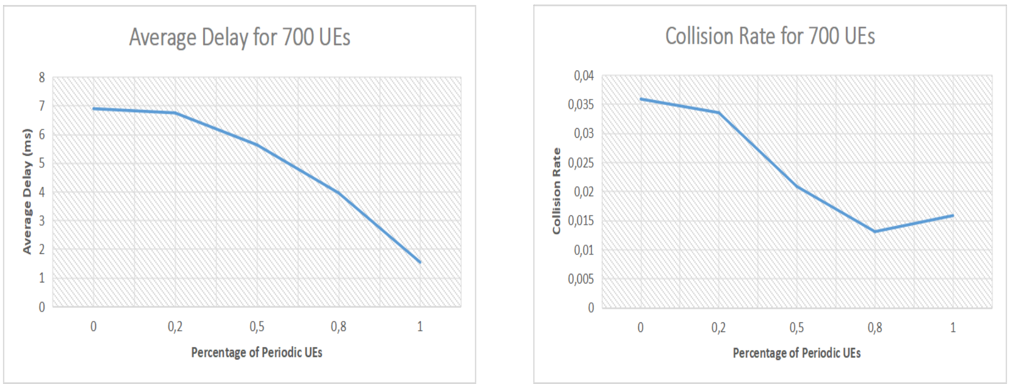
In this section we present the performance evaluation of the proposed system. First, we quote the results that we get for each Case and then we analyze them, in order to emerge the effect of each strategy in the network.

4.2.1 Results

Next, we present the experimental results for each scenario.

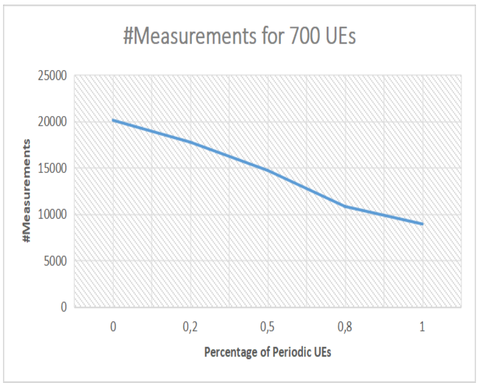
4.2.2 Normal Case

First, in the Normal Case scenario, we run our simulator with tuning parameter the percentage of random and periodic UEs, in order to see how the average delay and the collision rate is fluctuating. In figure 4.5, we present the results while the number of UEs is 700 and we change the percentage of the Periodic UEs.



(a) Average Delay

(b) Collision Rate

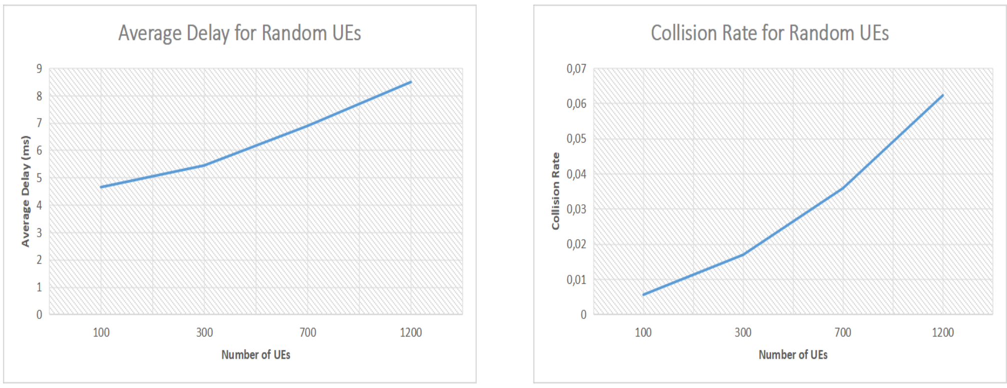


(c) Number of Measurements

Fig. 4.5: Sub-figure (a) presents the average delay while the percentage of periodic UEs is increased, Sub-figure (b) presents the collision rate while the percentage of periodic UEs is increased, Sub-figure (c) presents the number of measurements that generated while the percentage of periodic UEs is increased.

We observe that, as we increase the percentage of periodic UEs, by increasing the number of periodic UEs and decreasing the number of random UEs, the average delay and the collision rate are decreasing. This is due to the probability of a random UE to generate a measurement in a given moment. As we see in chapter 3, the probability of a random UE to generate a measurement in 1 second is 0.25, while, in the beginning of chapter 4, we state that the period of the UE is at least 10 seconds. Thus, a random UE will generate, on average, more measurements in a specific time period, than a periodic UE. This is the reason why the average delay and the collision rate are decreasing. In 4.5(c) we evaluate this by noticing that the number of measurements that were generated, are decreasing as we increase the percentage of periodic UEs, which means that we increase the number of periodic UEs and decrease the number of random UEs.

Next, we calculate the average delay and the collision rate for only random UEs, while we increase the number of UEs.



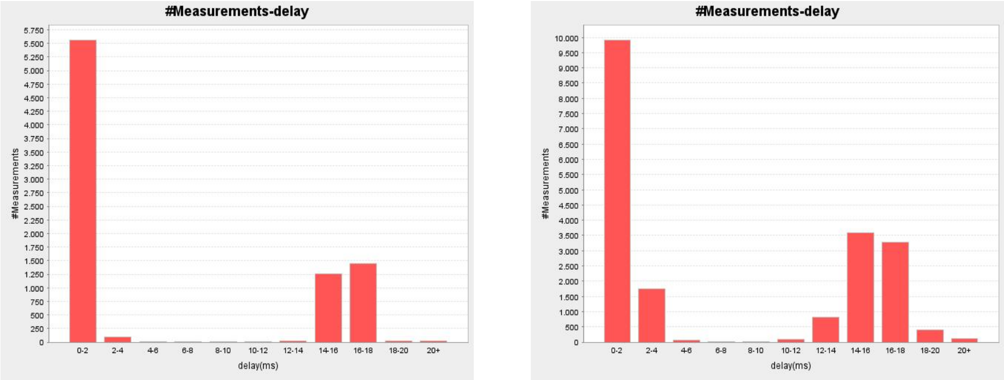
(a) Average Delay

(b) Collision Rate

Fig. 4.6: Sub-figure (a) presents the average delay while we increase the number of UEs and sub-figure (b) presents the collision rate while we increase the number of UEs.

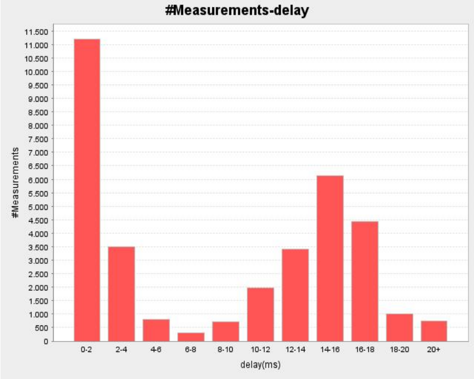
In figure 4.6, we observe that as we increase the number of UEs, the average delay and the collision rate is increasing respectively. This is expected because as we increase the number of UEs, we increase the number of measurements that are generated, leading to a higher number of requests to the channel. This can cause more collisions, resulting in a higher delay.

In experiments mentioned above, we compute that the standard deviation is approximately 7.1, thus, instead of calculating the confidence intervals, we calculate how the measurements are distributed based on their delay. In figure 4.7, we present the results. We see that as we increase the number of UEs, the number of measurements whose delay is above 8 milliseconds, is increasing respectively.



(a) 300 UEs

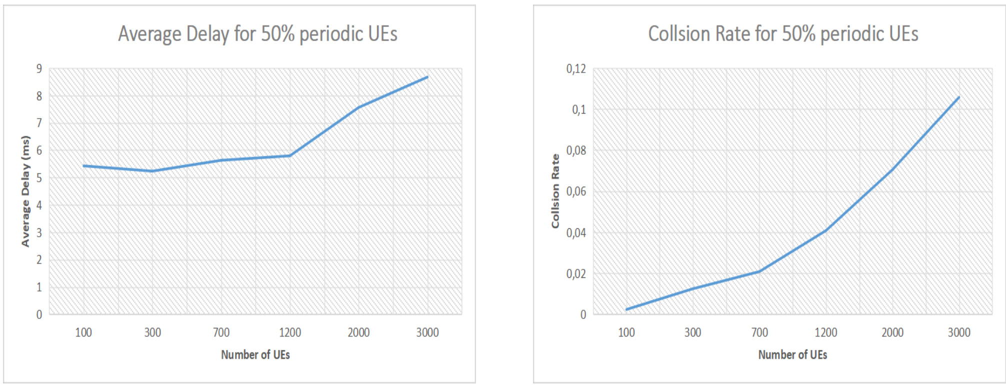
(b) 700 UEs



(c) 1200 UEs

Fig. 4.7: These figures present the distribution of the measurements, based on their delay, as we increase the number of the UEs(Percentage of Periodic UEs 0).

Following, we run the same experiments, but for the percentage of periodic UEs set to 50%, which means that we have the same number of periodic and random UEs.



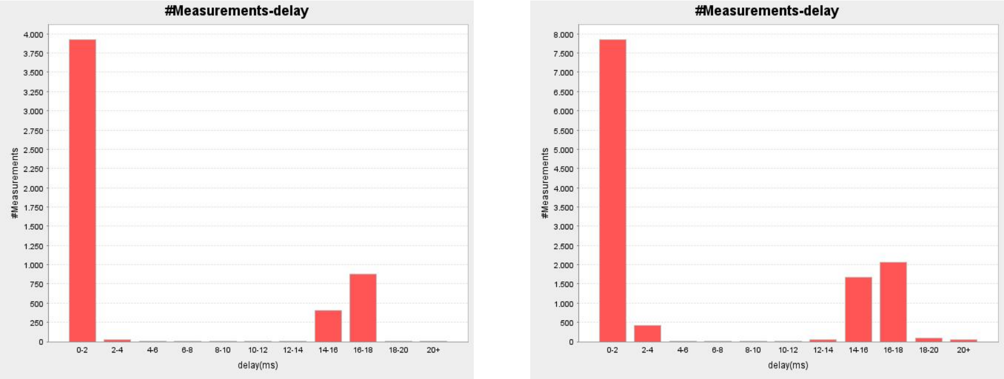
(a) Average Delay

(b) Collision Rate

Fig. 4.8: Sub-figure (a) presents the average delay while the number of UEs is increased and sub-figure (b) presents the collision rate while the number of UEs is increased (Percentage of periodic UEs 0.5).

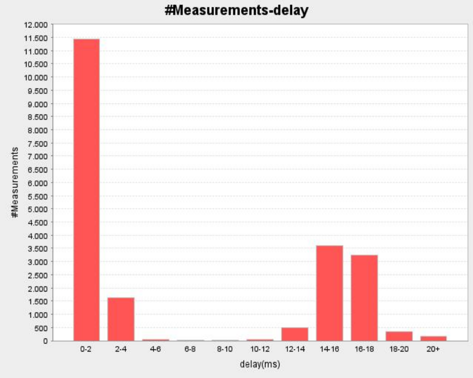
In figure 4.8, we see, that the average delay and the collision rate is increasing as we increase the number of UEs, but the average delay, when the number of UEs is between

100 and 1200 remains somewhat steady. This is, probably, due to the tolerance that the system may have, up to a specific number of UEs.



(a) 300 UEs

(b) 700 UEs

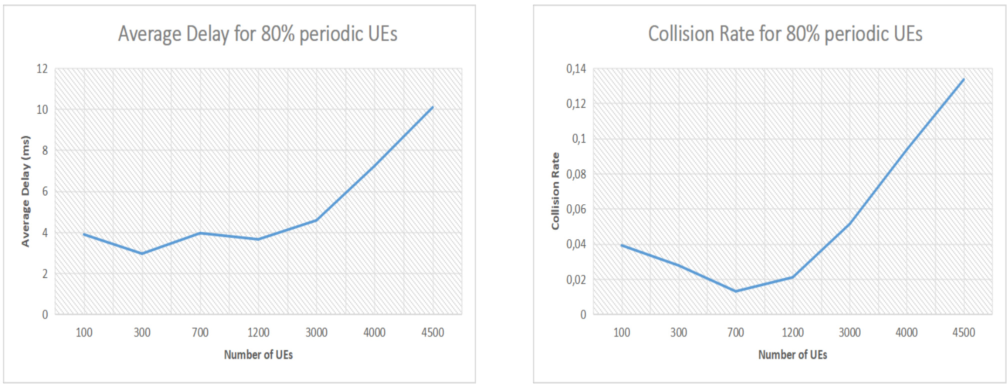


(c) 1200 UEs

Fig. 4.9: These figures present the distribution of the measurements, based on their delay, as we increase the number of the UEs (Percentage of Periodic UEs 0.5).

Again, in figure 4.9, we present the distribution of the measurements, based on their delay. It follows the same pattern with figure 4.7, but the number of the measurements that are above 8 milliseconds is smaller. This is due to the smaller number of measurements that were generated in this experiment. For example, in the experiment of figure 4.7(b), 20097 was generated, while in the experiment of figure 4.9(b), 17736 was generated.

Finally, we run the same experiments, but for the percentage of periodic UEs set to 80%. In figure 4.10 we present the results.



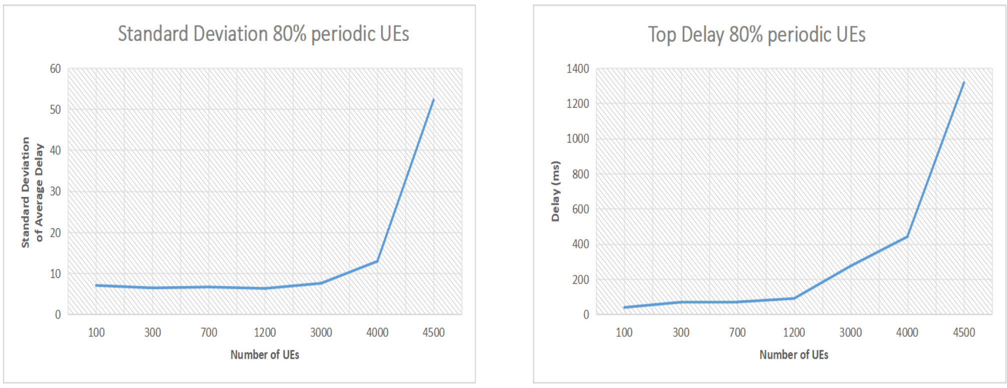
(a) Average Delay

(b) Collision Rate

Fig. 4.10: Sub-figure (a) presents the average delay while we increase the number of UEs and sub-figure (b) presents the collision rate while we increase the number of UEs (Percentage of periodic UEs 0.8).

We observe that in figure 4.10(a) the average delay has a upward trend, but up to the value of 3000 UEs it is approximately 3.5-4.2 milliseconds, which means that the system, with these parameters, has a tolerance up to 3000 UEs.

In figure 4.11, we present how the standard deviation and the Top Delay of all measurements are fluctuated. Up to 3000 UEs the standard deviation remains 7 and for value greater or equal to 3000 UEs, it is increasing exponentially. This shows that the system set with these parameters can operate normally,up until 3000 UEs. Also in 4.11(b), we observe that the Top Delay is increasing as we increase the number of UEs. Actually, up to 1200 UEs the Top Delay is increasing slowly and steadily, and when we pass this number, it is increasing with an exponential rate.



(a) Standard Deviation of average delays

(b) Top Delay

Fig. 4.11: Sub-figure (a) presents Standard Deviation of average delays while the number of UEs is increased and sub-figure (b) presents the Top Delay of all measurements for each experiment (Percentage of periodic UEs 0.8).

4.2.3 Case 1 - AIMD

In Case 1, we first tune the system parameter *"threshold"*, in order to figure out the optimal value. We run 20 different experiments, that we have divided into four groups:

- *Group A*: Periodic Percentage 0.5, variable number of UEs and no threshold (Normal Case)
- *Group B* : Periodic Percentage 0.5, variable number of UEs and threshold 10 ms
- *Group C* : Periodic Percentage 0.5, variable number of UEs and threshold 15 ms
- *Group D* : Periodic Percentage 0.5, variable number of UEs and threshold 20 ms

Figure 4.12 shows the average delay of the above experiments.

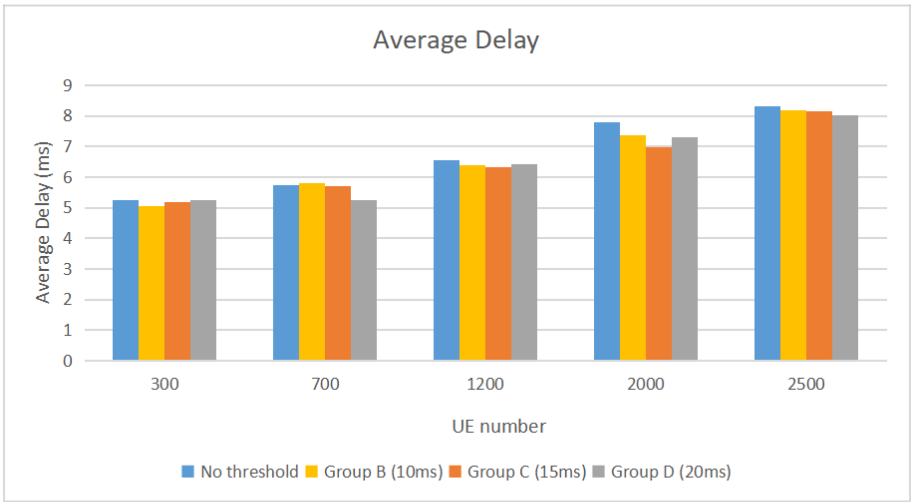


Fig. 4.12: This figure shows the average delay for the different group of experiment while we increase the UEs.

We observe that the average delay has an upward trend as the number of UEs increased, which is normal because if the number of UEs is increasing, then the probability of o collision is increasing.

In order to figure out if the Case 1 is improving the RACH performance, we create the figure 4.13, where we illustrate the difference of average delay of group B,C and D from group A. In other words, we plot how much the average delay has decreased when we apply Case 1.

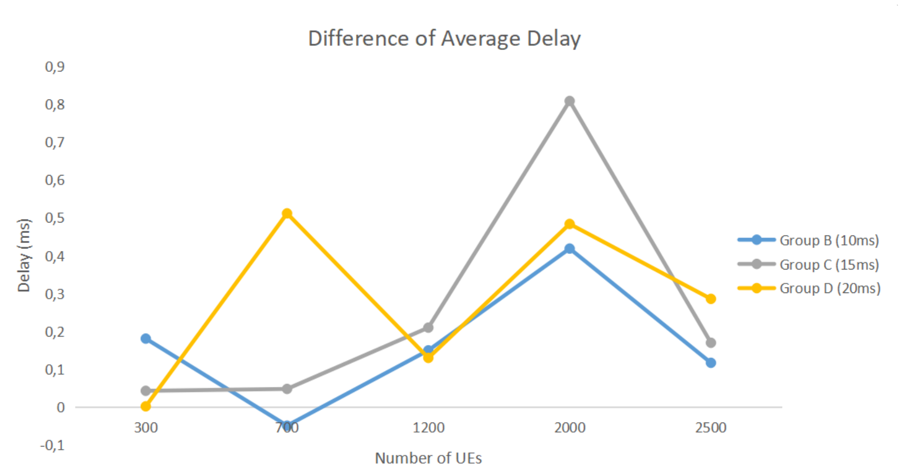


Fig. 4.13: This figure shows the difference of average delay of group B,C and D from group A.

We notice that in the value of 700 UEs,Group B (with threshold 10ms) does not perform better from the Normal Case. This can be a result of a simulation error or if no collisions have occured in this experiment, so the delay has never exceeded the threshold. Generally, we observe that up to a number of 1200 UEs we have a standard improvement of 0.2 ms in the average delay. Then as we increase the number of UEs up to a number of 2000, the scenario ameliorates the performance of RACH in average delay by 0.4-0.8 ms. In addition, as we keep increasing the number of UEs the scenario still performs better, but is not the optimal. We conclude, that threshold of 15ms is the best, based on the average delay.

Next, we will present similar results, but for the collision rate.

Figure 4.14 shows the collision rate of the experiment group A through D.

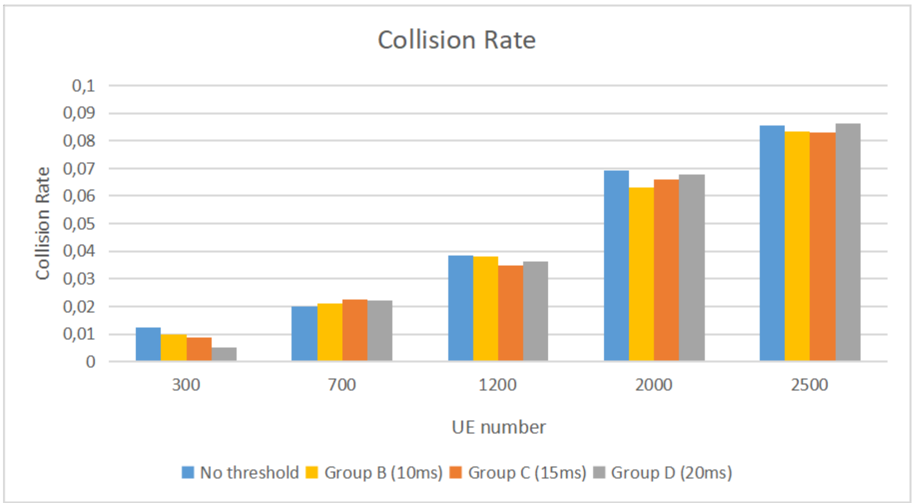


Fig. 4.14: This figure shows the collision rate for the different group of experiment while we increase the UEs.

We observe that the collision rate has an upward trend as the number of UEs is increased, which is normal because if the number of UEs is increasing, then the probability of o collision is increasing.

In figure 4.15, we present the difference of the collision rate between the experiments of group B to D, from the experiments of group A (Normal Case).

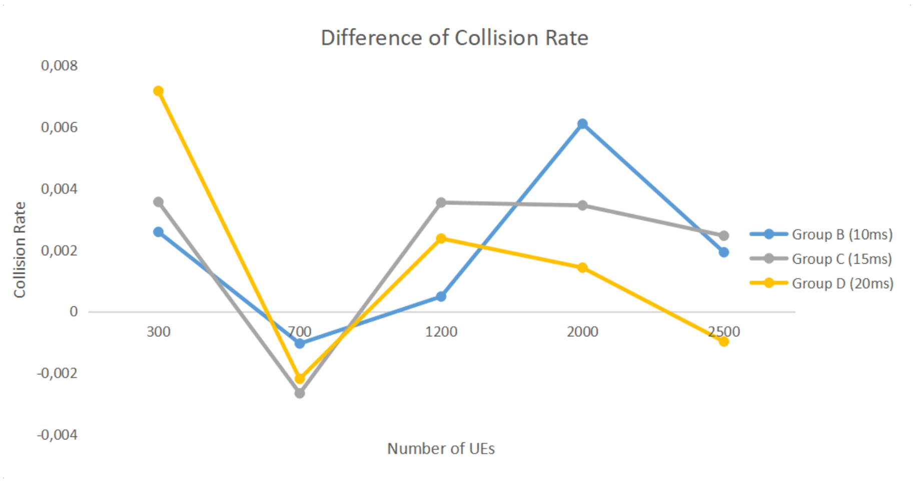


Fig. 4.15: This figure shows the collision rate for the different group of experiment while we increase the UEs.

First, we notice that the difference between the collision rate is in the range between -0.002 to 0.007. This is a small number since, in scenario 1, we seek to reduce the collisions and the average delay by increasing the UEs' period. Thus, it is reasonable not to have a big difference in the collision rate because we reduce the collision, but we also decrease the total measurements. Nevertheless, this is acceptable, as we assume in chapter 3, because some applications have tolerance in the decrease of the rate that receives data.

However, in figure 4.14 we see that as we increase the number of UEs more than 1000, the Group C and B are improving the RACH performance. Especially, group C (15 ms), displays the best and more stable performance from the other groups.

In figure 4.16, we present the Variability of the delay in the experiments (Group A-D), which is the fraction of the average delay and the standard deviation of the average delay. We observe that the variability is approximately the same for all the experiments. So we conclude that the scenario 1, with a threshold of 15ms ameliorates the RACH procedure, even more by increasing the number of UEs above 1200.

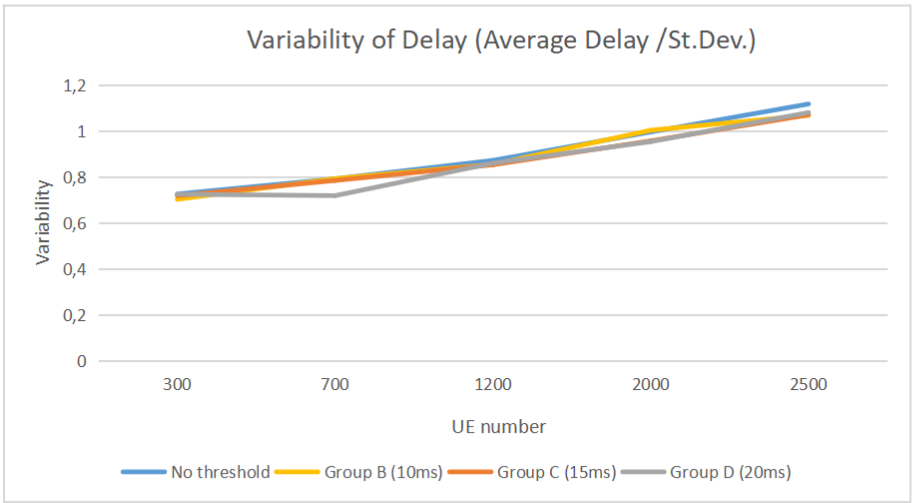
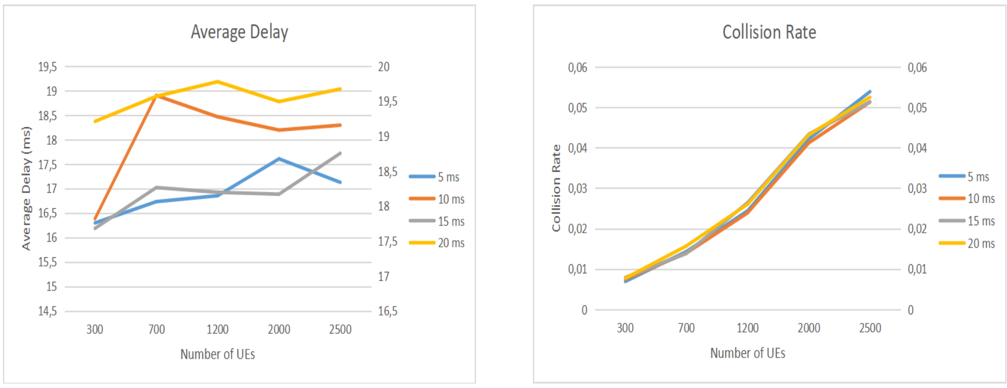


Fig. 4.16: This figure shows the variability for the different group of experiment while we increase the UEs.

4.2.4 Case 2 - Gateways

First, in Case 2 we run our simulator with tuning parameters the number of UEs and the gathering period, in order to see how the average delay and the collision rate are fluctuating. In figure 4.17, we present the results, while we change the gathering period of the gateways to 5,10,15 and 20 ms.



(a) Average delay

(b) Collision Rate

Fig. 4.17: Sub-figure (a) presents the average delays of different gathering periods, while we increase the number of UEs and sub-figure (b) presents the collision rate of different gathering periods, while we increase the number of UEs (Percentage of periodic UEs 0.5).

We observe that as we increase the gathering period, the average delay is increasing respectively. The reason is that the gathering period is the actual period that the measurements are sending. When a measurement is produced, the device is first sending it to the gateway, and the gateway will send it to the network, after the gathering period time has passed.

Thus, in a higher gathering time, the measurements will be sent to the network more sparse than in a lower gathering time.

Moreover, we notice that the collision rate is similar, between the gathering periods, which means that this Case, has a tolerance in the increasing number of UEs. This is reasonable, because the gateways are those that are requesting in the wireless channel. So, if the number of UE that each team can host is 3, for example, and the total UEs are 1200, then only $\frac{1200}{3} = 400$ devices (gateways) will interact with the wireless channel.

Next, we tune the number of UEs in each team. Figure 4.18 presents how the average delay is varying, while we change the number of UEs in each team.

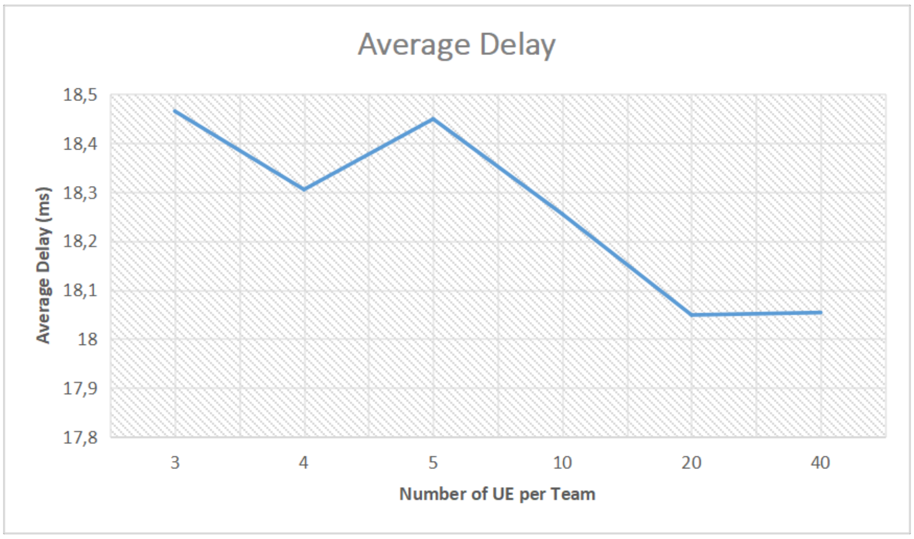


Fig. 4.18: This figure shows the fluctuation of the average delay as the number of UEs in each team. is increased. The total number of UE is 1200 and the gathering period is 10 ms.

As the number of UEs in each team is increasing, the average delay has an downward trend. Indeed, for 20 UEs in each team and more, the average delay remains the same. This result, shows that the system in the Case 2, is tolerant in the total number of UEs, but at the same time the average delay is much higher from the other Cases.

In figure 4.19, we choose 20 UEs per team to observe the tolerance of the system to a large number of total UEs. Indeed, the average delay is not increasing dramatically as we increase the total number of UEs to 5000.

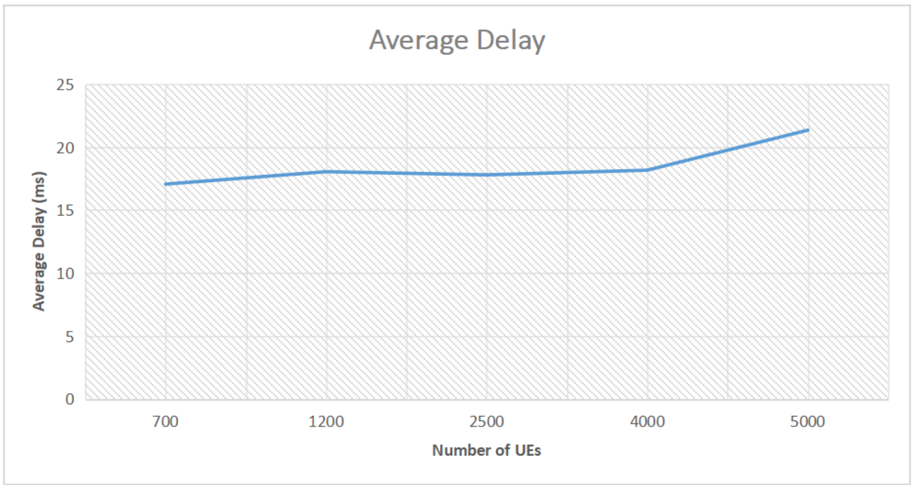


Fig. 4.19: This figure shows the fluctuation of the average delay as the number of total UEs in each team is increased. The number of UEs in each team is 20 and the gathering period is 10.

Finally, we compare the Case 2 with the Normal Case. In figure 4.20, we present the results. Although the average delay of Case 2, is much higher from Normal Case, the tolerance of Case 2 in a big number of UEs makes it a good strategy for a Massive IoT environment.

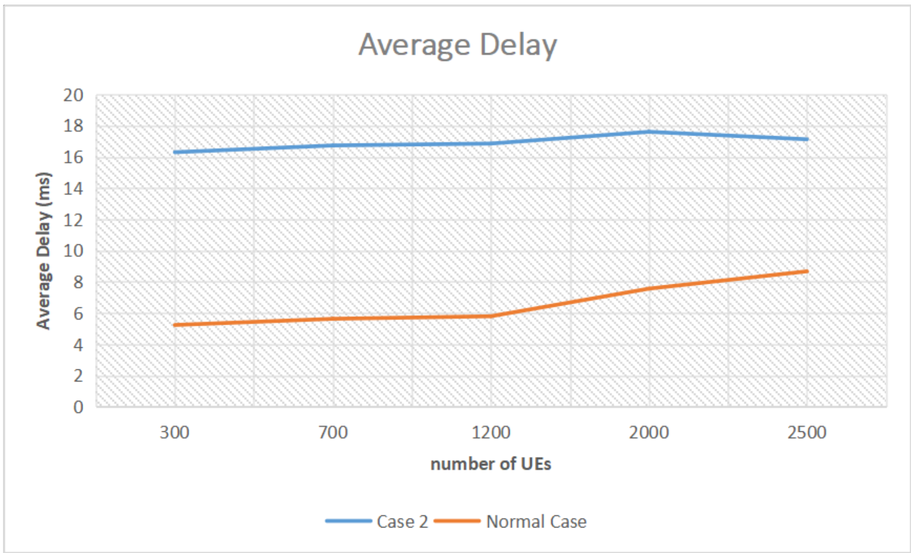


Fig. 4.20: This figure shows the average delay of Normal Case and Case 2.

5

Conclusions

As the number of devices increases, the RACH procedure encounters an overload issue, as described in chapter 2. To investigate this issue, we have built a Simulator, which reproduces the RACH procedure at the exact moment that a sensor generates a measurement until it is sent to the network. Furthermore, we use the basic idea of the AIMD algorithm applied by the TCP, to reduce the sensors' flow without evolving the lower layers. Also, we examine how we can reduce the flow using gateways. Both of these scenarios were implemented in the simulator.

The results revealed that as the number of devices is increasing, the delay is increasing respectively. The idea of the AIMD algorithm performs a little better than the normal RACH, while the gateways architecture almost triples the delay. Nevertheless, we figure out that if the number of UEs is increasing dramatically (5000 and more), the gateway architecture can keep the performance stable.

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

Bibliography

[AK16] O. Arouk and A. Ksentini. “General Model for RACH Procedure Performance Analysis”. In: *IEEE Communications Letters* 20.2 (2016), pp. 372–375 (cit. on p. 6).

[AKT16] O. Arouk, A. Ksentini, and T. Taleb. “How accurate is the RACH procedure model in LTE and LTE-A?” In: *2016 International Wireless Communications and Mobile Computing Conference (IWCMC)*. 2016, pp. 61–66 (cit. on p. 6).

[Bir+15] Andrea Biral, Marco Centenaro, Andrea Zanella, Lorenzo Vangelista, and Michele Zorzi. “The challenges of M2M massive access in wireless cellular networks”. In: *Digital Communications and Networks* 1.1 (2015), pp. 1–19 (cit. on p. 6).

[Erion] Erik Dahlaman, Stefan Parkvall, Johan Skold. *4G, LTE-Advanced Pro and The Road to 5G.* Elsevier Ltd., Third Edition (cit. on p. 3).

[Gila] David Gilbert. *JFreeChart (A free Java chart library)* `www.jfree.org` (cit. on p. 5).

[Gilb] David Gilbert. *LENA+ (a version of the LTE-EPC Network simulator extended with a realistic RACH model)* `https://github.com/signetlabdei/lena-plus` (cit. on p. 6).

[Pol+16] M. Polese, M. Centenaro, A. Zanella, and M. Zorzi. “M2M massive access in LTE: RACH performance evaluation in a Smart City scenario”. In: *2016 IEEE International Conference on Communications (ICC)*. 2016, pp. 1–6 (cit. on p. 6).

[PP14] A. Pourmoghadas and P. G. Poonacha. “Performance analysis of a machine-to-machine friendly MAC algorithm in LTE-advanced”. In: *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2014, pp. 99–105 (cit. on p. 6).

[vsi19] vsiris, fotiou, mertzianis15, polyzos. *Smart Application-aware IoT Data Collection.* Journal of Reliable Intelligent Environments, 2019 (cit. on p. 10).

[WCT12] C. Wei, R. Cheng, and S. Tsao. “Modeling and Estimation of One-Shot Random Access for Finite-User Multichannel Slotted ALOHA Systems”. In: *IEEE Communications Letters* 16.8 (2012), pp. 1196–1199 (cit. on p. 6).

[WCT13] C. Wei, R. Cheng, and S. Tsao. “Performance Analysis of Group Paging for Machine-Type Communications in LTE Networks”. In: *IEEE Transactions on Vehicular Technology* 62.7 (2013), pp. 3371–3382 (cit. on p. 6).

[Xie+19] J. Xie, W. Cheng, T. Zhang, et al. “Active and Adaptive Application-Level Flow Control for Latency Sensitive RPC Applications”. In: *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. 2019, pp. 352–359 (cit. on p. 6).

[YEZ15] Mohamed Yousef, Hussein Elsayed, and Abdelhalim Zekry. “Design and Simulation of Random Access Procedure in LTE”. In: *International Journal of Computer Applications* Volume 110 - Number 16 (Jan. 2015) (cit. on p. 6).

[Yur] Yuriy-g. *Simple Java Plot* `http://yuriy-g.github.io/simple-java-plot/` (cit. on p. 5).

List of Acronyms

UE	User Equipment
eNodeB	Evolved Node B
RACH	Random Access CHannel procedure
TCP	Transmission Control Protocol
RPC	Remote Procedure Call

ATHENS UNIVERSITY OF ECONOMICS & BUSINESS
ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

List of Figures