TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Department of Computer Science, Institute of Systems Architecture

Chair of Computer Networks

Minor Thesis

A Test Infrastructure for DTN Software Implementations

Thomas Hareau

Born on: 2nd May 1994 in Alençon
Matriculation number: 5481354
Matriculation year: 2016

5th May 2017

Supervisor

Dr.–Ing. M. Feldmann

Supervising professor

Prof. Dr. A. Schill

Contents

1. Introduction

Creating a communication network using satellites may be useful, for instance to bring the Internet to places where traditional facilities are too difficult to install. In this case, an efficient solution is the Ring Road approach.

The software µPCN was created for this purpose. It is designed to be embedded in the low-cost satellites CubeSats. It also provides a Delay Tolerant Network (DTN, or Disruption Tolerant Network) implementation.

1.1. Motivation of the thesis

The µPCN project is in development since 2014. Nowadays, unitary tests exist, which aim to check if the standalone feature is correctly implemented.

However, the current testing environment does not check the behavior of the software in a high ring road environment. The simulator *The ONE* (The Opportunistic Network Environment) can be used for these tests. However, it has to be modified in order to be able to communicate with the µPCN instances.

1.2. Goals of the thesis

For this purpose, two goals stand out.

- Creating a backend infrastructure to be able to launch tests in a ring road environment. Indeed, such an infrastructure, which simulates the ring road scenario and directly tests a DTN implementation, does not exist.
- Being scalable enough to simulate a realistic ring road scenario.

1.3. Structure of the thesis

This report will present modifications done in µPCN and in The ONE, and the implementation of the lifecycle manager. It will be structured as follows: Chapter 2 will present the different concepts and software used in this report. Chapter 3 will present the main goals of the project. Chapter 4 will describe the chosen solution, and chapter 5 its implementation. Finally the chapter 6 will test and evaluate the provided solution.

2. Background and State-of-the-art

This part will describe the Ring Road Approach(2.1) and present other tools that also implement this concept. The software µPCN will be described (2.2). Then, (2.3) will present a simulator for DTN software. Finally an analysis of the state of the art will be done in 2.4.

2.1. The Ring Road Approach

Some locations may be in such a geographical position that providing an Internet connection there may be very difficult or very expensive. In these cases, it could be more interesting to use a constellation of a few hundreds of LEO (Low Earth Orbit) satellites, which have the ability to communicate with earth. Thus the location could be connected with this constellation, which is itself connected to the global network, and then benefit from an Internet connection. This approach was presented in [1] and is named the Ring Road Approach*.

The satellite *CubeSat** [2] can be used for this approach. Indeed, it is a low-cost solution since it has a small size and is light. Such a satellite can be considered as secondary payload from different launchers. It can, for example, be launched from the International Space Station.

However, the current communication protocols are not in compliance with this approach. Indeed, it does not behave well in case of cut-off during the communication. Taking the example of the TCP protocol, a communication can only be established if the participants are interconnected. With the Ring Road approach, to deliver a permanent connection, each satellite should be connected to all the subnetworks[1] at the same time, which is not possible. The DTN* (Delay Tolerant Network, presented in [3]) protocol aims to provide a new protocol, which supports cut-offs. The core idea is the following: the node of the network shall be able to store the data that needs to be transmitted until a connection with another node is established. This node will do the same and the data will advance through the network until it reaches the destination.

To connect different subnetworks, satellites are used as "data-mules". Each time a satellite is accessible from the subnetwork, a connection is established. The satellite delivers the packets to the subnetwork, and receives other packets to transmit. The satellite will continue its movement, and will transmit the received data to the intended subnetwork. It can also deliver the packets to a subnetwork which is not linked with the final destination, but which will be reached by another satellite, which will be able to complete the transmission.

The Bundle Protocol* (BP) is an exchange protocol which aims to be used in a DTN, presented in [4]. It provides a specific message format to transfer data through a delay tolerant network. For instance, it contains the name of the sender, or the date of the creation. This data cannot be computed because the message may come from a different node than the creator. Other data

[1] In this Ring Road approach, a subnetwork is a network which is not connected to the whole network (in a permanent way, without taking into account the constellation).

allows the recipient to know if the bundle is still valid. The Bundle Protocol also provides a lot of possibilities to reduce the size of a bundle.

µPCN is a software for DTN communications that uses the Bundle protocol and that was developed to be used on a CubeSat.

2.2. The Micro Planetary Communication Network

The Micro Planetary Communication Network (µPCN, presented in [5]) is a light-weight and free implementation of DTN protocol. It can be used on POSIX operating system and also on the ARM Cortex STM32F4 microcontroller series (the micro controller embedded in the CubeSats). It will be tested in space, on the European Space Agency OPS-SAT, a version of CubeSat.

The main goal of µPCN is to provide a routing approach, in order to be able to forward the messages from the source to the destination.

The messages use the BP, in order to be in compliance with a DTN usage.

The µPCN also provides an implementation of the DTN IP Neighbor Discovery. This concept, presented in [6], provides a way for µPCN nodes to discover their neighbor, and thus to determine the best node to give the bundle to.

Before using this software in production, a solid test procedure shall be settled.

2.3. A simulator to test DTN communication

Some unit tests already exist in µPCN, however its behavior in a network has not been tested. The simulation shall be able to create a ring road approach scenario, and use instances of µPCN to manage messages.

Some DTN simulators exist, as *OMNET++* (see [7]), which is in C++, or *DTNSim2* which is not developed anymore. The project of The ONE simulator is the project which was given as prerequisite to test µPCN in a ring road scenario. It is written in JAVA, and is easily extendable.

2.3.1. The ONE

Presented in [8], The ONE, which stands for The Opportunistic Network Environment, is a simulator which is usable to create a DTN network. It simulates an environment populated with nodes. It can generate the movement of nodes following a provided configuration. Then it can route messages between nodes, following several routing algorithms.

However, The ONE does not support a Ring Road scenario. Also, satellite nodes are not fully supported. Thus, Riccardo Böhm modified the project to include this type of scenario, and to add satellite nodes. His work can be found in [9].

As said before, the goal of µPCN is to route the messages to the accurate groundstation. Thus, the routing part of The ONE shall be replaced by µPCN.

2.4. State of the art

Before speaking of simulation environments, some BP and DTN implementation, as µPCN, have to be presented.

DTN2, presented in [10] implements DTN and the BP. It is developed by the IETF DTN research group and aims to be the implementation of reference. Thus, it is not designed for performances, but rather to be as clear and pedagogic as possible.

On the other side, *ION* [11] is a more robust solution. It is developed by the Jet Propulsion Laboratory (JPL). It is available in numerous distributions (GNU/Linux, Solaris, MacOS, ...), and has also been tested in deep space during the DINET experiment.

Simulation environments that are capable of evaluating a DTN and BP implementation are rare but exist.

DTNperf is an evaluation tool presented in [12] and designed to provide logging and goodput for a BP implementation. The tool is usable through the command line interaction (CLI). The software is separated in three standalone operating modes. Firstly, the client creates and sends bundles. Several clients can coexist. Secondly, the unique server receives the bundles. Thirdly, the monitor receives status reports and control bundles. To manage the bundle, the software uses either *DTN2* or *ION*, however, DTNperf has been developed to be independent of the bundle protocol implementation in use. Thus, it can be extended easily to use µPCN. The software aims to provide information about the different phases of the communication. The monitor can be used to create a CSV file, which sums up the different information.

Using DTNperf with µPCN would not be useful. Indeed, the tool is not designed to test the behavior of the BP implementation in a ring road environment. Also, it does not seem to be possible to extend the software and to create this behavior, as every instance has to be controlled from the CLI. Furthermore, only one server can be created, which makes the transition of a bundle through several nodes impossible. The simulator The ONE can be extended for this usage.

The simulator The ONE is compliant with a bigger scenario. It can natively be connected with DTN2. As said earlier, the simulator can be used to test BP implementations. However, the ring road scenario is not natively supported but was implemented in [9].

The connection between The ONE and DTN2 is done by creating a connection with The ONE. To do so, some daemons of DTN2 are created before launching The ONE. The configuration file indicates which nodes shall be connected to which instance. Then the messages received from a node are managed by the instances of DTN2.

Even if this possibility is functional, it is not suitable for usage. Indeed, this concept is not scalable, since the instance shall be manually created. Furthermore, the instance in use per one node shall also be manually set into the configuration.

The solution provided in this report manages the instances' lifecycle in a more dynamic way. The instances can be automatically created by The ONE, making the constitution of the scenario more flexible. Indeed, changing the DTN implementation would become easier.

Thus, The ONE cannot be used as such, because the creation and the usage of the instances of the DTN software are too dependent on the scenario. It would be better for the simulator to create each instance on the fly, and match it with the node. As far as we know, such an implementation does not exist. The goal of this project will be to create a test infrastructure which supports this facility. To be reusable, The ONE will use an external software, which manages the lifecycle of the different instances of µPCN.

3. Requirements

Assuming the simulator is already created, a specific software, called lifecycle manager, has to manage the instances of µPCN. The following example can illustrate the different use cases of the software:

A project manager has to design a solution to exchange data between several nodes of a network. To do so, he has to choose between two DTN software (**A** and **B**), and has to select the solution which consumes the least amount of data. To test this characteristic, he can simulate the network with a specific tool, which uses a lifecycle manager that controls the instances of the tested software. He intends to run two simulations, one with each software. He configures the lifecycle manager to be able to retrieve the data consumption of the instance: if **A** provides a direct command to query it, **B** does not, and the data should directly be queried from the OS. Finally, after the simulations have ended, he requests the data consumption from the lifecycle manager.

At the end of the test, it appears that **B** uses less data. The further customer query is an estimation of the energy usage while in production. To compute these data, the instances of **B** have to be launched directly on a real machine, and a test has to be launched on it. So, the project manager decides to use the same simulation, but not to launch the instances on the local machine but rather to use SSH to create the instances on the real machine. He also changes the configuration in order to access the energy consumption.

Analyzing these use cases, the following requirements can be extracted.

Requirement /R1/ Managing instances: The lifecycle manager shall be able to create new instances of µPCN. There are two ways to create them: either by creating only one instance, either by creating several instances in one command.

The user shall be able to retrieve communication handlers about each already created instance.

The lifecycle manager shall also be able to kill the already created instances.

Requirement /R2/ Getting statistical information: The user shall be able to retrieve statistical information for every instance.

Requirement /R3/ Supporting remote SSH machines: The lifecycle manager shall be able to create new instances on the local machine or on remote machines. It shall be able to choose by itself the ideal machine to create the new instance when the user asks for it.

The user shall be able to specify the best strategy to create instances.

Requirement /R4/ Being scalable: The lifecycle manager shall be scalable enough to be used in a ring road environment simulation.

Requirement /R5/ Having a extendable configuration: The lifecycle manager shall be highly configurable. The user may change the address and default port of µPCN. S/He can also change

the log directory, and decide whether the created instances shall be killed after exiting the software. The list of SSH machines shall be editable, and the IP, the port, the user and the priority shall be modifiable. The statistical information list shall also be extended, the user shall be able to specify a description, the command and the type of the result for each item.

Requirement /R6/ Being independant from the usage: The lifecycle manager shall be independent of any external tool. It shall be usable in any type of usage (embedded in a simulator, in a website, in command line, ...). The DTN software may also be changed.

4. Concept

The lifecycle manager is a software which manages the lifecycle of another software. From the point of view of the user, there are two ways of using the lifecycle manager. Either s/he can connect it to a simulator, which aims to test some features of the target software, or s/he can use it as a standalone project. S/he may combine both, connecting the lifecycle manager to the simulator, and interacting with it at the same time (for instance to acquire the statistical information of the /R2/).

4.1. Using the lifecycle manager with a simulator

The goal of this project is to connect a software to a simulator, using a lifecycle manager. The figure 4.1 is a representation of the overall result. The next part of this section will give an overview of the different components of this figure.

4.1.1. The different components

The figure 4.1 exposes three different components: the managed software, the simulator and the lifecycle manager. The requirement to use a software as one of these components will be described in this part.

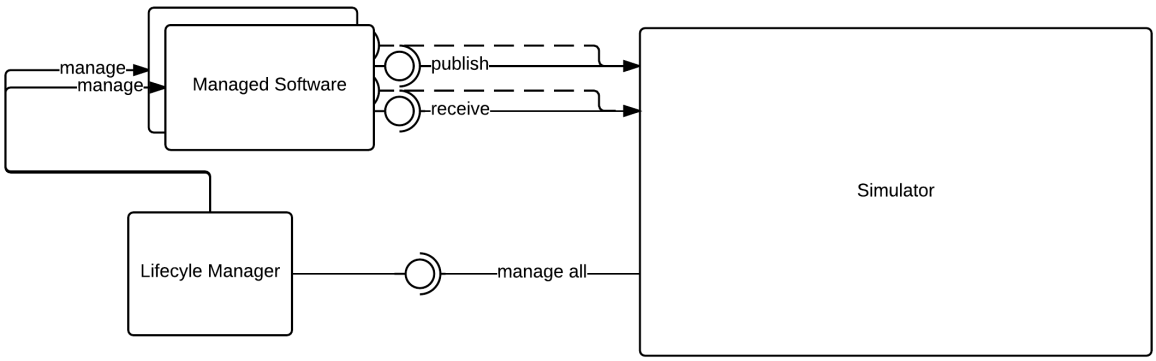

Figure 4.1.: aimed architecture

4.1.1.1. The simulator

The simulator aims to test the behavior of a software. To be used in this project, the use cases presented in the diagram 4.2 should be completed. The component may be able to ask the lifecycle manager to create and destroy the software, and may communicate with the software using two communication managers, one to publish and one to receive.

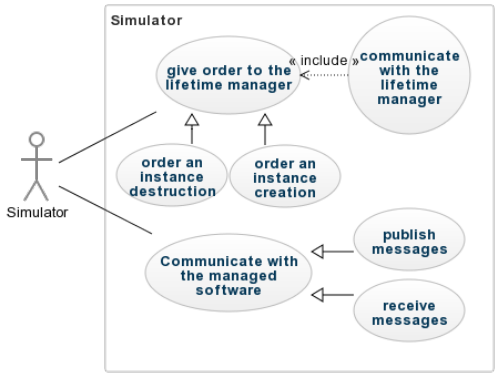

Figure 4.2.: Use case diagram of the simulator

4.1.1.2. The managed software: the couple µPCN–netconnect

To be managed by the lifecycle manager and used by the simulator, the software needs to comply to the use cases of the diagram 4.3. Its lifecycle shall be manageable by an external software, and it should be launched several times on the same machine. It shall provide two communication managers: one to receive messages and one to publish messages.

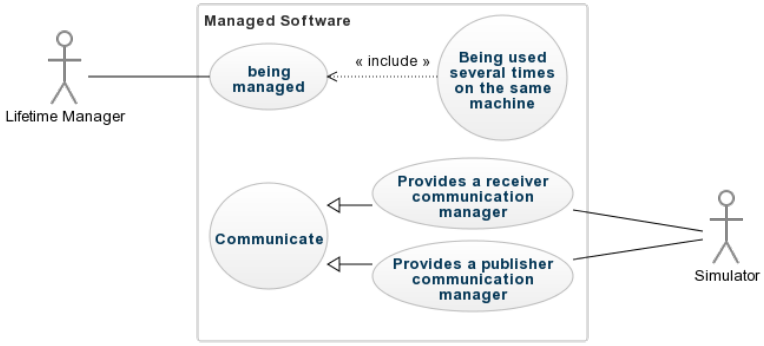

Figure 4.3.: Use case diagram of the managed software

4.1.1.3. The lifecycle manager

The lifecycle manager shall manage the lifecycle of the instance of the managed software. It shall also provide a communication manager to receive orders. No use case diagram will be shown here, because the requirements which are presented here are only a part of the final requirement. The full use case diagram can be found in the section 4.2.

4.1.2. Interaction between components

In the architecture described by the figure 4.1, the simulator is the first software to be launched, and it controls the lifecycle manager. The lifecycle of the simulator can be separate in three different steps:

- the initialization of the simulator (figure 4.4). It happens after the user launches the simulator, or may also take place anytime a new instance of the software is needed. During this step, the simulator shall order the creation of an instance from the lifecycle manager. Once the instance is acquired, it may initialize the instance by communicating with it;

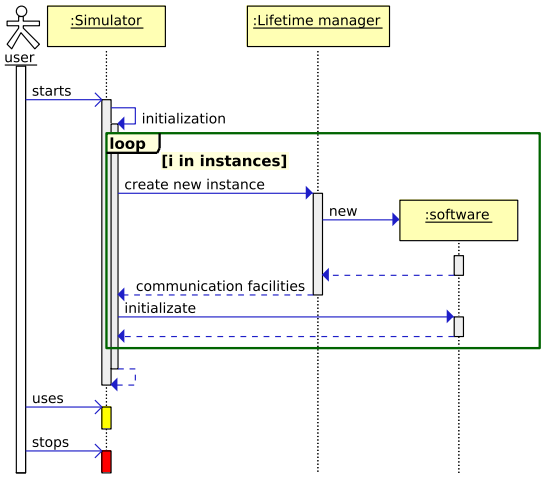

Figure 4.4.: Initialization of the simulator

- the utilization of the instances (figure 4.5). This step appears when the simulator needs to interact with the instance. At that time, it may communicate with the instance, using the two communication facilities described in 4.1.1.2.

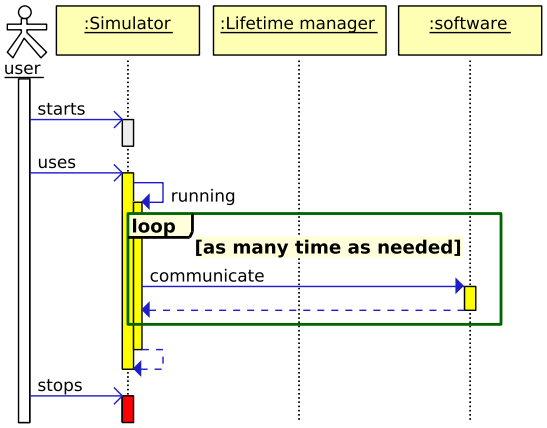

Figure 4.5.: Utilization of the simulator

- the extinction of the simulator (figure 4.6). This stage occurs when the utilization of the simulator has ended, or if some instances are not needed anymore. The simulator will then ask the lifecycle manager to delete the concerned instances.

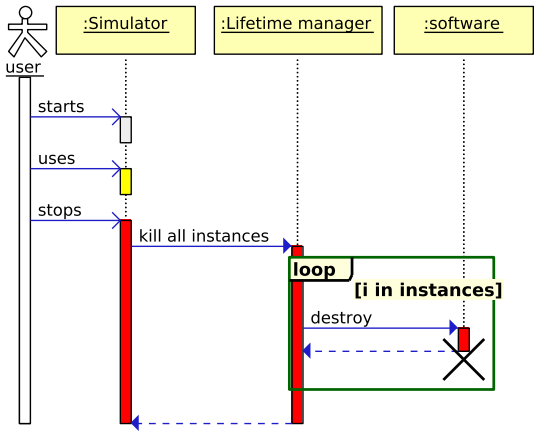

Figure 4.6.: Extinction of the simulator

4.2. Directly interacting with the lifecycle manager

The lifecycle manager may easily be controlled by a simulator, but it shall deliver other services, in particular those defined in /R2/. It can also be used in a standalone way, if a user needs a lot of instances for example.

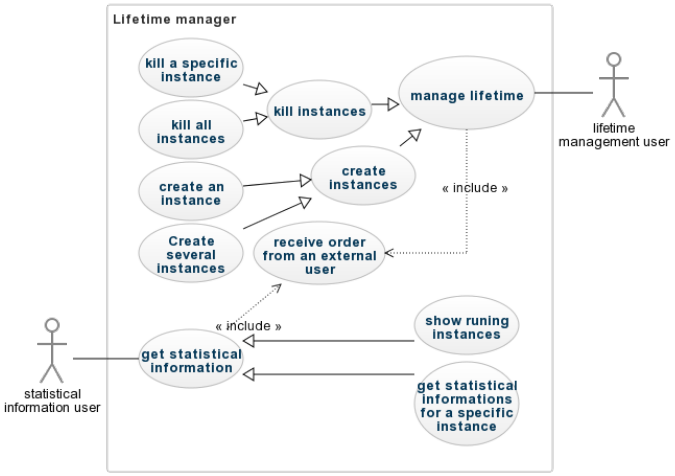

Figure 4.7.: The full use case of the lifecycle manager

The use case diagram in figure 4.7 presents the different possibilities. This diagram also presents some requirements defined in the part 4.1.1.3. In fact, the actor *"lifecycle management user"* may be the simulator.

The use case "manage lifecycle" is separated in two. Each time, the actor shall use a single action, or a multiple one. S/he shall be able to create as many instances as s/he wishes on the same time. S/he may kill one specific instance or kill all running instances.

Concerning the statistical aspect, two use cases are possible. The user has the possibility to get the list of all running instances. S/he can also acquire the statistical information defined in /R2/.

5. Implementation

In chapter 4, three modules were defined: the simulator, the managed software and the lifecycle manager. For the implementation, the software The ONE is used as the simulator. The managed software is µPCN associated with netconnect, which transforms the unique communication manager of µPCN into one publisher and one receiver. No available software exists for the lifecycle manager. Thus it will be created from stack.

Two variants of µPCN exist: one is compatible with the POSIX system, and the other is compatible with STM32 system. STM32 is the 32 bit micro-controller which is on-boarded in CubeSat satellites. POSIX is a type of operating system. Most Linux distributions are POSIX compliant. The tool netconnect is mainly provided in order to be used on a Linux machine.

On the other hand, The ONE is a software developed in JAVA, and requires a graphical interface. So it can also be used in every Linux-based operating system.

The remote controller should be compatible with µPCN, netconnect and The ONE, thus, the software needs a Linux-based software. The development has been done in *Fedora*, but shall also be compatible with most Linux systems.

Before implementing the lifecycle manager, µPCN shall be modified a little. Indeed, the POSIX variant was not created to be used more than once on the same computer.

The ONE cannot be directly used either, because it was not designed to communicate with the lifecycle manager.

This section will be devoted to the minor changes on µPCN (5.1), present the modification done in The ONE (5.2), describe the remote controller development (5.3). If needed, a usage guide can be found in the appendix B.

5.1. Changes in µPCN

As said above, it is not possible to use several instances of µPCN on the same machine at the same time. A TCP socket is created, to allow users to communicate with the software. This socket is bounded to the port 4200 on the local machine. This creates two issues.

- Trying to create a second instance on the same machine will fail. µPCN fails while trying to create a new socket on the port 4200, which is already used by the first instance.
- Assuming that creating several sockets on the same port is possible, this is still problematic. Indeed, it would not be possible to communicate with a specific instance. This would break some requirements (/R1/, /R2/).

Thus, µPCN shall be modified to allow the user to specify the socket port when a new instance is launched. This modification should not modify the normal behavior.

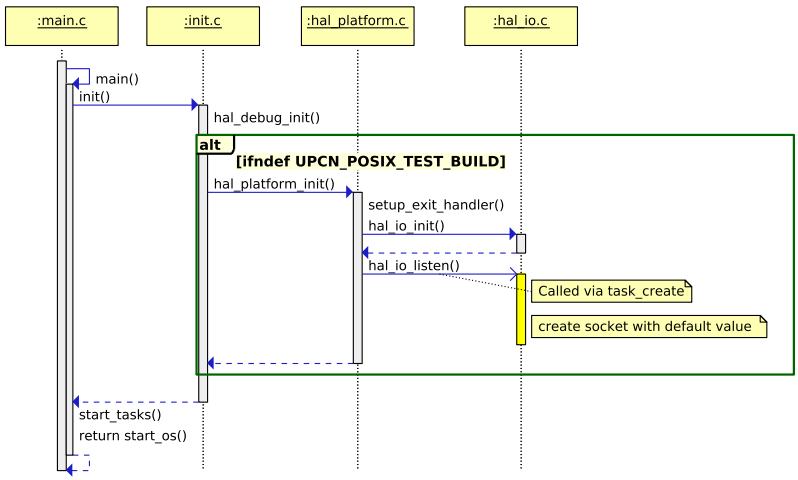

Figure 5.1.: former µPCN socket creation procedure

The usage of µPCN will be the following: the port will be specified as first argument of the main function of µPCN. Calling it in bash will be: `upcn <port>` where *<port>* is the wanted port. If no port is specified, the default value (4200) will be chosen.

The procedure for the creation of a socket is described in the figure 5.1. This shows that only four files are used to create the socket. So the modification will be very small. The final result is illustrated in the figure 5.2.

main.c[1]

This file contains the main function of µPCN. The work here is to retrieve the arguments, if there are any, and to convert them from string to integer (`uint_16_t`). To do so, the function strtoimax, provided by `<inttypes.h>`, was used. If no argument is provided, the default value is set to 0.

The call to *init* was modified, by adding a parameter to the function, which is the provided by the socket port.

init.c[2]

This file is used to initialize µPCN. The prototype of this function was modified to accept one argument, which is the port to use. The header of this file was also modified.

In the corpus of the function `init`, only one part is interesting for this functionality: the call of `hal_platform_init`. The socket port was forwarded in this call.

hal_platform.c[3]

The function `hal_platform_init` in this file is used to initialize the hal platform. It contains a call to a global initialization function and an order of asynchronous call to a socket management

[1] This file is located in `components/upcn/src/main.c`
[2] This file is located in `/components/upcn/src/init.c`
[3] This file is located in `/components/hal/src/posix/hal_platform.c`

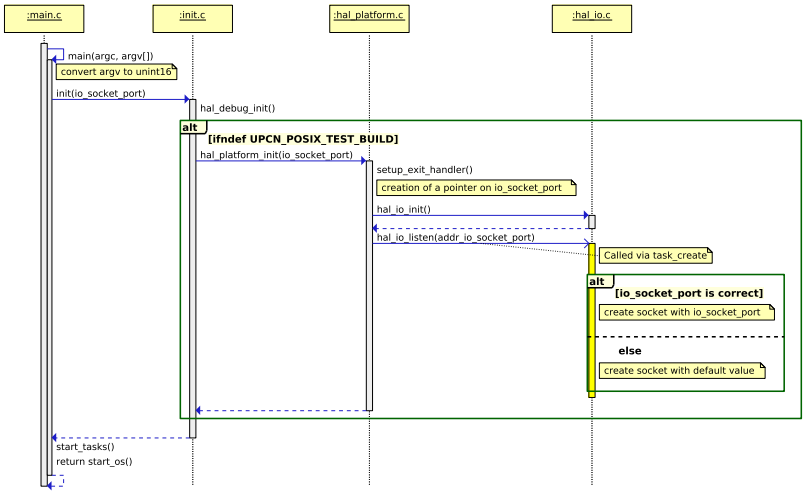

Figure 5.2.: actual µPCN socket creation procedure

function (`hal_io_listen`). The asynchronous call consists of a call to `task_create` with a pointer of the desired function. The possible argument of the function shall be given as a void pointer to the `task_create` function.

Once again the prototype of the function was updated to add the parameter `io_socket_port`. The header file was modified, but also the stm32 variant of this file, to fit with the header. Nothing was done with this parameter in the stm32 variant of the file.

In the corpus of the function, a pointer on `io_socket_port` was created. Indeed, this will be an argument of the function `hal_io_listen`, which is called in an asynchronous way. This pointer was added as a parameter of the function `task_create`. The pointer will be free in the function `hal_io_listen`.

hal_io.c[4]

Only the function `hal_io_listen` is modified in this file. It manages the socket creation, and ends by listening infinitely to the socket.

The same modifications as in `hal_platform.c` are done on the prototype: adding an argument, and adding those in the header and the stm32 variant of the function. Once again, this argument will not be used in this function.

In this function the socket is created. So it is in this function that the argument will be used. As described in paragraph `hal_platform`, the argument is the pointer to an integer, which represents the socket port to use. The default value of the variable is 0. In this case, the default value hard-coded in µPCN will be used to create the socket. Otherwise the socket will be created using the variable as port. Every other steps during the creation of the socket are not been modified.

Once the variable is not needed anymore, the last step is to free the pointer. This is done just before the beginning of the listening phase.

[4] This file is located in `/components/hal/src/posix/hal_io.c`

5.2. The ONE implementation

The component of the simulator was defined in the part 4.1.1.1. The project of The ONE simulator with the the Ring Road approach integration can be used for this purpose, however this implementation does not perfectly suits the requirement.

It natively has some facilities to communicate with external sockets, however this facility is not compliant with the provided socket of µPCN. One of the biggest lacks is the impossibility to transmit a bundle to µPCN.

Second weakness, it is not able to create instances on the fly, or to give orders to an external lifecycle manager.

Thus, The ONE shall be modified to include these features. Three steps are needed. Firstly, the communication with the lifecycle manager and with µPCN shall be established. Secondly, the serialization and the deserialization of a bundle shall be implemented. Thirdly, a special class of The ONE shall be created, in order to use the instance of µPCN in The ONE. The two first steps take placed are independent of The ONE, thus is in a separated package, and can be transformed into a library. The last step is highly dependent of The ONE, and takes place directly in the core of the software.

The implementation of this component was done with the help of Olivier De Jonckère.

5.2.1. The architecture of the solution

This part will present the architecture of the solutions which were developed. The class diagram in figure 5.3 shows the main classes of this architecture. The idea is to provide a prototype of integration. The goal is to be able to launch a simple simulation with only one contact at the same time for a satellite. The work is based one the version of The ONE presented in [9].

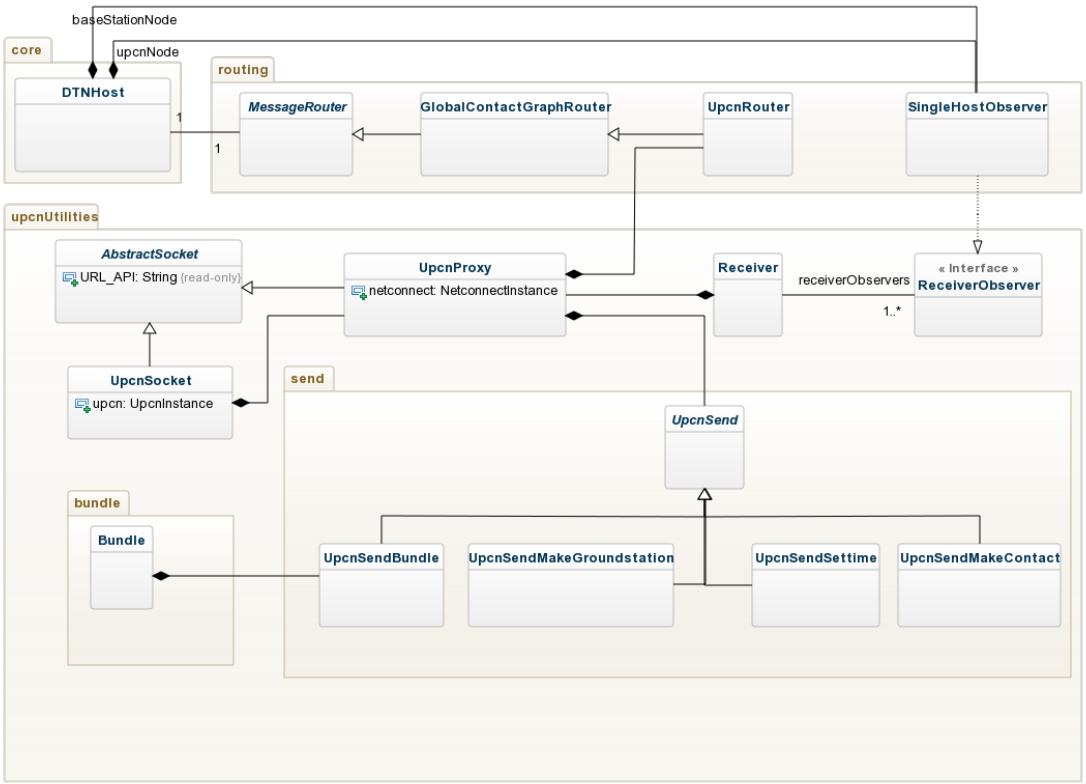

Figure 5.3.: Non exhaustive class diagram of The ONE

The user interface of The ONE simulator represents a map on which are placed different nodes. Two types of nodes exist: the network spots, which are immobile, and the satellites, in movement. When the satellite is in the area of a spot, contact can be established. During this contact, both node can exchange messages. In the provided solution, the satellites are backed with instances of µPCN.

In the core of The ONE, every node is represented by the class **DTNHost**. The difference between a spot and a satellite is mainly its speed. Each instance owns an instance of the (abstract) class **Router**, which manages the sent and received messages.

Two implementations of the class **Router** are used in this part of the project. Firstly the class **GlobalContactGraphRouter**, which has not been modified, is used to route messages without the integration of µPCN. This behavior is suitable for the spots, which do not have to be backed up by µPCN, thus this router is used for this type of nodes. In this configuration, the messages are stored by this class, and are forwarded to the recipient when a contact occurs.

Secondly, the class **UpcnRouter** is new. It extends the previous class, in order to back it up with µPCN. Then, the messages are not stored by this class[5], but rather forwarded to µPCN. Then, during the appropriate contact, µPCN will send the bundle. A listener in The ONE will retrieve this message and directly forward it to the suitable node. The development of this part will be discussed in the part 5.2.4.

The package **upcnUtilities** provides the facilities to interact with µPCN. **UpcnRouter** uses an instance of the class **UpcnProxy**. This class has several facilities. First, it manages the lifecycle of the instance of µPCN (contained in the class **UpcnSocket**) and netconnect. It can also send messages to µPCN, thus it can be used to send bundles and other commands. The messages use a specific format, thus some specific command (the class included into the package **upcnUtilities.send**) can be used to simplify the expedition of messages.

The reception of messages is more complex, thus an instance of the class **Receiver** is attached to **UpcnProxy**. It is responsible for retrieving the messages sent by µPCN. Observers can be attached to this class. On each event, they will be notified by the receiver. These observers shall implement the interface **ReceiverObserver**. One implementation exists: **SingleHostObserver**, which is used to forward the message to the accurate router.

The implementation of this package will be discussed more in details in the part 5.2.3.

The last package, **upcnUtilities.bundle**, is responsible for the serialization and deserialization of the bundle. Indeed, µPCN needs to receive it in a specific format, and then forward it with the same format. Also, the bundle is a complex data type, thus builders and observers are also provided to simplify its usage in The ONE. The part 5.2.2 will describe more in detail this package.

The package **upcnUtilities.bundle** is totally independent from the rest of the project, thus it is possible to use it in any other project. Likewise, the package **upcnUtilities** single dependency is the aforementioned package bundle, thus it is also possible to export this package in another project.

5.2.2. The management of the bundles

A bundle is a very specific data type, created to be compliant with a DTN. Its definition is quite complex, because a lot of facilities aim to reduce the length of the bundle.

The whole architecture can be found at the page 17 of [4]. The two main fields are the dictionary and the payload blocks. The dictionary aims to provide information as the source or the destination of the bundle. To avoid redundant information, the offset of the different fields are also provided. Thus, if two fields have the same value, it only has to be written once, and different offsets will have the same value. The payload, a set of bundle block, contains the data of the bundle.

[5] In some cases, µPCN is faster than The ONE. Then the messages are stored until The ONE has finished to compute them.

The address of a node is called an endpoint identifier (EID). It uses the following format: `scheme:ssp`, where *scheme* describes the type of communication (DTN, TCP, ...) , and scheme-specific part (SSP) is the identification of the node.

The remaining fields are provided to help the recipient check that the bundle is still valid. For instance the length and the lifetime is provided.

The different utilities provided in relation to the bundle are placed in the package **upcnUtilities.bundle**, and the class diagram 5.4 represents them. The following part will be separated into two sections: first the utilization of the bundle in The ONE will be discussed, then the serialization and the deserialization of the bundles will be examined.

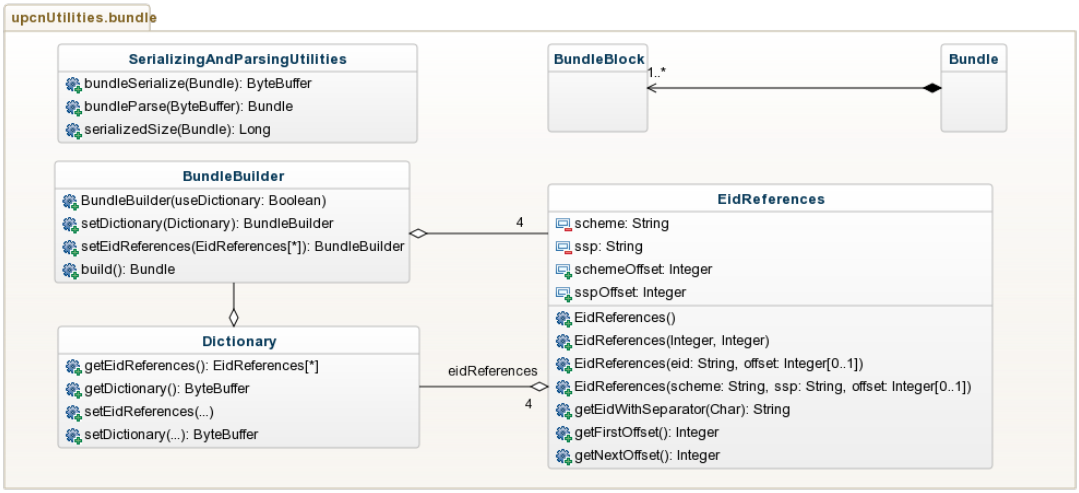

Figure 5.4.: The class diagram of the package upcnUtilities.bundle

The list of the method is non exhaustive

5.2.2.1. The utilization of the bundle

For some reasons, presented in the part 5.2.2.2, the class **Bundle** contains a lot of attributes and most of them are very low-level. The best example is the dictionary: it is an optimized data structure to store EID. Nine fields are directly related to it: the offset of the different EIDs, and a ByteBuffer which stores the data. Thus, some helper classes were provided to simplify the utilization of µPCN.

The class Bundle: This class represents the bundle using primitive types, in order to be the closest with the bundle defined in µPCN. The class contains twenty-one parameters, including the definition of the dictionary (eight eid's offsets and a bytebuffer containing the data), the blocks (a list of **BundleBlock**, a pointer to the payload block, and information about the blocks as their size), and information about the bundle, as the date of the bundle's creation, the size of the bundle, ...

The important function in the bundle is the serialization, which simply calls the method of the class **SerializingAndParsingUtilities** (discussed in the next section) and the function `equals`, which tests whether another object is equal to this bundle.

The numerous fields of the bundle make it difficult to create and to use. Thus, the class **Dictionary** and **BundleBuilder** simplify its utilization. They will be presented in the following paragraphs.

The class BundleBlock: This class represents a block, i.e. a piece of data embedded in a Bundle.

The class EidReferences: This class represents an EID. It contains the scheme and the SSP, and their offset. Two usages are possible. Several constructors are available. They can be categorized in two types: the creation when knowing the value of EID and the creation when knowing only the offsets.

The constructors using the value of the EID are those who use as argument four strings. If two strings are provided, the first will be the scheme, and the second the SSP. If only one string is provided, the constructor will try to extract the scheme and the SSP. The offsets are automatically computed. The optional parameter *offset* can be used to add this value to both the offsets. The default constructor uses the value `dtn:none`

The constructor using the offsets is taking two integer arguments. This can be used when the EID is not known, this can happen when decoding a raw dictionary.

An instance of this class can be used to retrieve the lowest offset between both, and the offset of the next field in the dictionary.

The class Dictionary: This class can be used to represent a dictionary in a more usable manner than what the **Bundle** does. Originally, to get an EID, the user would read the dictionary field of the Bundle from the offset to the first null byte. Also, creating a dictionary can be fastidious, because the offsets have to be set manually.

Instead, the class **Dictionary** can be used to read fields from a bundle. To do so, the user initializes a Dictionary with an instance of **Bundle**. Then he/she/it will be able to retrieve the different EIDs thanks to the different getters provided by the class.

The class can also be used to construct a bundle from several EIDs: the user has to initialize the instance with the EID (as string or **EIDReference**) and then retrieve the ByteBuffer dictionary. The different EIDReferences will be created or updated, and the dictionary constructed.

The class BundleBuilder: This class follows the design pattern *builder*: it shall be used to create a bundle more easily. It provides a lot of setters which can be used to indicate which value to give when creating the bundle. Finally, the function build can be used to create the bundle.

Two types of utilization are possible: with the use of an instance of the class **Dictionary**, or without it. In the second case, the offsets and the raw dictionary have to be set up by hand. During the creation of the BundleBuilder, the user can indicate if he/she/it wants to use a dictionary or not. If he/she/it does not indicate this, the instance will check that the values are coherent before building the bundle.

5.2.2.2. The serialization and the deserialization

The serialization – transforming the bundle into a byte array – and the parsing – transforming a byte array into a buffer – is a complex part. Indeed, the serialized bundle shall be understandable by µPCN, and The ONE shall be able to decode a bundle sent by µPCN. Thus, the two implementations shall be exactly compliant. These methods could have been done in The ONE, but any modification in µPCN would imply to modify the simulator to keep the compliance. The best solution is to use directly the functions of µPCN in The ONE. Because µPCN is coded in C and The ONE in JAVA, a wrapper shall be implemented.

The class **SerializingAndParsingUtilities** and the shared object `bundleUtilities.so` are used to connect the JAVA and the C code. The first class is coded in JAVA. It is used to call the appropriate function in the shared object. The second is a C library, and is used to transform JAVA object to C data structure and vice versa.

The serialization of a bundle, which happens when a bundle has to be sent to µPCN, is presented in the figure 5.6. First the function serializes of the Bundle instance shall be called. It will directly call the function bundleSerialize of the class **SerializingAndParsingUtilities**, which itself calls the native method defined in `bundleUtilities.so`. Moving the call to the native method in a dedicated class avoids to pollute the library with useless definitions. The C library will then retrieve the fields of the java Bundle, and construct a C bundle. Then it will call the serialization function of µPCN with the new bundle. The result, a byte array, will be transformed into a JAVA ByteBuffer and then return to the JAVA function.

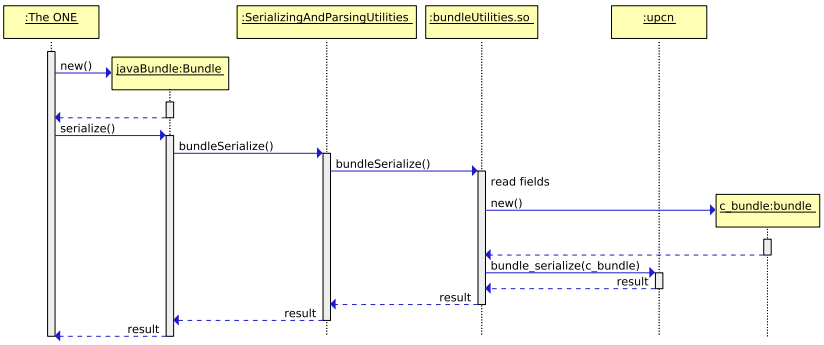

Figure 5.6.: The serialization of a bundle
The timeline `upcn` represents the functionalities included in µPCN.

When The ONE receives a message from µPCN, a ByteBuffer data has to be parsed to a Bundle. The figure 5.7 represents this procedure. Its entry point is the call to the function bundleParse of the class **SerializingAndParsingUtilities** with the ByteBuffer as argument. The same method is called in the shared object, which will call the bundle parser facility of µPCN. This will construct a C bundle, that the shared object will retrieve, translate into a JAVA object and return to the JAVA code.

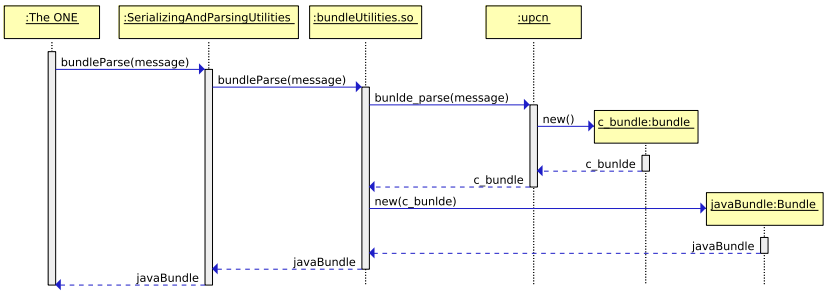

Figure 5.7.: The deserialization of a bundle
The timeline `upcn` represents the functionalities included in µPCN.

A special `Makefile` was created to construct the shared object `bundleUtilities.so`. Indeed, using the functions `bundle_serialize` and `bundle_parse` implies having access to the dependencies of their files, thus some part of the code of µPCN has to be included in the library, which is done thanks to this file.

5.2.3. The management of the instances of µPCN

To integrate µPCN in The ONE, different actions with µPCN are needed. The first action is the management of the lifecycle of the instances. For this, the lifecycle manager server is used, this part will show how it was integrated. The next action is communicating with the instances. For this, the ZeroMQ sockets provided by netconnect are used. The second section of this part will concern this communication aspect.

5.2.3.1. Managing the lifecycle of the instances

The management of the lifecycle of the instances takes two parts: creating the instance and killing the instance. Two softwares have to be managed: µPCN but also netconnect which takes as parameter an instance of µPCN. Thus, two different lifecycle managers are used.

The classes involved in this part of the project are shown in the figure 5.8. The list of functions is not exhaustive: neither the testing utilities nor the communication utilities with the instance are displayed. The class **UpcnSocket** contains the instance of µPCN, and the class **UpcnProxy** can be used to communicate with netconnect. The class **AbstractSocket** is used to communicate with a specific lifecycle manager.

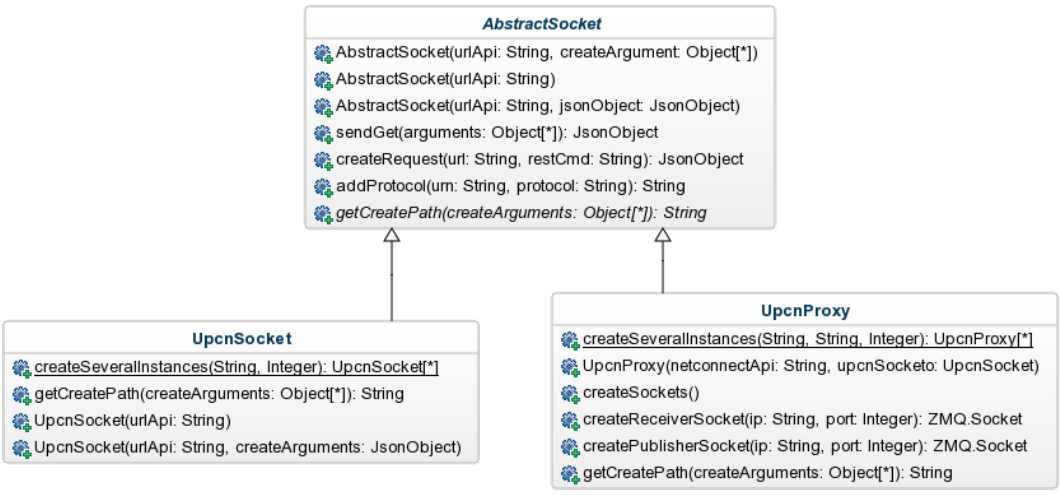

Figure 5.8.: The class diagram of the socket acquire from µPCN

Creating the instance: Different constructors are provided for the class **AbstractSocket**. Those which do not take a JsonObject as argument call the appropriate lifecycle manager to create the instance, whereas the other assume the instance is already existing.

The call to the first type of these constructors is described in the figure 5.9.

The first main step is the acquirement of the resource identifier to call the lifecycle manager. The abstract method getCreatePath, which shall be overridden by the child classes, aims to retrieve this resource. It accepts a list of objects as arguments, which can be used to create the resource. In the case of **UpcnProxy**, the associated instance of the class **UpcnSocket** shall be given using this parameter.

Then, the call to the function createRequest with this resource can be done. It will send a `GET` request to the lifecycle manager, and parse the result into the JsonObject.

The last constructor may be used when several instances are created at the same time. In this case, the lifecycle manager returns a list of JsonObject. Each element of the list can be directly used with the last constructor.

In every case, the JsonObject is used to retrieve information about the created instance. The constructor of **UpcnSocket** retrieves from it the port and the IP of the created instance, whereas the constructor of **UpcnProxy** retrieves the IPs and ports of the receiver socket and publisher socket.

On top of that, the constructor of **UpcnProxy** has also to initialize the communication tools. The two ZeroMQ sockets are created with the retrieved swapped IP and port (the subscriber of netconnect is the publisher of The ONE, and vice-versa). The **Receiver** is also created, it uses the

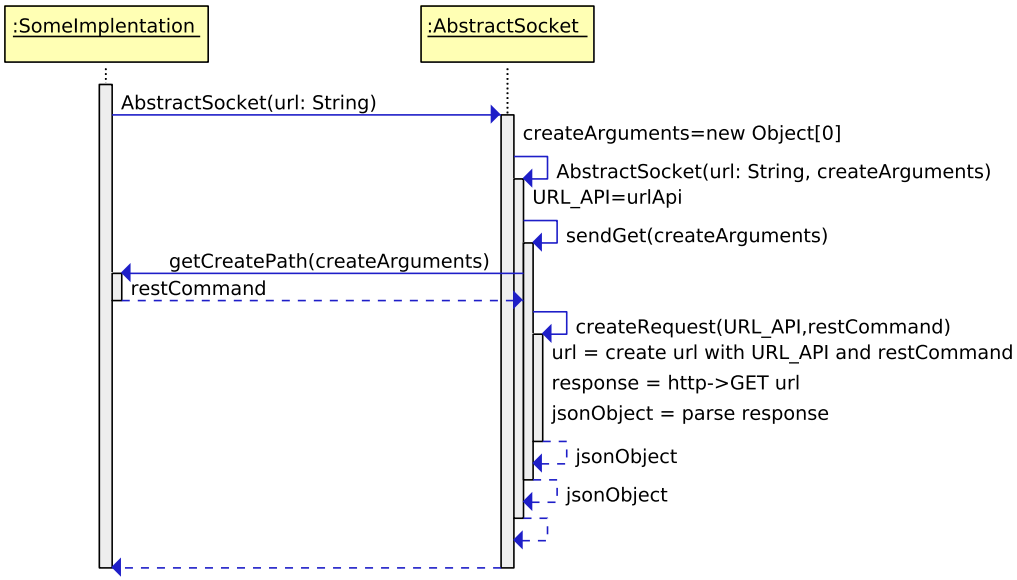

Figure 5.9.: The creation of an instance thanks to the AbstractSocket constructors

subscriber socket to do so. The utilization of these sockets and classes will be discussed in the part 5.2.3.2.

Killing the instance: Killing an instance of **UpcnSocket** is quite simple. The appropriate URL to kill the instance is computed and sent to the appropriate lifecycle manager. This procedure is defined in the class **AbstractSocket**, and is not overridden.

Killing an instance of **UpcnProxy** also asks to kill its instance of **UpcnSocket**, to do so, the kill method is called. Then the same call is done to the netconnect's lifecycle manager. On top of that, the different sockets have to be closed, and the receiver interrupted.

Once the lifecycle of the instances of µPCN has been managed, the user can communicate with them using the package **upcnUtilities.send** and the classes **Receiver** and **ReceiverObserver**. These tools are described in the following sections.

5.2.3.2. Communicating with the instances

Communicating with the instances of µPCN implies two different actions: to send messages and to receive messages. This can be done thanks to the publisher and receiver sockets. The communication with these sockets uses the ZeroMQ library. It aims to create client-server architecture, including different types of communications and is available in most of the languages. Sending or receiving messages uses a very different procedure, thus both parts will be discussed separately.

5.2.3.2.1. Sending messages: For this purpose, the package **upcnUtilities.send** can be used. It provides seven different messages. The specification of these messages can be found in the appendix A. This part also describes the format of the messages, which will be used thereafter. The structure of this bundle can be found in the figure 5.10.

The interface **SendCommand** only asks for an execute function to be implemented. Two implementations exists: the class **SendSeveral** which only sends some commands, and the abstract class ***UpcnSend***. This last class is the most important class of the package, because it manages the different steps of the expedition of a message.

This class is an implementation of the strategy pattern: the function execute calls in this order the functions beginSend, corpusSend and finishSend. The first and last functions retrieve information from child classes, but do not have to be redefined, whereas the second function is abstract.

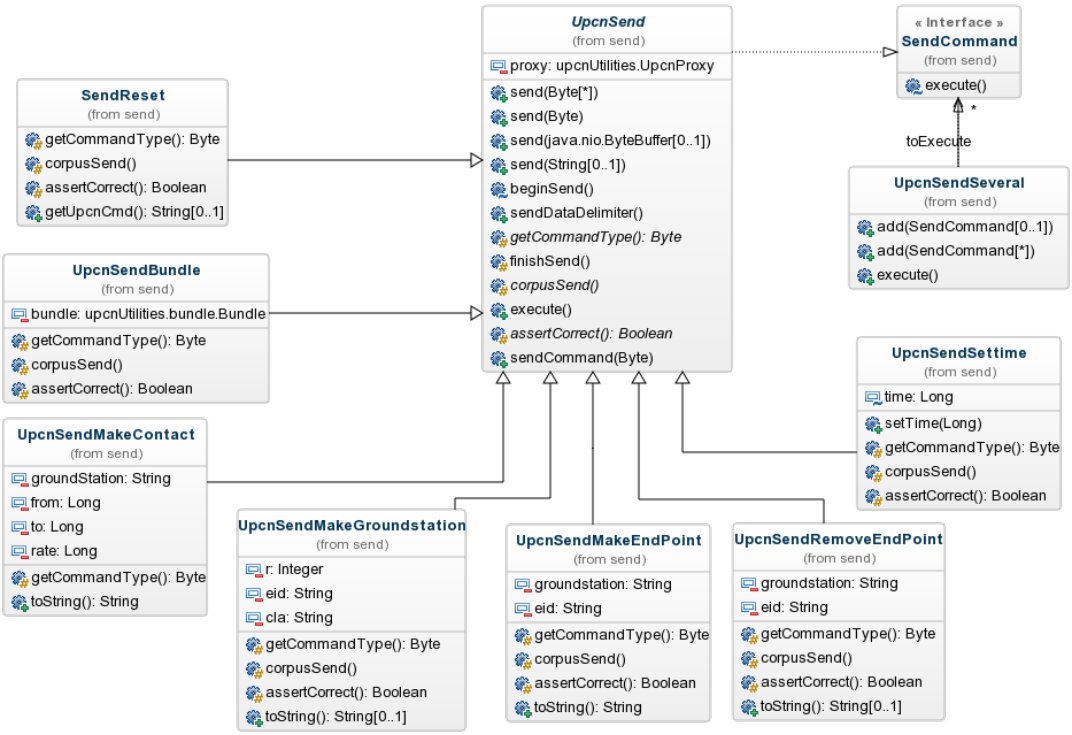

Figure 5.10.: The upcnUtilities.send package

Two types of messages exist: the data-messages which can be used to send data to µPCN, and the command-messages which execute actions on the instance. For instance, sending a bundle is a data-message and creating a groundstation is a command. These two examples will be considered in the following of this part.

The abstract class specifies that the following functions should be overridden:

assertCorrect: checks that the command has been correctly initialized, and raises an exception if it is not the case;

getCommandType: returns the command type of the current command (see appendix A to have more information about the command type);

corpusSend: sends the core of the command

The two following examples will show how these functions are used.

The procedure to send a bundle is represented by the sequence diagram of the figure 5.11. The first action is the initialization of the data-message. Thus the user shall create the object, and set the bundle to be sent. Once this has been done, the method execute can be executed. It has not been overridden then the function of the parent class will be executed. On the beginning of this method is done the call to method assertCorrect.

If no exception is raised, the method beginSend is executed. It will send the header of the command, computed with, inter alia, the result of the method getCommandType. To send the command, the function send of the proxy is done. It will simply send the given bytes to the ZMQ socket.

The call to corpusSend is done next. The provided bundle is serialized, and sent to the proxy with the same manner as before. To conclude, the function finishSend is called, which sends the footer. At the end of this function, the bundle will have been totally sent to the µPCN instance.

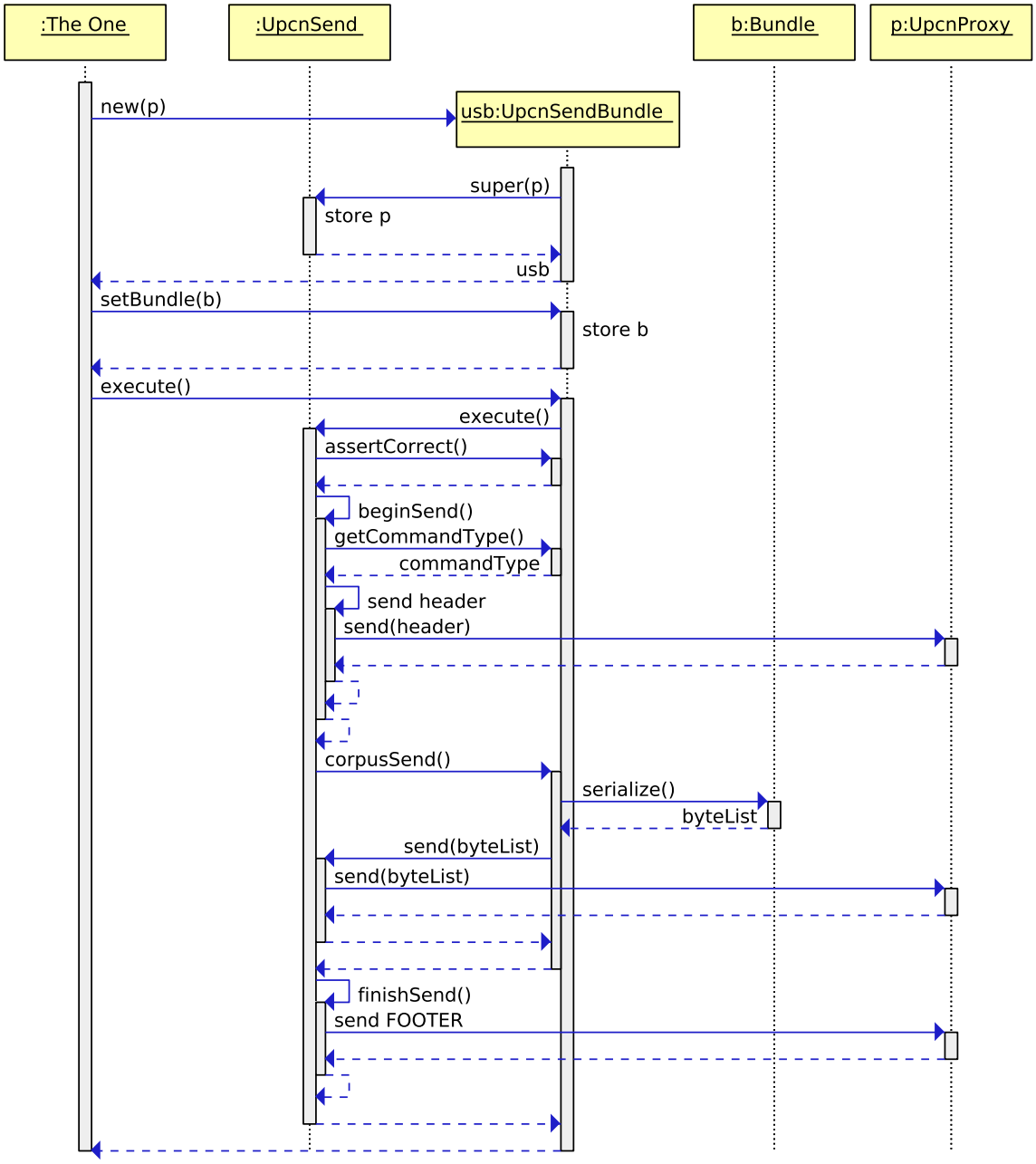

Figure 5.11.: The procedure to send a bundle

The second example is *mkgs*, which creates a groundstation on the µPCN side. This example differs from the first because it is a command, thus it changes a little. However, the structure is the same: the functions accertCorrect, beginSend (and getCommandType), corpusSend and finishSend are called in the exact same order.

The only difference is the content of corpusSend. Indeed, it will only call the function sendCommand of the parent class, with `0x01` as argument, which is the specific byte to create a groundstation. This function will then send the content of the toString method of the child class.

Both of these patterns are used for every of the commands which are provided. The only exception is the command **SendSeveral**, which singly executes every command it has received in the order.

Thus, the class **UpcnProxy** has the ability to communicate with a ZMQ socket, and the different commands of the package **send** can be used to send messages to µPCN. The following part will show how the proxy handles the message reception.

5.2.3.2.2. Receiving messages from µPCN To receive a message, a waiting mechanism has to be settled. Indeed, the software has to wait for a message to be sent to receive it. ZMQ does natively provide a method to wait for the next message, but it cannot be used as such for two reasons: firstly, it only manages one message reception, it is not possible to buffer several messages; secondly, this method is a blocking mechanism: the current thread is blocked until a message occurs.

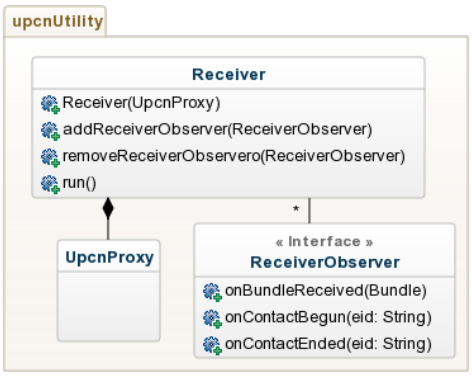

Figure 5.12.: The classes involved in the reception of messages

To solve this issue, the architecture on the figure 5.12 is used. The design pattern *Observer* was used. The class **Receiver** is the observed object, and has the responsibility to notify its observers (implementations of the interface **ReceiverObserver**) when an event occurs. Three events exist: onBundleReceived, onContactBegun and onContactEnd.

Thus, the reception of messages is done thanks to the class **Receiver**. To fix the issue discussed in the beginning of this part, the accurate method to retrieve a message is called an infinite loop, running on a separate thread. Thus, the receiver has to be launched. This is done when creating the instance of **UpcnProxy**.

When a message occurs, the type can be easily retrieved. Thus the messages can be filtered, and only the bundle messages and contact status are proceeded. In the first case, the received message is parsed and the function onEventListener(bundle) is called, with the newly created bundle as argument. In the second case, the accurate listener is simply called following the content of the message.

For one receiver, several observers can be used. This is especially useful when several contacts occur at the same time. To add or remove an observer, the functions addReceiverObserver or removeReceiverObserver can be used.

One implementation of the **ObserverReceiver** exists. It is placed in the core of The ONE, and highly dependent on it, thus it will be discussed in the following part.

5.2.4. Using µPCN in The ONE

Once the package **upcnUtilities** has been developed, it becomes possible to communicate with µPCN during the execution of The ONE. Two steps are involved: the management of the events, and the management of the messages. Three important classes were created: **BundleMessage**, **SingleHostObserver** and **UpcnRouter**.

As said above, a node of the network is represented by an instance of the class **DTNHost**, which contains a router to manage the messages. These messages have to be an implementation of the class **Message**, which is not compatible with the bundle implementation. Thus, the class **BundleMessage**, which extends **Message**, was created. It contains a bundle, and can be forwarded to µPCN.

The class **SingleHostObserver** is an implementation of **ReceiverObserver**. It can handle one contact at the time. Thus, it can react to the reception of bundles and to the changing status of a contact. It also provides a synchronization mechanism: the user can call the function waitContactStart to block its thread until a contact begins, and the function waitContactEnd to wait until the contact ends.

The class **UpcnRouter** is the most important for this purpose. It is able to communicate with µPCN and thus is able to initialize it. It also manages the messages: stores them and transmits them to the accurate node if needed.

5.2.4.1. The events management:

When the core of The ONE creates a messages, begins or ends a contact, some specific events are raised. Three new events were created, in order to be able to manage µPCN.

BundleMessageCreateEvent: This event is used when a bundle message is created. When the information it receive, it can create a BundleMessage and adds it to the accurate DTNHost, which will forward it to its router.

BundleStartContactEvent: This event is launched when a contact begins between a satellite and a basestation. It adds a SingleHostObserver to the proxy, in order to follow the event. Its task is also to set the time of the µPCN instance. Indeed, the clock of the simulator is faster than µPCN's. Thus, before each contact, the clock of the external software has to be synchronized. To do so, the command **upcnUtility.send.Settime** is used.

Because µPCN uses a real clock, and contacts in The ONE can measure several minutes, the time also has to be set to a few seconds before the end of the contact. This is done in two steps. First the event waits for the contact in µPCN has started (using the SingleHostObserver function). Then the time is set to two seconds before the end of the contact. Doing so, it allows time for the software to send bundles and other information, and it ensure that it does not miss the beginning or the end of the contact.

BundleEndContactEvent: This event is raised when a contact, in The ONE, ends. Its tasks are to wait for the contact to end in µPCN, and then to remove the SingleHostObserver, which is not needed anymore.

5.2.4.2. The messages management:

This paragraph will be focused on the classes **SingleHostObserver** and **UpcnRouter**. The first manages the reception of bundles from µPCN and the second is dedicated to the forwarding of messages from and to µPCN.

As said above, **SingleHostObserver** is an implementation of **ReceiverObserver**. Thus, it provides, inter alia, the method onBundleReceived which is called when a bundle is received from µPCN. In this case, it simply forwards the bundle to the accurate instance of **UpcnRouter**.

The class **UpcnRouter** is more complex. It has to manage messages, but also to initialize UpcnProxy. During the initialization of the class, it does indeed create a new proxy, which involves the creation of an instance of µPCN. Then it sets the time to 1, in order for the external instance to not be ahead of The ONE.

The handler which is called by µPCN is prepareTransferToOtherNode. It mainly stores the messages to the next class, using some utilities of the parent class. The message will be transmitted to the accurate node once a contact has been established (which, in most of the cases, is already done).

The second handler, addToMessages, is used when a message arrives from another node. In this case, the message is immediately forwarded to µPCN. To do so, the corresponding contact has to be created in µPCN, indeed the software would not accept the bundle if it does not know what to do with this bundle. To solve this, the software has to know the groundstation owner of the contact. It has also to know that the recipient of the bundle is reachable from the next groundstation. Thus, the following commands are executed: `mkgs` with the next recipient, `mkgs` with the final recipient, `mkct` with the next contact, and `mkep` with the final recipient. Finally, the bundle can be sent.

This last recipient also launches the synchronization, using the event **BundleStartContactEvent**.

5.3. The implementation of the lifecycle manager

The lifecycle manager corresponds directly to the component of the same name, described in sections 4.1.1.3 and 4.2. Thus it shall fit with the use case diagram presented in figure 4.7. The two main use cases are managing the lifecycle of the managed component – µPCN– and providing statistical information. These two goals are also the requirements /R1/ and /R2/. The requirement /R5/ asks the software to be as configurable as possible, and /R6/ asks an independence from the other components.

This part will present the development of the lifecycle manager. Firstly, the different tools will be presented. Secondly, the architecture of this component will be described, to give an overall view of the project. Thirdly the implementation will be discussed.

5.3.1. Language, tools and module used

The development of the lifecycle manager was done using Python. The language and its interpreter are not developed to be efficient, but to be an intuitive language. Its lack of performances is not an issue for developing the lifecycle manager. Indeed, the different tasks that the server shall do are not complex. Even if it were complex, the usage of the lifecycle manager is to test a software in a ring road approach. It would never be used with more than 500 instances, thus a lack of efficiency is not an issue.

Python was developed with other concerns. One of the main advantage is its simplicity. The syntax of Python is very simple, and that makes it very easy to understand. A developer who does not know Python would still be able to understand what had been done, and may be able to correct a bug without having to learn the language. The simplicity of the syntax also produces light applications, and speeds up the time needed for the development.

Another big advantage is the numerous number of available libraries. PIP, a well-known package management system for python tools, allows to easily install some new libraries, which can thereafter be used in the code. There are almost a hundred thousand available packages. Four of them were used during the development of the lifecycle manager.

First of all, because the lifecycle manager is a web server, the *Flask* module was used. It is a micro-framework, used to implement the web services. This framework is very light and implements a RESTFul* request dispatching. It can be embedded in the source code with an annotation system, allowing the module to be almost invisible.

The configuration files are stored in a YAML format (see figure B.1 for an example). This file format has the advantage to be minimalist. It avoids any unneeded characters, making it quick to create. To manage theses files, the module *pyyaml* is used.

The lifecycle manager will have to launch some system commands. For this purpose, the package *subprocess* is used. It can execute a software in background or in foreground. If the command is running in background, it is also possible to detach the process from the python script, avoiding to kill the process when the server is terminated.

Using the last tool, it is possible to send commands to another machine, using the SSH command. However, this is not always the best solution. Doing so, a new connection would be created each time a process is launched, which requires a lot of time. This would slow the server. Instead, the *paramiko* can be used. It can open a SSH connection, and then send as many commands as needed. But the module does not manage well background processes. For this case, *subprocess* will be used.

Another package is used to have information about the current machine. To create an instance of µPCN, a port must be provided, and it shall be unique. To do so, the storage system described in section 5.3.2.3 is used, but it does not have access to the other ports which are used by the system. Thus the module *psutil*[6] can be used to obtain them. However this would only work on the local machine.

5.3.2. The architecture of the lifecycle manager

The lifecycle manager may be used from any location on a computer, by several users or softwares at the same time, or even from another computer. The most common possibility to do so is a web server.

The MVP architecture, imposed by the Flask module, is used in this software. This design pattern, standing for Model View Presenter, is used to split the project in several folders (or modules, using Python terminology) in a logical arrangement. Three main parts shall be defined: the *Model*, which stores the data and executes the logical operations; the *View* which is the representation of the data, accessible from the user; and the *Presenter*, which retrieves the data from the *Model* and creates the *View* with them. The extra module *Helper* was defined, it aims to provide tools that are to be used in the model.

This section will present these four modules. Before describing the different components of each module, the global file system will be described.

5.3.2.1. Global file system

After retrieving the application, the file structure described in figure 5.13 is present. Some files are used to give information about the software, to install it and to launch it, as `Makefile`, `setup.py` or `README.md`. They have no utility once the server has been launched. Every file named `__init__.py` is aimed at defining the current repository as a module in python. Most of these files are empty. The file `run.py` is the entering point of the server. Launching `make run` will directly call this file.

[6] This module has in reality a lot of other features, but only this one is useful for this application.

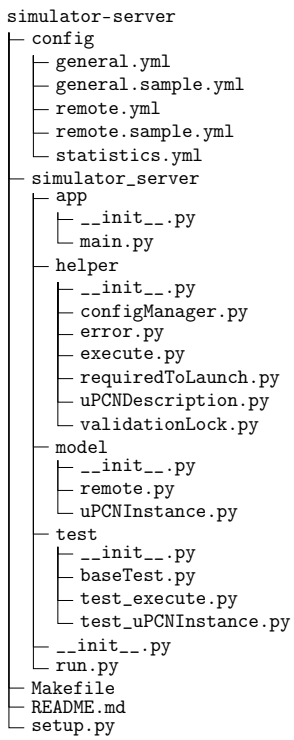

Figure 5.13.: The different files

The folder `config` contains some configuration files, described in section B.1. The folder `simulator_server` contains the core of the application. It contains four different modules: app, `model` and `helper` which will be described in the following of this section, and `test`, which provides testing tools (unit test) for the lifecycle manager.

If the default configuration is kept, the folder `log` will be created at the root of the project. It contains log files, representing the output of the different instances. Every other file or folder is either created with the environment (see the section B.1), either documentation. Every file described in this paragraph can be deleted without issue for the lifecycle manager. However, the folder `log` shall not be removed during the execution.

5.3.2.2. The module App: the definition of the different routes

This module corresponds to the Presenter of the software, in the MVP architecture. It obtains the different requests of the user and will create the view to send it to the user.

Because the view are simple JSON files, it would be useless to define them in a specific module. Thus, they are defined directly in the presenter. The available routes and their effects can be found in the table 5.1.

The module contains the files `__init__.py` and `app.py`. The first file deserves to initialize and in a direct linked way the application. The second file contains the definition of all the routes.

5.3.2.3. The module Model: in charge of the storage

This module corresponds to the model of the software, in the MVP architecture. Every storing system is defined in this module. There is no persistence: data are only kept during the lifetime of the server.

Two types of data are stored:

µPCN instances The lifecycle manager needs to store every launched instance, in order to be

route	action	view
/ /index	fetches every running instances	a JSON array containing every port of the instance
/create	creates a new instance, letting the lifecycle manager automatically choose the route.	a JSON object, whose `port` value is the assigned port, and `ip` it the IP
/create/at-port/<ip>/<int:port>	create a new instance using the specific key. If the port is already in use, an error is returned	a JSON object, whose `port` value is the assigned port, and `ip` it the IP
/create/several/<int:n>	create *n* instances. The ports are automatically chosen.	a JSON array. Every field contains the objects described for a single creation
/<ip>/<int:port>	fetches the statistical information for the specific instance which is running at the specific port	a JSON object, containing the couple name of the statistical command – result of the statistical command
/kill/<ip>/<int:port> /<ip>/<int:port>/kill	kills the instance at the specific key	
/kill/all	kills all the instances of µPCN running on all known machines.	

Table 5.1.: The list of the available routes and their effects

able to interact with them (requirements /R1/ and /R2/). Also, the lifecycle manager shall ensure that two instances are not assigned to the same port on the same machine. Thus, the combination port–machine shall be unique. An instance can then be identify with the combination ip – port

Thus, the data-structure will be a dictionary. The key will be the tuple `(ip,port)`, and the value the description of a µPCN instance: an instance of `UPCNInstance`.

SSH connections In order to create instances on other machines, SSH connections are created. Two facilities show the need of storing the connections.

It could be possible to create a new connection each time a command is sent through SSH, but this would create performance issues. It would be more efficient to create one instance per machine, and use it several times. This involves storing it somewhere.

The requirement /R3/ asks that the best remote machine be chosen automatically. So the lifecycle manager needs to know how many instances are running on each machine.

But the data-structure should store the number of instances on each remote machine and the connection to the remote machine. Other information is also stored (the priority and the maximums of instances per machine). For each SSH connection, a unique key is created. Five lists are created for storage: the key, the priorities, the maximums, the SSH instances and the number of running instances per machine. All those array allow a quicker way to retrieve information.

The module contains the files `__init__.py`, `remote.py` and `uPCNInstance.py`. The first only has a semantic utility. The second defines the SSH connection instances. The last defines the µPCN instances. Both files also provide functions to use them.

5.3.2.4. The module Helper: every other file

This module stands for every class and file which have no place in any of the three modules of the MVP design pattern. They are mainly tools, and deserve to simplify the code inside the Model. For instance, launching a system software from the python side is a frequent feature in the code of the lifecycle manager, and it takes many lines of code. A computer science good practice consists of factorizing this call in a new function. The location of this function may be an issue. Placing it in the Presenter or in the View does not have much sense, and it does not compute data, therefore it does not stand in the Model either. This functionality finally stands in the module Helper, in the new file `execute.py`.

The file `configManager.py` is a tool to ask information to the OS. It may use the functionalities provided by `execute.py`. The file `UPCNDescription.py` computes the statistical information asked in /R2/. It may also use the two previous files. `requireToLaunch.py` is a test to be launched just after the start-up of the server. It tests whether the configuration files (described in B.1) are correctly fulfilled. The two last files (`error.py` and `validationLock.py`) only provide exceptions and locks, used by other files.

5.3.3. The implementation

This part will describe the different use case implementations. It includes two main parts: the management of the instance's lifecycle, and the retrieving of the statistical information. Both use the `execute` and `remote` modules, which can execute commands on local and distant machines.

5.3.3.1. The SSH connection, and executing commands

This part will describe the creation of a SSH connection, and then how to use it. It uses the two modules `helper.execute` and `model.remote`. The second contains also the class with the same name.

5.3.3.1.1. Initialization of the different remote machine As described in the section 5.3.2, the module defines several lists and dictionaries. The first task is to initialize these fields.

The private function `__construct_remotes` operates this initialization. It simply parses the YAML file `config/remote.yml`. It acquires the list of objects representing the different available connections, and the loop on it, ignoring the cases where the given priority is null.

A unique identifier is computed. It is a string, which contains in this order: the IP of the machine, the port of the connection and the user. The different keys are stored in a special list, in order to be able to iterate over the keys. For each instance is stored, in a dictionary with index: the key, the priority, the maximum, the instance of the class Remote corresponding to a key, and the number of µPCN instances which are running on the remote machine.

The only non-trivial initialization is the dictionary *description*. The Remote class instances are only created on its first use (lazy instantiation). Thus, the dictionary stores the key on the instantiation. On the first use, the user shall test if the stored value is an instance of the class `dict`. In this case, it shall replace it with an instance of `Remote`.

5.3.3.1.2. Acquiring a remote instance The static function `get_remote_machine` is provided by the module `remote` (a sequence diagram is present in figure 5.14). It chooses the best remote machine to use next. The function can either return an instance of the class `Remote`, or return `None` if no remote machine is available.

The initialization uses a lazy instantiation. Thus the first action of the function will be to check whether the lists exist. If it is not the case, then it will call `__construct_remote`.

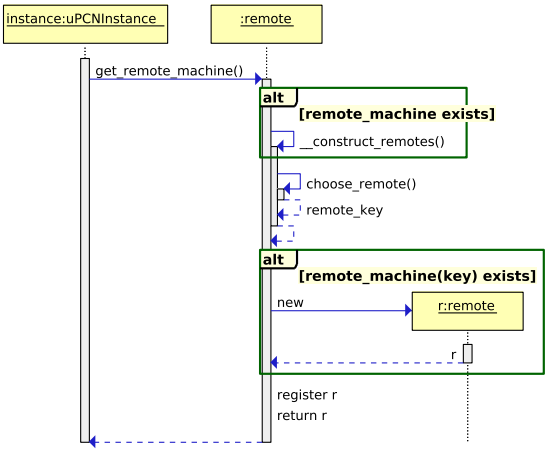

Figure 5.14.: The acquirement of a SSH remote machine

Then it will choose the best remote machine. To do so, it will compute, for each available machine when the priority is not 0, the ratio of the number of instances launched on the machine divided by the priority. The lowest value will determine the instance to use.

Because of the lazy instantiation of the `Remote` objects, the function shall test here whether the acquired object is a key or an instance of `Remote`. In the first case, a new object is created, and stored instead.

The user may use `get_remote_machine` and more generally the acquired remote in two different cases.

- Either he/she/it wants to create a new instance of a remote machine. Then the creation shall be registered to the `remote` module, in order to stay compliant with the maximum and priority requirements. The function may be called without parameter, or with `register` at `True`.
- Either he/she/it just want to launch a command. In this case, it shall not be registered. The function `get_remote_machine` shall be called with the parameter `register` at `False`.

5.3.3.1.3. Creating an instance of Remote: The class `Remote` is managed with a flyweight. This design pattern avoids to create a new object if another with the same parameter exists. This is usually used to save memory, but here the goal is to avoid the replication of SSH connections.

The figure 5.15 represents the creation of a `Remote` object[7].

The flyweight begins with computing a unique key with the received arguments. It then checks if the key is already known from the list `instance`. If it is not the case, the constructor `Remote` is called.

This constructor initializes all the fields. About *paramiko* – the SSH connection manager module –, it only initializes the object, and will not start a connection. The real connection will be created on the first use of it, in order to avoid a useless connection.

5.3.3.1.4. Executing a command: Executing a bash command can be done in several manners. It can either use a SSH connection, either be done on the local machine. It can also run in foreground or in background.

Running a command on the local machine can be done with the help of the module *subprocess*. It can run commands in background as in foreground, and provides a lot of other utilities (managing the log files, detaching the command from the python script...).

[7] The real code is a little different: `Flyweigth` is not a class, but an annotation which handles the constructor of `Remote`. Thus the user does not call `Flyweigth.create_instance(...)` but only the constructor of a Remote (`Remote(...)`).

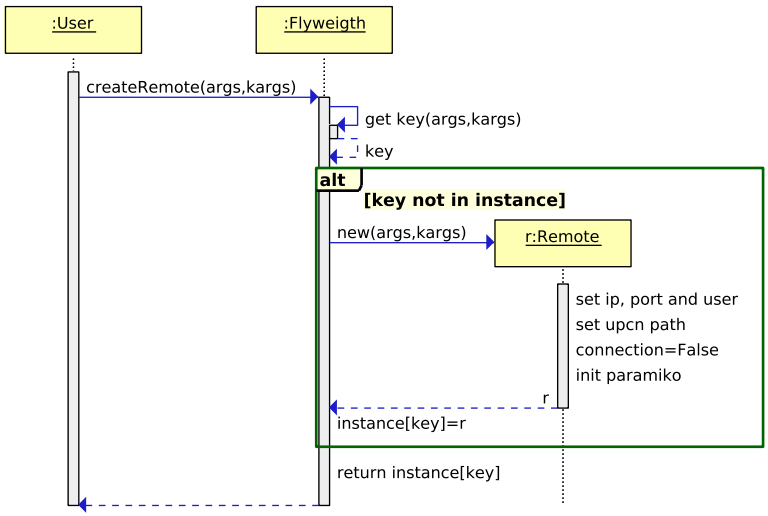

Figure 5.15.: The creation of a Remote instance

The module *paramiko* can easily launch commands on the foreground. However using it to launch commands on the background is less interesting than using the *subprocess* module. It does not natively support SSH connection, but the call to the system SSH software[8] can be used. This creates a new SSH connection, thus it will be slower. Because the use of background is only used when launching an instance of µPCN, it does not happen often. The additional delay will only exist one time per instance creation.

So the module *paramiko* is used to create SSH commands on foreground. Every other cases uses *subprocess*.

When a user launches a command, it automatically calls the module *execute* first, giving the SSH instance as parameter if present. The module will automatically manage the different cases.

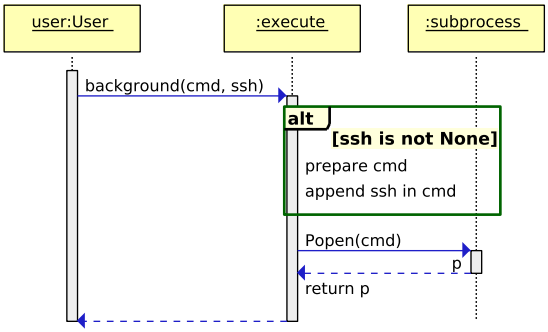

Figure 5.16.: Executing a command in background

The procedure to launch a command in background can be found in figure 5.16. Calling the function `background` begins with appending the SSH command before the original command. The goal is to produce this type of command: `ssh user@ip -p <port> <some bash command>`. Thus the address to use is firstly computed, using the instance of `Remote` SSH to acquire the IP, user and port.

Then the function `Popen` of the module *subprocess* is called. It launches a new instance in background. Some other parameter, hidden on the sequence diagram, can also specify the output files and whether the instance shall be killed on the shutdown of the server.

The call to a command in foreground is presented on figure 5.17.

[8]SSH can be used in this way: `ssh user@ip command`. Then it launches the command directly on the remote

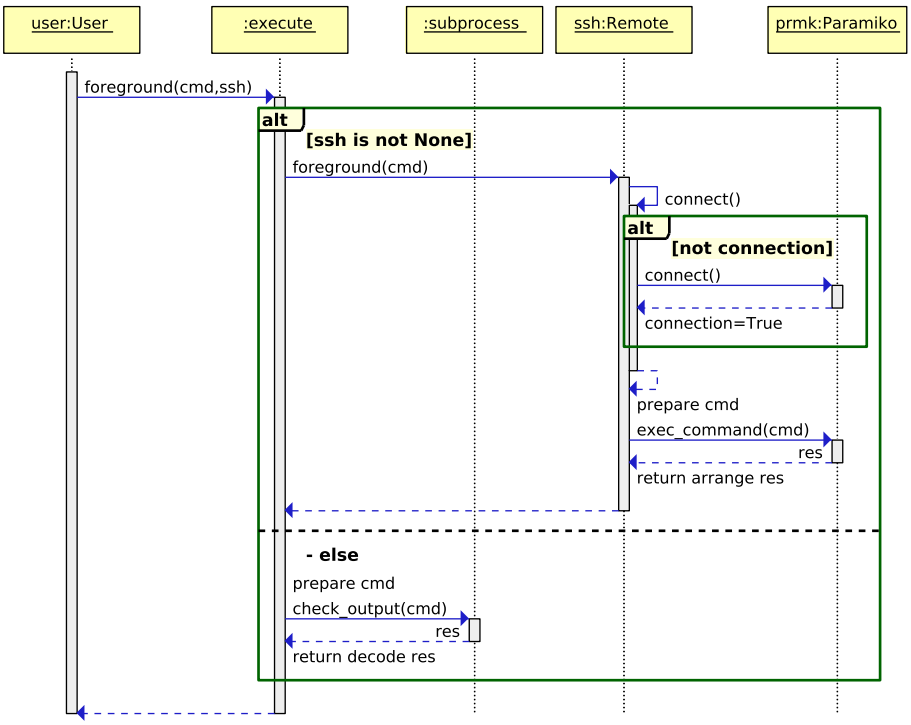

Figure 5.17.: Executing a command in foreground

Two cases are present on the diagram. The first is the presence of a SSH connection. Then the module `execute` will simply forward the command to the `Remote` connection.

As said in the paragraph 5.3.3.1.3, the connection is only established on the first use of the command. The little *alt* fragment describes this lazy instantiation. Then, the command is sent to the paramiko module. The result is parsed into a correct format, and then returned to the user.

The second case happens when no SSH connection exists. The command is first transformed to be compliant with *subprocess*. Then the command is executed via the `check_output` function. Because the result is a byte array, it shall be parsed into a UTF-8 string. Then it is returned to the user.

5.3.3.2. The lifecycle management

The following actions will be described in this part: creating an instance, creating several instances, killing an instance and killing all the instances.

5.3.3.2.1. Creating an instance: The concerned routes are `/create` and `/create/at-port/<ip>/<port>`

The entry points of the queries `/create` and `/create/at-port/<ip>/<port>` are the functions `app.main.do_create` and `app.main.do_create_at_port`. The sequence diagram 5.18 presents the algorithm of the action.

The two possibilities call the function `__create`, which accepts one optional argument, that the second action uses. The function contains two main steps: creating an object of `UPCNInstance` and launching the instance.

The execution of the constructor: Firstly, a new object of the class `UPCNInstance` is created. During this initialization the different attribute of the object are initialized. The constructor accepts two optional arguments: the IP and the port. A specific sequence diagram is present for this method in figure 5.19.

machine.

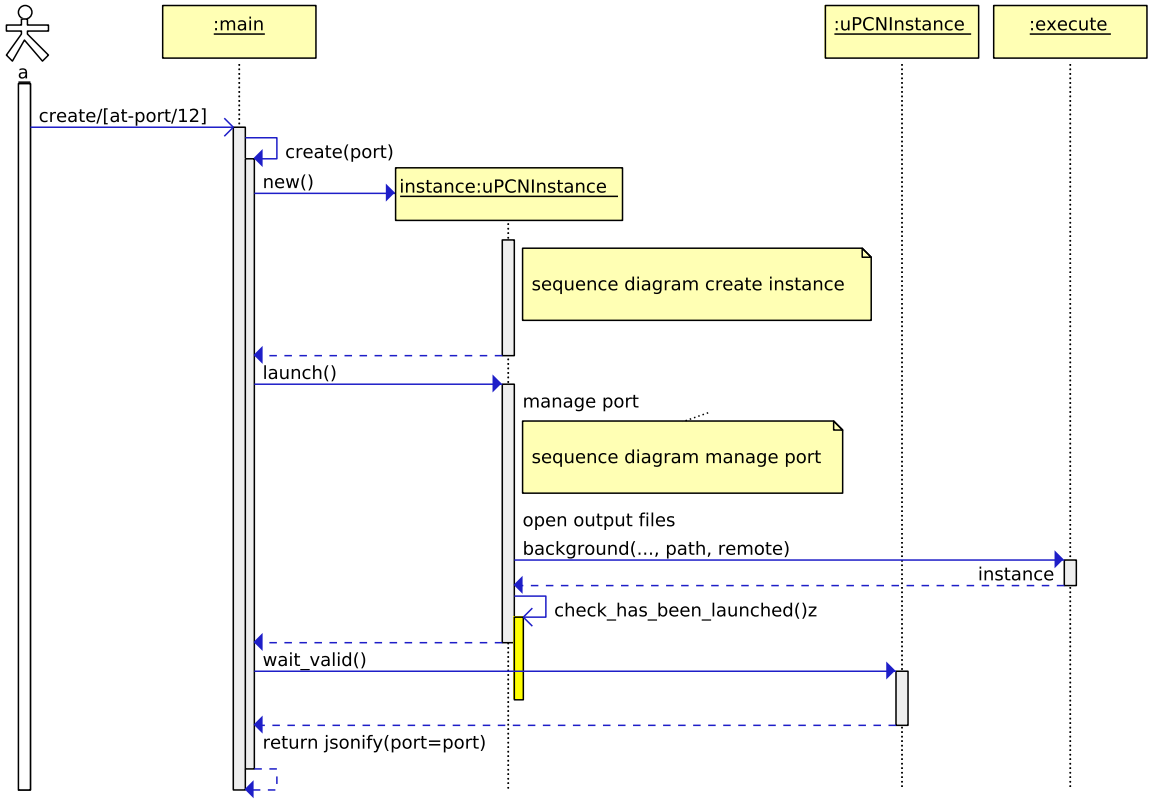

Figure 5.18.: The full sequence diagram of the creation

One of the first actions is to acquire the next remote machine to use. This is done with the function `get_remote_machine` described above. If an IP was given as a parameter, if is also given the function, which will only lookup instances with the same IP. It returns either the instance, either `None`.

The next optional step is to check, when the port is provided, that it is not already used by one of the other instances of µPCN. This can be done by checking that the port is not a key of the dictionary which contains all the instances. If the port already exists, an exception is launched.

If the port was not specified, the real value is not yet computed. It will be explained later that the acquisition of a port requires a system command, which can ask time, especially when using a SSH connection. Thus, a negative value is used in order to specify that it has not been chosen yet.

The last step is to initialize the remaining fields.

Launching the instance Once the instance is initialized, it shall then be launched. The first part is to check whether the port is still available, and also to check if the system does not use it. The sequence diagram 5.20 presents these verifications. The port may also not have been set. In this case the action is done.

Two cases can be observed

- The port has been set during the initialization. Because it may have been changed, the same verification as in the initialization is launched.

  The next step is to acquire a list of the ports in use by the system. A call to the corresponding function in `configManager` can be done. This function uses the function `foreground` of the module `execute`

  Finally the list of ports is retrieved. The lifecycle manager only has to check that the provided port is not in this list. If the port is present on the list, an exception is raised.

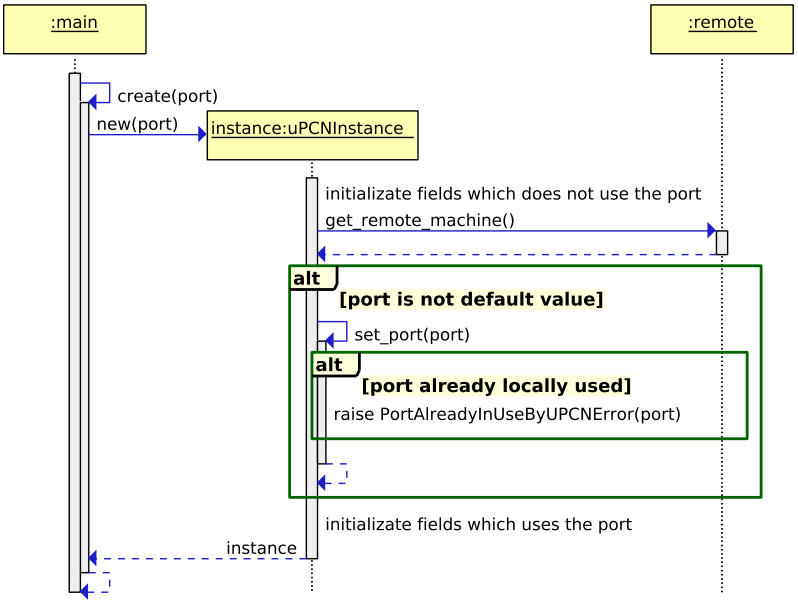

Figure 5.19.: The creation of an instance

- The port was not set during the initialization. In this case, a simple incrementation is done from the default port. The list of the ports in use by the system is retrieved. Then the port is incremented each time that the port is either a key of the dictionary, either in the list. Finally the port is returned.

The next step is to open the log files. Then the module execute is called to invoke the instance of µPCN.

The call to the function `check_has_been_launched`, `wait_valid` and the use of the lock will be sketched later.

5.3.3.2.2. Creating several instances The creation of several instances, whose entry point is `/create/several/<number>` is simple in appearance. It is only a loop, creating *number* instances without specifying the port. However, some modifications should be done in order to fix two issues.

- The performances. Even if the speed is not a requirement, the first tests resulted with two seconds to create one instance with local SSH instances.

  The improvements were to reduce the number of SSH communications, which are slow. Every lazy instantiation was created for this facility

- A bug on the creation of the instance. The utilization of several instances of µPCN on the same machine creates a bug which kills about 10% of the instances. It was not possible to reproduce this bug out of the server. Thus, this solution was computed: after launching an instance, a pause is done in order to let the time to the SSH communications and the creation of the instance to be finished. Then a poll is done in order to know if the instance is running. If it is not the case, the instance is relaunched, by erasing the current instance and calling recursively the method `launch`. This procedure is the core of the function `check_has_been_launched`

  This fix works, but is slow. So it is launched on a new thread. Thus, the main thread can continue to launch the other instances. A locking mechanism was needed. Indeed, they were concurrent accesses on the port acquisition. The lock rendered the port acquisition atomic. Another lock is then used to store the number or instances which are in the validation phase.

Figure 5.20.: Verification of the integrity of the port

It allows to be able to execute the function `wait_valid` at the end of the creation, to know if the creation of instances is terminated.

5.3.3.2.3. Killing a specific instance Two routes provide this facility: `/kill/<port>` and `/<port>/kill`. The handler executes the actions present in the figure 5.21. It begins with retrieving the instance corresponding to the given port. If the instance does not exist, the handler stops here.

Otherwise, the kill method is called. The *subprocess* instance can be killed with the method `terminate`. However, if SSH is in used, it will only kill the SSH instance. In this case, a system terminate command is sent via SSH.

It may happen that the normal terminate does not work. It can be determined by using the `wait` method of the *subprocess* instance, which blocks the thread until the process dies. If a specific timeout is specified, an exception occurs if the timeout is expired. In this case, the `SIGKILL` signal is sent, which occurs when killing brutally the process. The issue is that the µPCN instance may not have had the time to close correctly the port. In this case, it will not be possible to use it anymore.

At this point of the execution, the process has been killed. The object can now be cleaned. Two actions are meaningful: removing the instance of the `UPCNInstance`, which allows to use the port again (if it is available in the system of course), and notify the `remote` module that one new

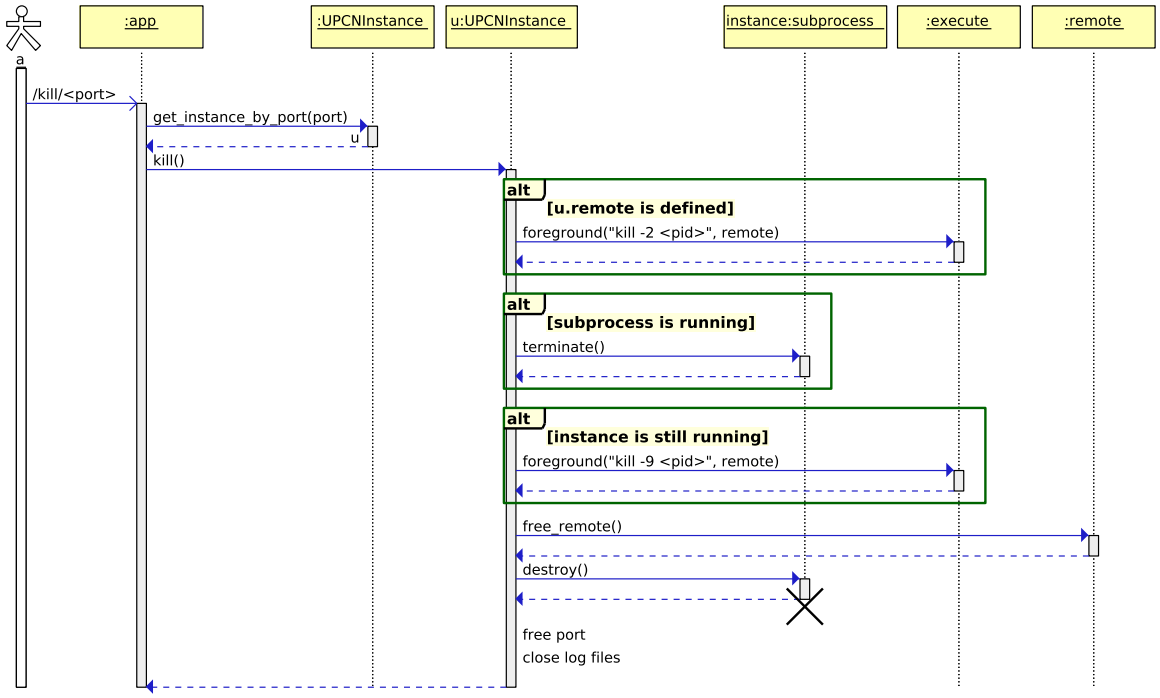

Figure 5.21.: Killing an instance

instance is available.

5.3.3.2.4. Killing all the instances This action is raised after a query on the route `/kill/all`. Instead of looping on all the instances and calling the `kill` method, calling the system method `killall` is faster, because it has to be done only once per machine. Then, the kill method is called with the parameter `killProcess` at `False`. It avoids the need to send the terminate commands.

5.3.3.3. The statistics information management

The computation of the statistics information is done to be extendable, referring to the requirement /R5/. Thus, the list of the requirement can be change dynamically. The idea is the following: the user can provide bash commands, specifying where the lifecycle manager shall introduce arguments.

The different information are stored in the file `config/statistics.yml`. Every item contains (not exclusively) the bash command, a list of arguments, and the cast. The bash command uses the `string.Formatter` to include arguments. The arguments have to be fields of the class `Remote`.

Acquiring the statistic information starts with the route `/<ip>/<port>`. The sequence diagram is present in figure 5.22.

Calling the function `get_description` computes a python dictionary, with the name to be printed as key, and the result as value. It always provides information about the status of the port (is it initialized, is it launched, ...). If the instance is running, it also provides the PID of the instance and the statistic information, computed with the help of the function `__append_stats`.

The class `UPCNInstance` provides the argument PID. This field is not set during the creation of the instance. It is one of the properties of the *subprocess* result, but the value is relevant only if no SSH connection is used. In the other case, the value is the PID of the SSH instance. In order to have the real PID, a system command shall be sent to the remote machine after the end of the creation of the µPCN process. Doing this just after the call to the process, which is not instant, requires to pause, which would slow the launching of an instance. The solution is to, once again, use a lazy instantiation.

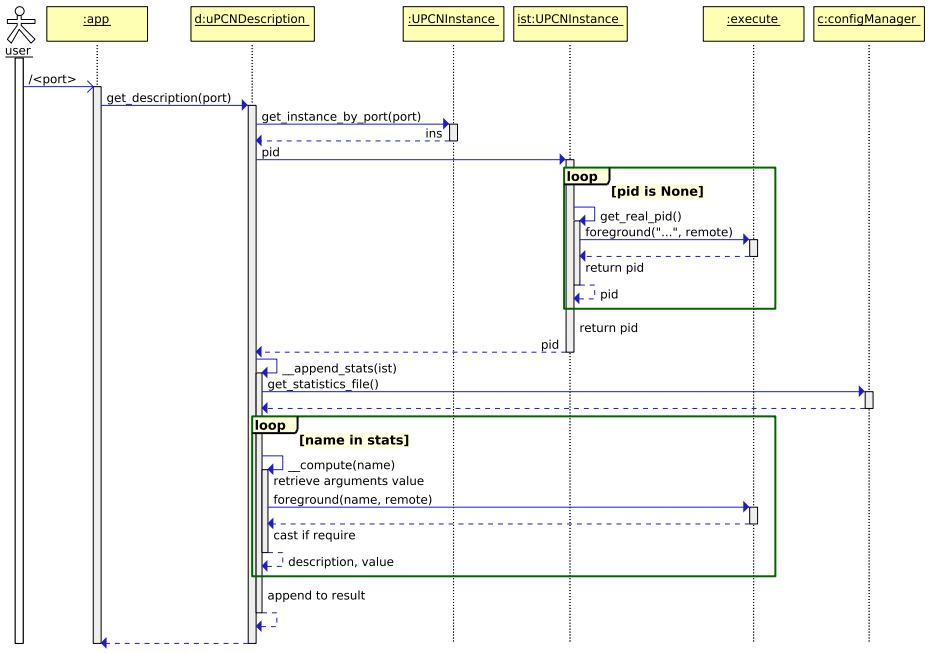

Figure 5.22.: The computation of the statistical information

Then, in the getter of `PID`, the command to read the PID from the machine is launched in a loop until the PID is not `None`. Then it is returned.

The following function to be called is `__append_stats`. It loops over all the statistic commands. On each iteration, the `_compute` calculates the result as follows: firstly, the arguments are retrieved by calling the corresponding field of `UPCNInstance`. Secondly, they are included in the command string with the help of the `string.format` function. Thirdly, the result is sent to the machine with the `foreground` function of the *execute* module. Lastly, the result of the function is cast into the required object. The couple formed with the description of the command and the result is returned.

The first function can appends this result on the result dictionary. Once the loop is over, the result is forwarded back to the app module.

6. Evaluation

In this chapter, the evaluation of the requirements defined in the chapter 3 will be done. Thus, a battery of tests will be launched. The goal is to evaluate whether the lifecycle manager is compatible with a simulation of a ring road environment or not.

The evaluation is split in two main parts: functional tests and performance tests. The different tests are described in the figure 6.1. It summarizes the different tests.

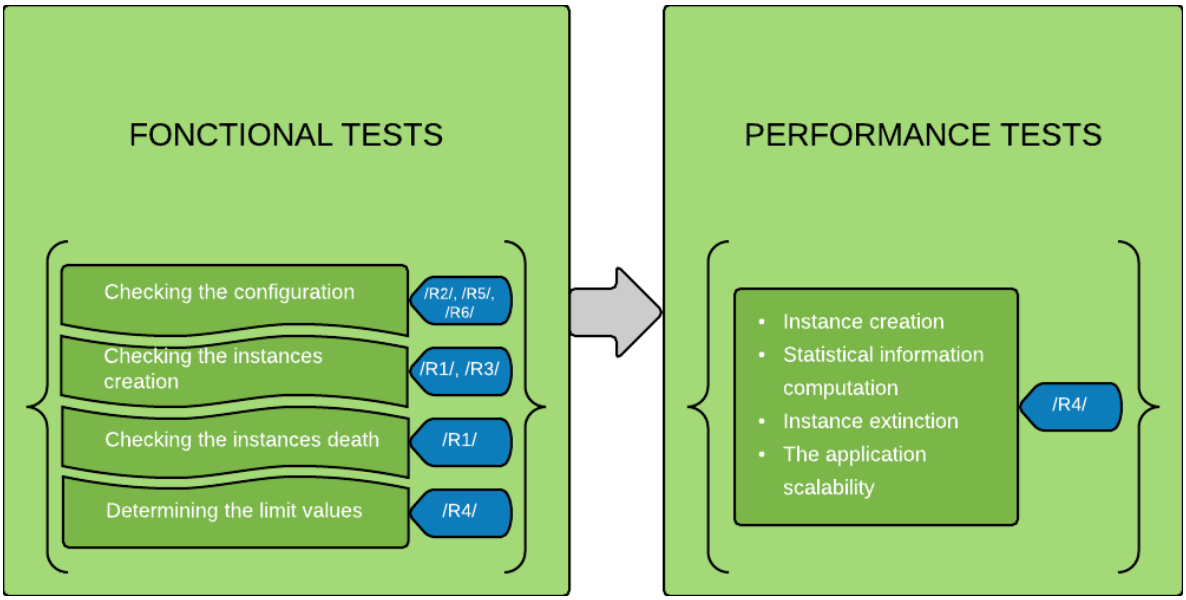

Figure 6.1.: The different evaluation steps

6.1. The functional tests

This section aims to check that all the features are correctly implemented. This can be the execution of the different actions (which can be found in the table 5.1), but also being able to correctly configure the application.

6.1.1. Checking the configuration

This first test aims at determining if the configuration file is correctly interpreted.

When launching the software, it runs tests over the configuration before launching the server to check whether or not it has been correctly fulfilled by the user. Part of the *general* configuration file and the whole *remote* configuration file are tested.

To test these features, launching the software will be enough to check if the configuration was correctly interpreted. If it is not the case, some specific actions are needed.

6.1.1.1. The general configuration file:

This part intends to test the interpretation of the `config/general.yml` configuration file.

Experiences The configuration files are changed to test each feature. For these tests, the different SSH machines, not involved, have to be deactivated. These different tests are launched:

1. launching the server with a full and correct configuration;
2. launching the server with one mistake in the file location. First the mistake was in the path, then in the executable;
3. launching the server with the parameter `killInstancesOnExit` at false, launching some instances, killing the server, then checking that the instances still exist.

Results: The three tests described above are launched. The results are listed below:

1. no mistake: the server is launched successfully,
2. mistake in the file location: The error is correctly detected. The following log message appears: `Error: the file <wrong_path> on the local machine cannot be found.`
3. Testing `killInstancesOnExit`: After killing the server, the instances were not killed.

Interpretation: Since the three tests passed, the interpretation of `config/general.yml` is correctly done.

6.1.1.2. The remote configuration file:

This part intends to test the interpretation of the SSH configuration (`config/remote.yml`).

Experiences As before, the configuration file is changed between each test. These features are tested:

1. launching the server with a full and correct configuration;
2. launching the server with a mistake in the SSH IP, user or port;
3. launching the server with a mistake in the location, either in the path, either in the executable.

Results: The three tests described above are launched. The results are listed below:

1. no mistake: the server is launched successfully;
2. mistake in the SSH address: the error is correctly detected for each mistake. The following error is displayed: `Error: Unable to connect to the SSH target, please insure SSH server is running and the SSH keys are deployed`;
3. mistake in the µPCN path: the error is correctly detected. The following error is displayed: `Error: the file <wrong_path> on local machine cannot be found.`

```
---

test:
    description: "a full test"
    cmd: 'echo "{}.{}"'
    arguments: ["pid","port"]
    cast: float
...
```

Figure 6.2.: A testing file for statistical extension

6.1.1.3. The statistics configuration file:

The interpretation of the statistics configuration file (`config/stats.py`) can be tested.

Experiences: The content of the figure 6.2 can be used for the file .
Then the server has to be launched, an instance shall be created and the statistical information has to be asked. If the result contains correctly the key `"a full test"` and its value (e.g. a decimal number containing, as entire part the PID, and as the decimal part the port), then the behavior is coherent.

Results: The JSON formatted text displayed in the figure 6.3 is returned by the server. Only relevant information is reported.

```
{
  "a full test": 6385.4202,
  "ip": "127.0.0.1",
  "pid": 6385,
  "port": 4202,
  "running": true
}
```

Figure 6.3.: The (partial) JSON result when testing the statistical information

This file shows that the PID has the value 6385, and the port is 4202.

Interpretation: Since, the sentence corresponding to the key *"a full test"* contains correctly the PID and the port, then the interpretation of the statistical file is correctly done.

6.1.2. Checking the behavior when creating instances:

Testing the creation of several instances allows to insure that no bug is present, and more particularly that the bug sketched in the part 5.3.3.2.2 is correctly patched. It can also test whether the priority and maximum management is correctly implemented.

6.1.2.1. Testing the number of created instances:

This test aims at checking whether the maximum limitation is fully respected or not.

Experience: The test is the following: a known number of instances is created (250 was chosen), with the SSH machine disconnected. Several seconds later, the running processes of µPCN were counted[1].

The same test can be launched with all the distant machines connected. However, the SSH instances are also listed with the process, and they shall not be counted. The instances should rather be counted on each machine. The remote machines were each configured to accept 100 instances. The priority was set to 1 for both of them.

Results: Two tests were run: with or without the SSH machines. For the first, 250 instances were counted. For the second, 100 instances were counted on each remote machines, and 50 were present on the local machine.

Interpretation: Since in every case 250 instances were created and because this corresponds to the asked number, the correct number of instances is created. This can also assert that the bug with the immediately dying instances is correctly patched.

6.1.2.2. The maximums management:

The maximum was presented in the part B.1.2. The number configured in the YAML file `remote.py` represents the maximum number of instances launched via a SSH connection. Once the limit number is reached, the instances are created on the local machine.

This test aims at checking if this feature is correctly implemented.

Experience: The test is the following: configuring the project to have a different maximum number for each available remote connection (for example 2 SSH connections with 5 and 20 as maximums). Then more instance as the sums of the maximums shall be launched (following the previous example, at least 26 instances shall be launched; 30 was chosen).

Results: After having created the instances, 5 instances existed on the first remote machine, 20 on the second and 5 remained on the local machine.

Interpretation: The number counted is the same as the set maximum, and the 5 remaining instances are indeed on the local machine. Thus, the maximum management is correctly implemented.

6.1.2.3. The priority management:

The priority management was also described in the part B.1.2. It has the following behavior: the higher the priority is, the more this machine is used to create an instance. In other words: if the priority of two machines is set to 1 and 2, after asking for 3 creations of instance, 1 shall exist on the first machine and 2 on the last. The lifecycle manager shall also begin with the second machine, then use the first and finally use the last.

Experience: To test this feature, the following configuration was set: the priority of two machines was set to 1 for the machine **A**, and 2 for the machine **B**. The maximum has to be big enough to avoid instances being created on the local machine, 50 was chosen for both connections.

Then, 9 instances were created one by one, in order to have the time to check which machine was used.

[1] To do so, the command `ps -aux | grep upcn | grep -v grep | grep -v ssh | wc -l`, when *upcn* matches the executable name, can be used. It will return the number of processes containing the name upcn

Results: The result are displayed in the table 6.1.

Instance number	1	2	3	4	5	6	7	8	9
Chosen machine	B	A	B	B	A	B	B	A	B
Ratio of A machines	0	0.5	0.333	0.25	0.4	0.333	0.29	0.4	0.3

Table 6.1.: The expected result with a priority of 1 and 2 for the machines A and B

Interpretation: The results shows that the instances are created with a ratio of 2/3 on the machine **B** and **A** is used 1/3 of the cases. The ratio of **A** machines is also always close to 1/3, which is the desired value.

Thus, the priority management is correctly implemented.

6.1.3. Killing the instances:

This test aims at checking that killing the instances works correctly.

Two different tests are launched: one with the single kill and one with the multiple kill.

For these tests, 100 instances will be created, 40 on a first remote machine, 40 on a second, and the 20 last on the local machine.

6.1.3.1. Killing the instances one per one:

This first test checks the behavior of the single kill command.

Experience: First, 100 instances are created thanks to the route `/create/several/100`. It returns a json file, containing all the instances and ports. It can then be parsed in order to retrieve the port. Then all of them are killed thanks to the `/kill/<port>` route.

Results: On each machine, the number of running instances are counted after the call to the commands. Each time, no instances were found.

Interpretation: Since no instances remain, killing the instances one by one works correctly.

6.1.3.2. Killing the instances with the killall

This first test checks the behavior of the multiple kill command.

Experience: After the creation of all the instances, the `/kill/all` command was sent.

Results: On each machine, the number of running instances are counted after the call to the kill-all command. On each machine, no instance was found.

Interpretation: Since no instances remain, killing all the instances works correctly.

6.1.4. The limit values:

The goal of this test is to determine the limiting factor.

Experience: The idea is the follow: 200 instance creations are ordered. This test is repeated until a problem occurs. This test was only done with the SSH machine deactivated. It is run once with the log files activated, and once deactivated (by removing the corresponding line in the accurate configuration file).

Results: The creation with the log files activated fails after 508 instance creations. The result of the creation request is this one displayed in the figure 6.4, which underlines that too many files were opened for the OS.


```
{
  "OSError": "After 508 creations: [Errno 24] Too many open files",
  "instances": [
    {
      "ip": "127.0.0.1",
      "port": 4600
    },
    {
      "ip": "127.0.0.1",
      "port": 4601
    },
    {
      "etc, until ..."
    },
    {
      "ip": "127.0.0.1",
      "port": 4707
    }
  ]
}
```


Figure 6.4.: JSON result after three times 200 creations

With the log file disabled, the execution fails after 1280 instance creations. The kernel error indicates that the maximum number of allowed thread creation is reached.

Interpretation: In both cases, the errors are system limitations, and are not dued to the lifecycle manager. Furthermore, as the result shows, the error is well handled by the system, and the 507 or 1280 first instances can be used.

Thus, it appears that only the system limits the lifecycle manager, or at least that the OS limits the execution faster than the lifecycle manager does.

In every case, these limitations are not an issue, since a ring road environment would imply less than 500 instances. Thus a simulation with more than this number would not have so much sense.

6.2. Testing the performances:

In this part, the performances of the lifecycle manager will be tested. The goal is to measure how fast is the application. The same test were launched either with nor without activating the remote machines. In this part, the establishment of the benchmark tool will be described, then the different tests and their results will be shown, and lastly the results will be discussed.

6.2.1. The benchmark infrastructure

To measure as precisely as possible the speed of the application, three different metrics were computed: the `perf_counter`, the `process_time` and the number of cycles.

Performance counter: This value is provided by the function `perf_counter` of the module `time`. It is described in the official documentation as *a clock with the highest available resolution to measure a short duration*. The unit of the result is in fractions of seconds, even though it has no links with wall time seconds (i.e. the "real" seconds). This value shows little dependence on the architecture of the machine used.

The cycle number: This value is provided by the C function rdtsc[2]. It retrieves the current value of the CPU counter. The value is little dependent on the architecture of the machine used. It seems to be highly related with the performance counter.

Process time: This value is provided by the function `process_time` of the module `time`, and is the sum of the system (work in kernel) and user (work in the active thread) CPU time. It is also measured in fractional seconds. It does not include the time when the process sleeps, thus it is more accurate than the performance counter, but is more dependent on the machine.

This three values cannot be used as such, but shall be used to compute a delta. They are computed by retrieving the values before and after the entry routes of the application. More specifically, it was done around the whole functions `do_create`, `do_create_at_port`, `do_create`, `do_index`, `do_get_instance`, `do_get_instance` and `do_kill_all`. For the function `do_create_several`, the measurement was done inside the function, before and after each instance creation.

To perform these tests, the server was launched on a machine running on a Intel Core I7-3537U CPU (2.00 GHz) and with 8 RAM Gio. The distant machines were two Raspberry Pi model B. All the machine were interconnected using WiFi (1.2GHz 64-bit quad-core ARMv8 CPU, with 1 RAM Gio).

6.2.2. The different tests

The goal of the different tests is to evaluate the efficiency of the creation of the instances, the retrieval of an instance and the extinction of the instances. Also, the scalability of the instance creations was tested. For all the tests, the log files were deactivated. Each test was launched twice: once with SSH and once without. In the presented data, the first and last 10% were excluded, to avoid any OS optimization distortion of the data.

6.2.2.1. The efficiency of the instance creations

The goal of this part is to test the efficiency of the creation of one instance, using the three different facilities (creating an instance without specifying the port, creating an instance at a specific port, or creating several instances at once).

Experiences: To test this feature, some instances were created in different ways. After each test, the lifecycle manager was restarted. First, 100 instances were created one by one (with the route `/create`), letting the lifecycle manager choose the port. Second, 100 instances were created, but this time the port was chosen by the user (using the route `/create/at-port/<port>`). Then, 100 instances were created all at once, using the route `/create/several/100`.

[2] This function was wrapped by Sebastian Schrader (see `https://github.com/sebschrader/pyrdtsc`)

Results: In the different graphs, the measures represents the creation of one instance.

Firstly, as an overview, the figures 6.5, 6.6 and 6.7 can be used. It represents the distribution of the creation duration, using the performance counter (fig. 6.5), the number of CPU cycle (fig. 6.6) and the process timer (fig. 6.7).

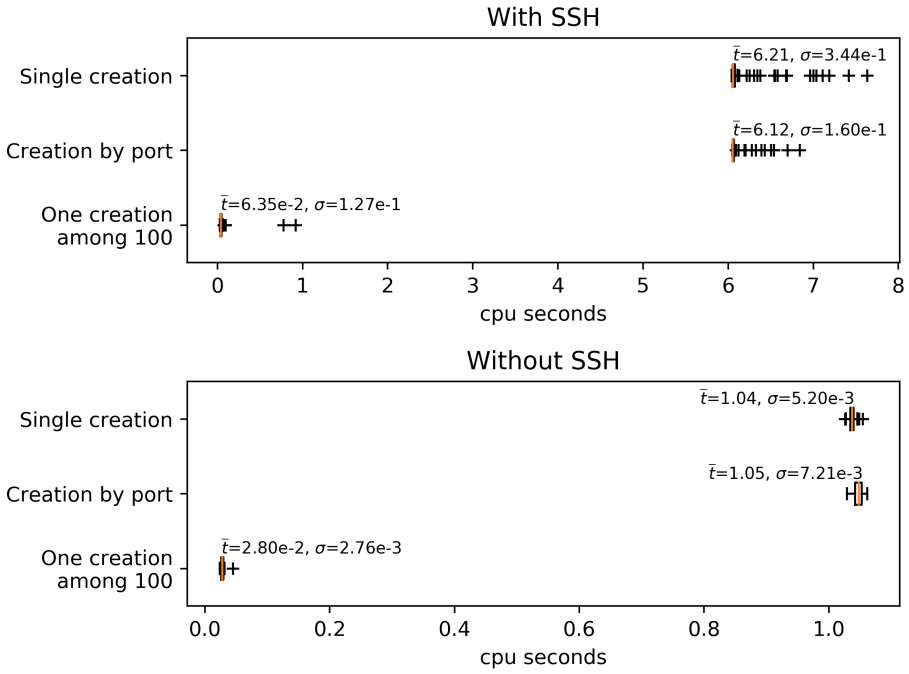

Figure 6.5.: The distribution of the creation duration, using the process counter

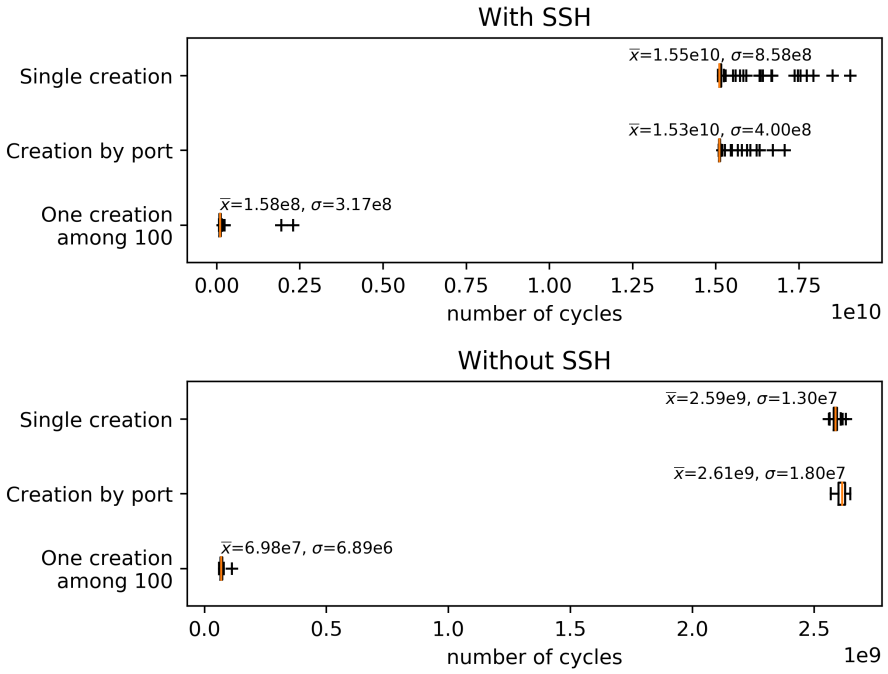

Figure 6.6.: The distribution of the creation duration, using the CPU cycles

The idea of these files is to get some information about the different types of creations. Thus, the single creation is much less efficient than the multiple creation. Regarding the process time,

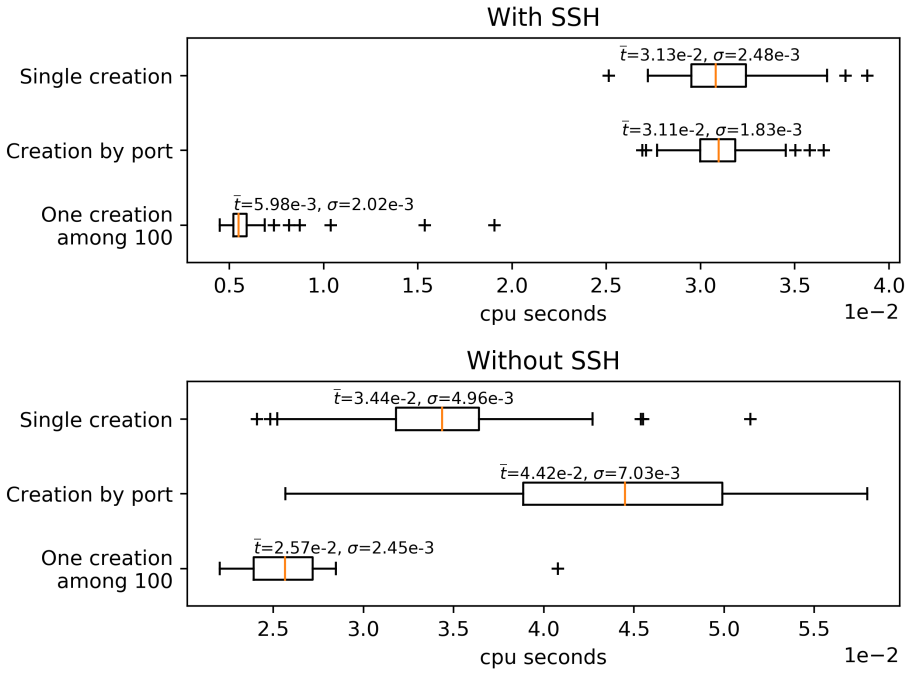

Figure 6.7.: The distribution of the creation duration, using the process timer

using SSH seems to be more efficient than using the local machine. Also, the values seem to be more dispatched if the remote machine is used.

To be more precise, the results were separated in several figures: the figures 6.8, 6.9 and 6.10 are focused on the single creation duration measured with the performance counter, the number of CPU cycles and the process timer, whereas 6.11, 6.12 and 6.13 display the multiple-creation duration using the process counter, number of CPU cycle and the process timer. *Be careful, the scale is not the same between the graphs with SSH and without SSH.*

When creating the instances one by one, specifying a port seems to increase the efficiency when using SSH, and on the contrary decrease when SSH is not used. As said earlier, the value are more dispatched when using a SSH remote machine. The performance counter measures show that using SSH delays from 5 seconds the creation from an instance.

In the other hand, creating several instances at once shows similar results. The dispatching of the values is also bigger when using SSH. it also appears that some rare instances need at least two times more duration to be created. In this case, the performance counter shows that not using SSH is a lot faster.

For every type of creation, the process timer shows that using SSH is more efficient than not using it.

Interpretation: According to the previous values, some facts stand out: creating several instances at once is faster than creating them one by one; some instances need more time than other to be created; the dispatching is bigger when using SSH; and the process time measurement shows a better efficiency when not using SSH.

The fact than creating several instances at once is more efficient than creating them one by one is an expected behavior. Indeed, in both cases, the creation of one instance implies launching a verification in a new thread in order to check whether or not the instance was correctly launched. Then, if one instance has been created, the process waits for the verification to terminate, whereas for the multiple creation, this waiting takes place after all the creations.

Thus, the final delay (one second without SSH, six with) lowers if the number of instances

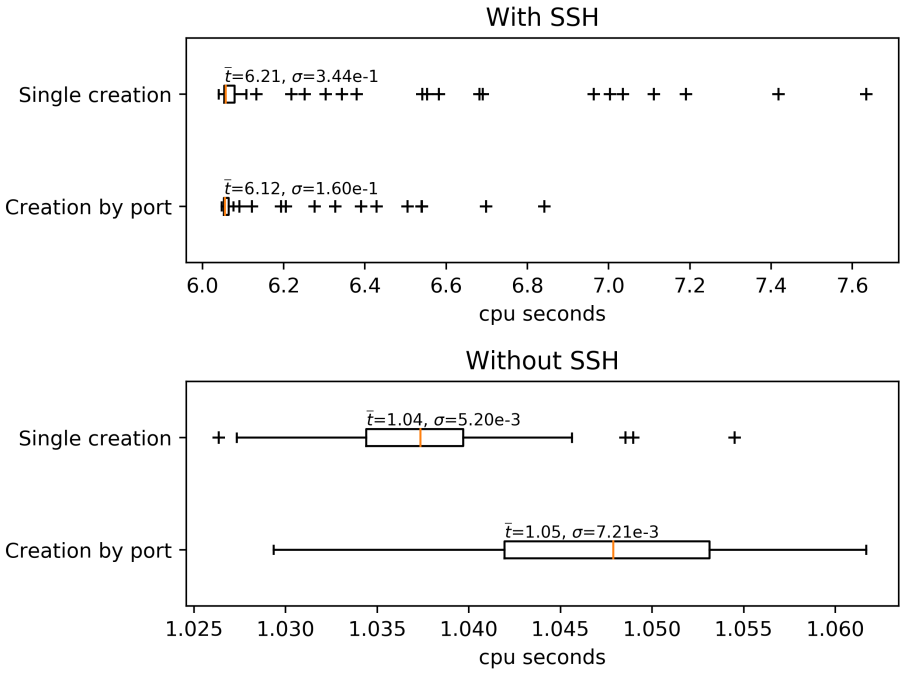

Figure 6.8.: The distribution of the single-creation duration, using the performance counter

to create increases. In the case where the instances are created one by one (figure 6.8), the performance counter shows clearly the delay of 1 and 5 seconds discussed earlier. On the contrary, this delay is not displayed on the figure 6.11).

The instances which need more time to be created are probably those which had to be relaunched. It would explain why two of them, on the figure 6.11, need four times to be created. This additional delay is created by the lifecycle manager, but is necessary, in order to check if an instance was correctly created.

The dispatching of the values, which is higher when using SSH, can be explained by the network activity. Indeed, some unexpected delays can appear when using a SSH connection, because of some latency on the network. There is no possibility to control this delay, thus this is not an issue of the lifecycle manager.

This delay can also be combined to the previous issue: if an instance fails to be created, then after the 6 seconds of verification, it is relaunched, and 6 new seconds of verification are needed. Thus it can reduce a lot the throughput.

With the process timer, the creation appears to be less efficient when creating the instances on the local machine. The main difference is that the lifecycle manager creates either an instance of µPCN (in the case of the local machine utilization) or an instance of SSH. The creation of the SSH instance may consume fewer resources than µPCN.

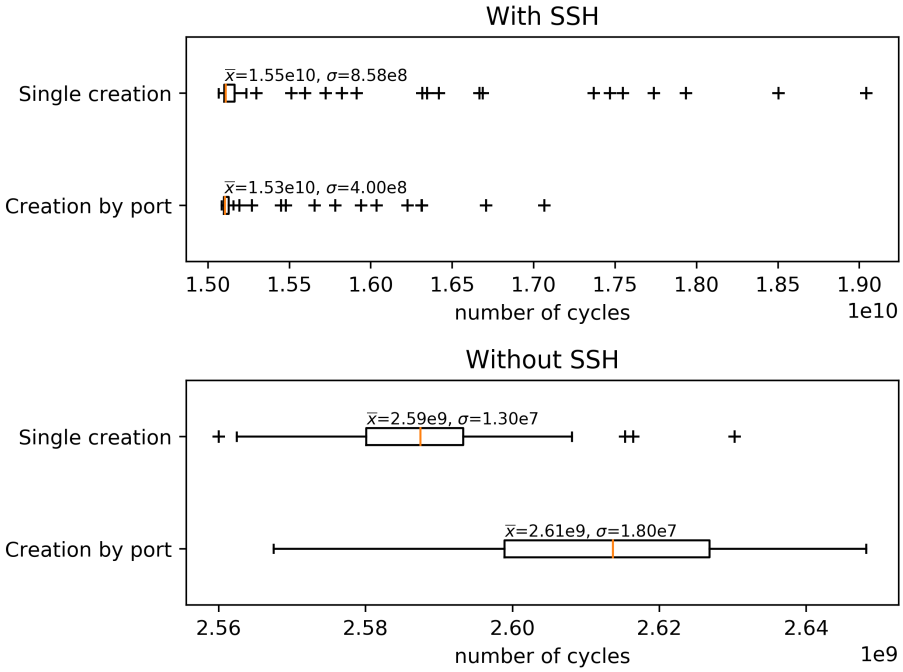

Figure 6.9.: The distribution of the single-creation duration, using the CPU cycles

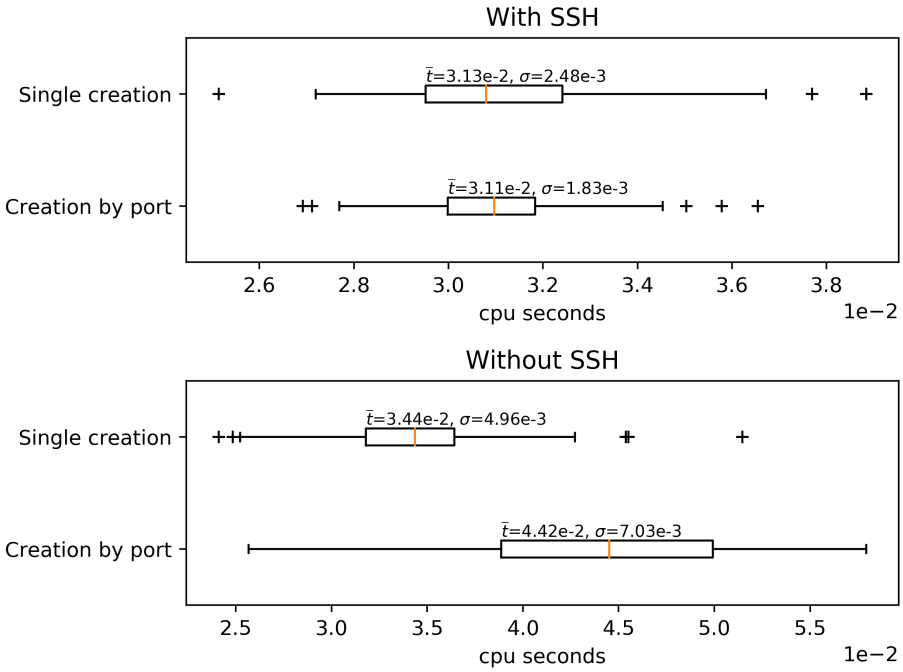

Figure 6.10.: The distribution of the single-creation duration, using the process timer

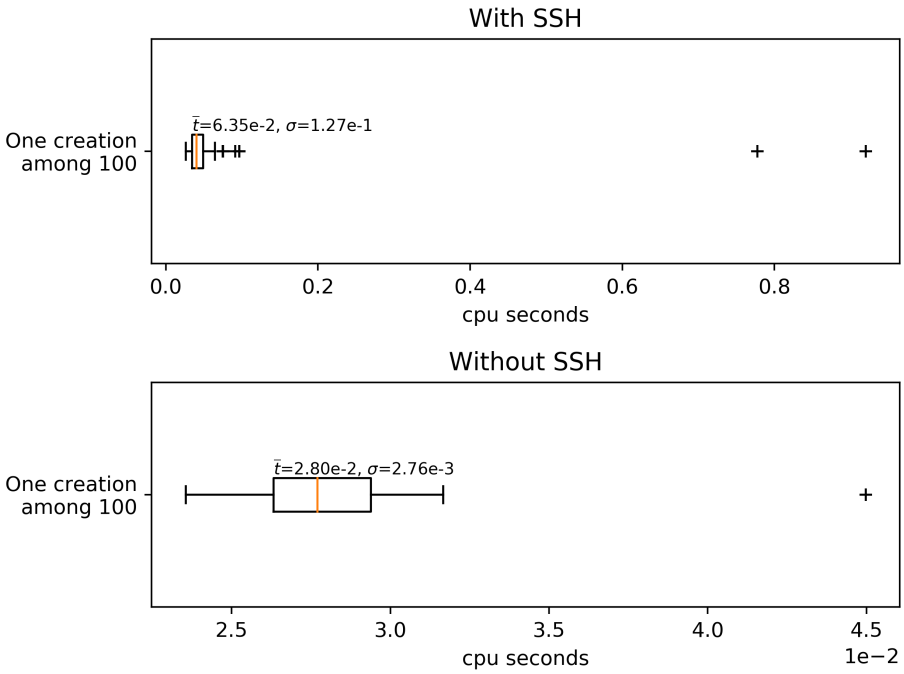

Figure 6.11.: The distribution of the multiple-creation duration, using the performance counter

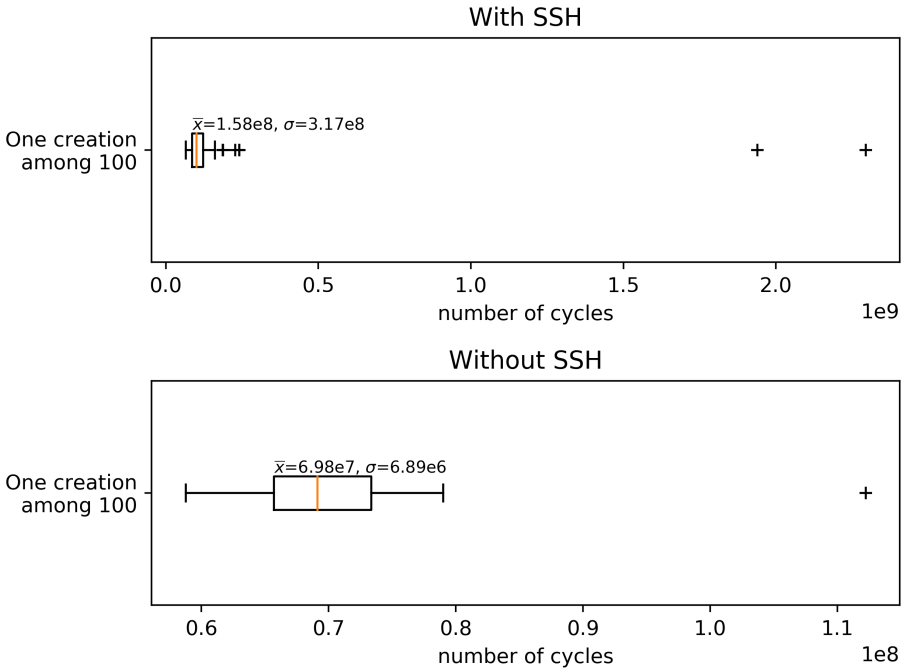

Figure 6.12.: The distribution of the multiple-creation duration, using the CPU cycles

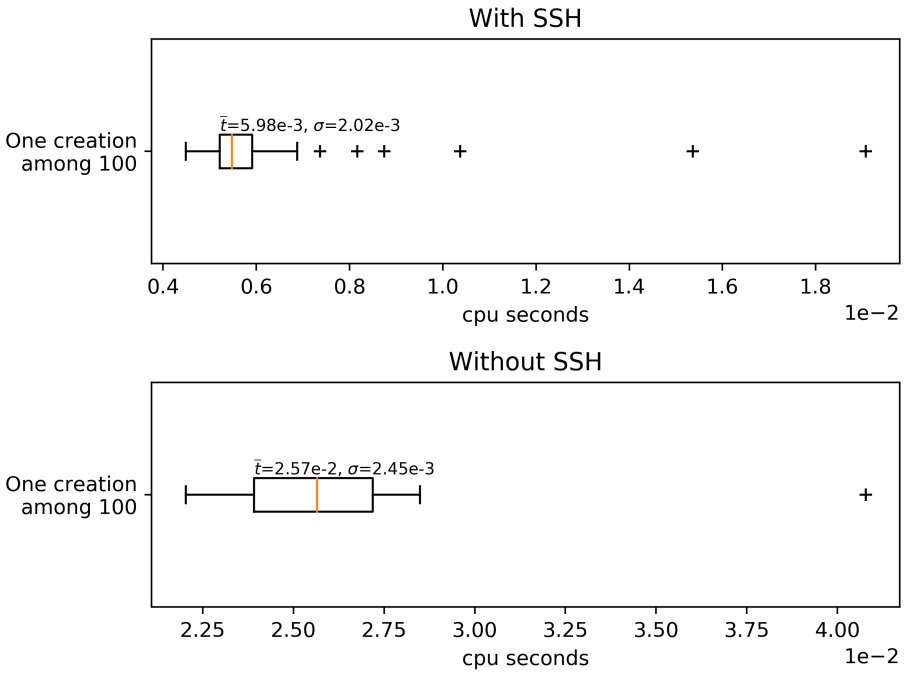

Figure 6.13.: The distribution of the multiple-creation duration, using the process timer

6.2.2.2. The computation of the statistical information

This test aims at the compute the efficiency of the statistical information. Since the different values can be changed by the user, this value can be very different with a different configuration. Thus the single values are not very interesting, but the comparison of this values has to be used.

Experience: To test this feature, 100 instances were created (with the three type of creation). Then, for all the instances, each one was accessed through the route `/<port>`. As said during the chapter 5, the PID is lazy implemented if using SSH, which involves that the first call to this route also computes the PID. Thus, each instance was accessed a second time, and the result for this last round is used for this analyze.

Results: The results can be seen on the figures 6.14 (using the performance counter), 6.15 (using the CPU cycles) and 6.16 (using the process timer). It represents the distribution of the retrieval duration, using the performance counter and the process timer.

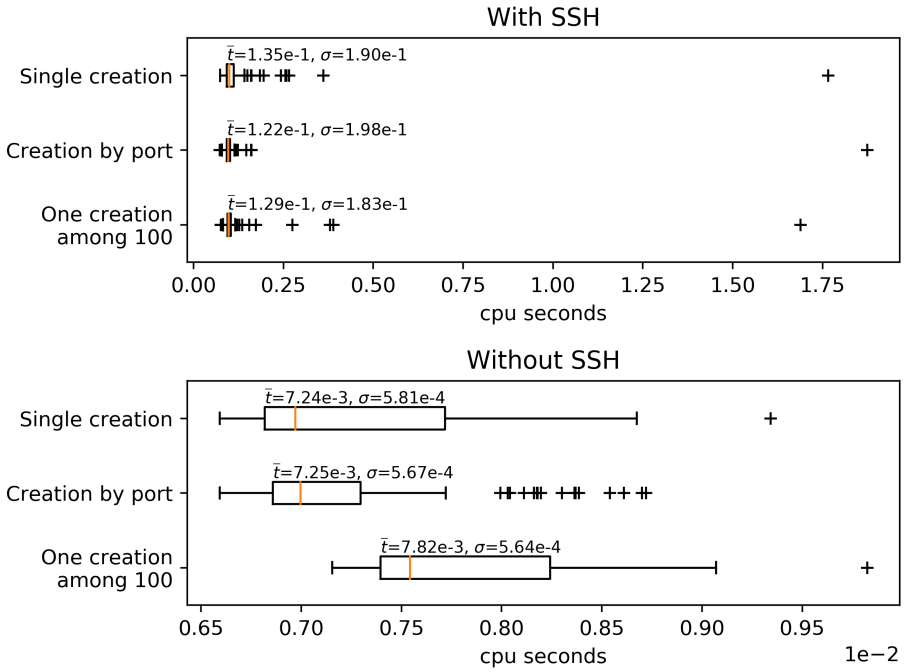

Figure 6.14.: The distribution of the retrieval duration, using the performance counter

On the figures, three different sets of values are present, they correspond to the duration (or usage) for one instance. The results show that the computation of the statistical information is as efficient for every type of creation. Only the retrieval for instances created with the multiple command while using SSH seems to be a little less efficient.

The comparison between the usage of SSH and not using it shows that its utilization decreases the efficiency.

Analysis of this result: The fact that the same order of magnitude is observed for every type of instance is perfectly logic, because the exact same function is observed.

Likewise, it is normal that the computation of the information is slower when using SSH. Indeed, the bash commands are sent via SSH instead of locally, which implies a delay.

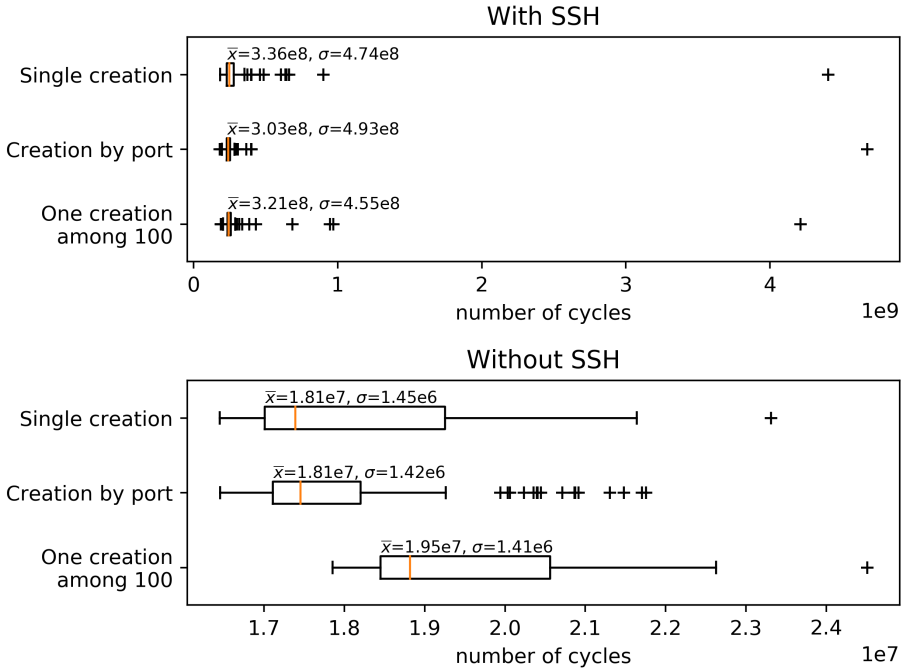

Figure 6.15.: The distribution of the retrieval duration, using the CPU cycles

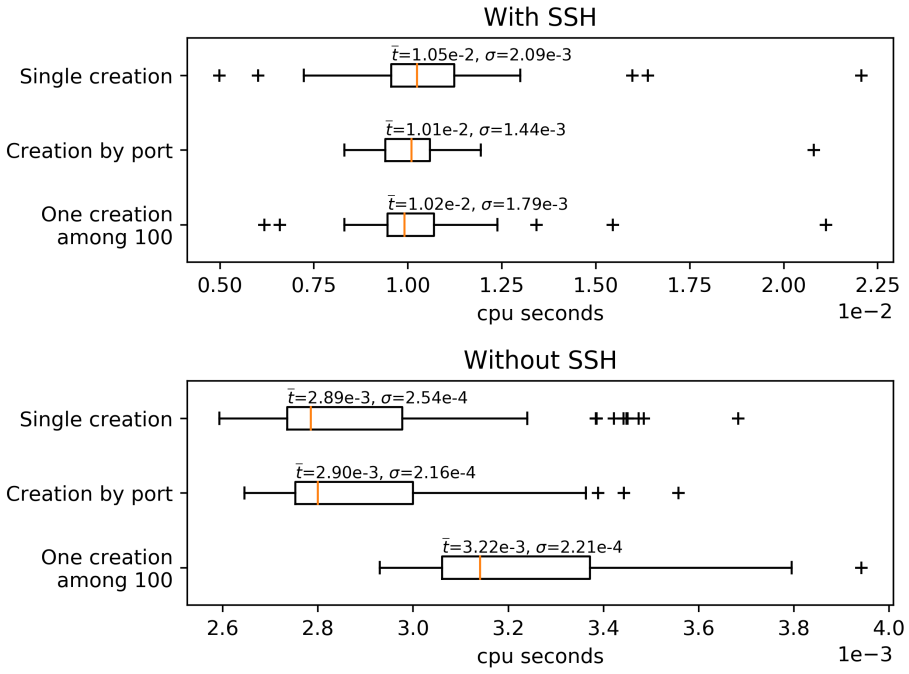

Figure 6.16.: The distribution of the retrieval duration, using the process timer

6.2.2.3. The extinction of the instances

This test aims at measuring the efficiency of the extinction of an instance.

Experience: The test is done as follows: 100 instances were created with the three type of creation. Then all of them were killed using the route `/kill/<port>`.

Results: The result is displayed in the figures 6.17, 6.18,and 6.19. It represents the distribution of the extinction duration, using the performance counter (fig. 6.17), the CPU cycles (fig. 6.18) and the process timer (fig. 6.19).

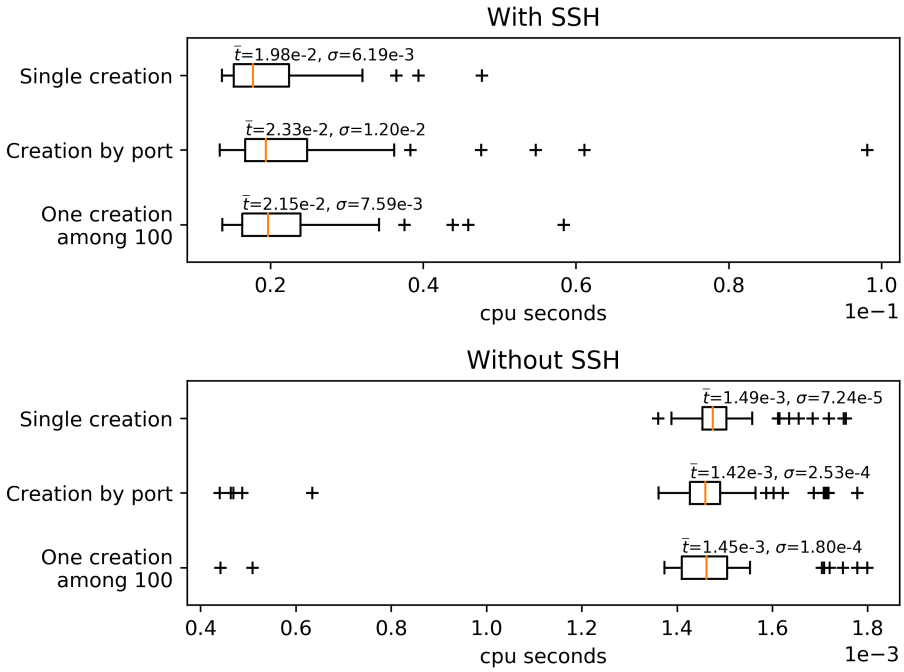

Figure 6.17.: The distribution of the extinction duration, using the performance counter

Once again the three types of instances can be compared. It is very clear that, for the same metric, the duration is very similar. Once again, the utilization of SSH decreases the efficiency.

Analysis of this result: For every type of instance, the extinction uses the same method. So the results shall be alike. The SSH decreases the throughput for the same reasons as before: the `kill` commands are sent either directly to the local machine, either through the SSH connection. Since the second possibility is slower, it decreases the efficiency.

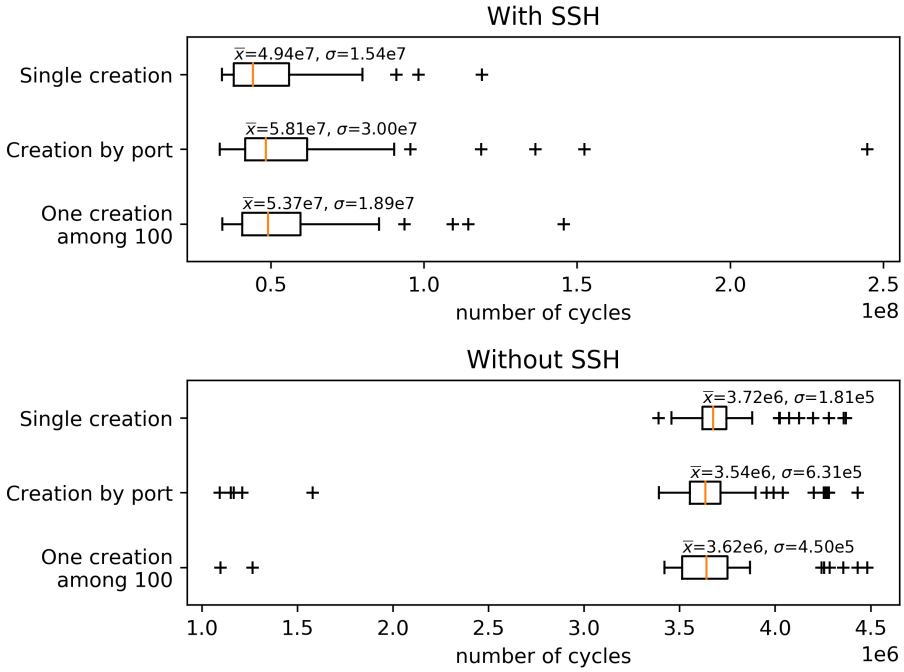

Figure 6.18.: The distribution of the extinction duration, using the CPU cycles

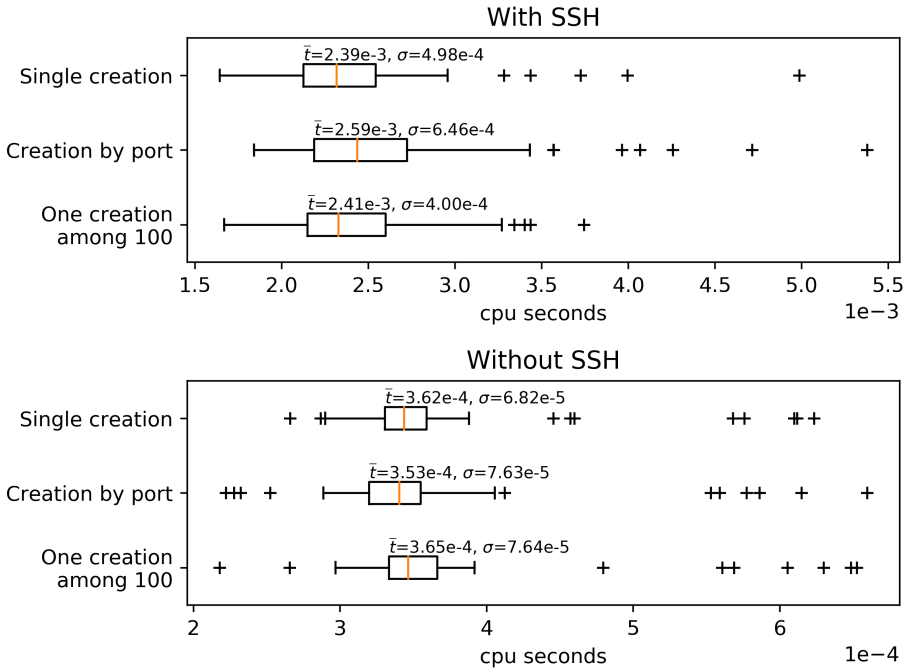

Figure 6.19.: The distribution of the extinction duration, using the process timer

6.2.2.4. The scalability of the application

The goal of this test is to highlight the evolution of the creation's efficiency. As said in the beginning of this part, the duration to create an instance depends a lot on the different configurations (SSH activated, type of creation in use, ...). But the previous tests could not highlight any scalability issue.

Experience: To detect any scalability issue, a battery of tests will be launched, with different SSH configurations. For each of them, 1000 instances will be created at once, and the lifecycle manager will be killed between each test. Two SSH machines will be configured, the machine **A** has 500 as maximum, and 1 as priority whereas the machine **B** has 300 as maximum and 1 as priority. First, any SSH connection is disabled, thus the instances will be created on the local machine. Then, only the machine **A** is enabled. Then, only the machine **B** is enabled. Finally, both machines are enabled.

Result: The figures 6.20, 6.22 and 6.21 display the result of these tests, measured with the performance counter, the process time and the CPU cycles. The graphs represent the evolution of the creation duration based on the number of already created instances. Once again, the 100 first and last results were omitted. On all graphs, the evolution of the duration needed for each instance creation is displayed as two curves. The first is computed when no SSH machine is in use, and is the same for all graphs in order to give a comparison point. The second is computed when using one or more SSH machines.

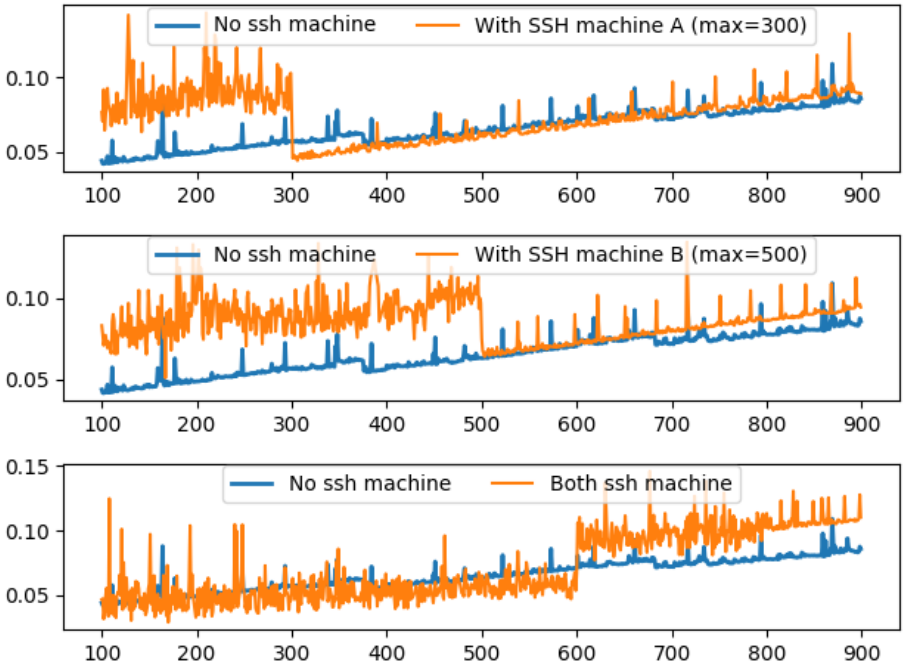

Figure 6.20.: The evolution of the creation duration, using the performance counter

Every graphs presents a linear augmentation, with some important level changes. These steps are the most easily visible on the CPU cycle representation (figure 6.21). On the first graph, the step occurs after 300, on the second it happens at 500. On the last graph, two stages exists: the first is visible at 600, and the second can be detected after 800 creations.

For the two first graphs, the steps correspond to a fall of around 10^8 cycles. On the last graphs, the first stage is a rise of circa 10^8 cycles, whereas the last corresponds to a dispatching reduction.

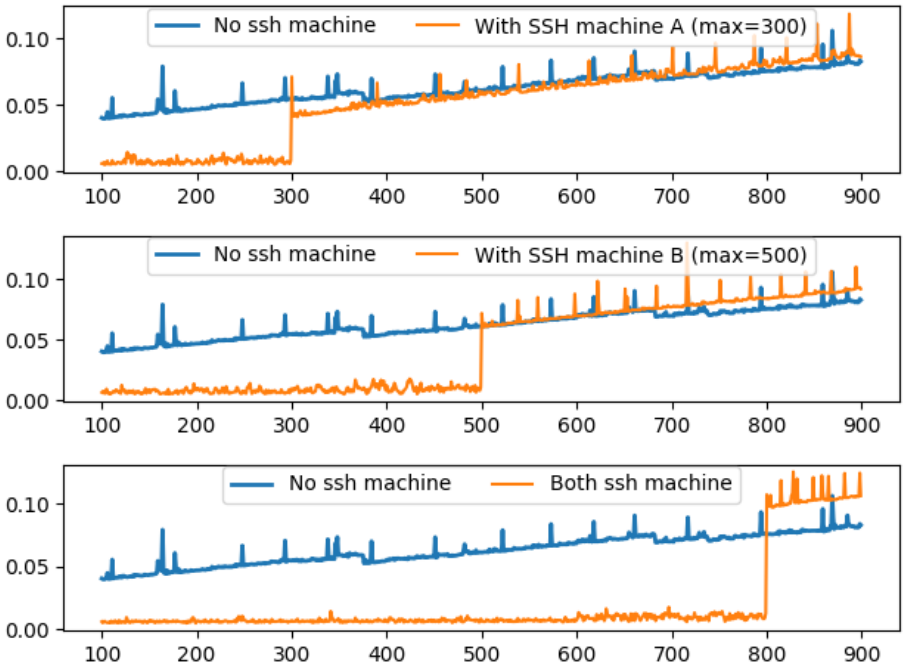

Figure 6.21.: The evolution of the creation duration, using the process tile

On the figure 6.20, the steps have the same behavior as those described earlier. For the figure 6.22, the stages are different. First the target at 600 on the last graph is absent. For the other graphs, the values of apparition are the same, but the behavior is different: in all the cases, the process time dramatically increases.

Interpretation: Three facts stand out. First, the process time dramatically increases for some steps. Second, the performance counter and the CPU cycle shows sudden level changes, which can be descending or uprising. Third, the duration to create increases linearly with the time.

The different steps in the process time occur when a SSH connection reaches saturation and cannot be used anymore. Thus, if only the machine **A** is used, the stage happens after the 300th creation, after the 500th creation for the machine **B** and after the 800th creation if both machines are used. Thus, the step happens when SSH stops being used. After this step the level is the same than when using only the local machine. As said earlier, creating an SSH instance is less resource consumptive than creating an instance of µPCN. The level increase is thus dued to this difference.

The different steps for the performance counter and the CPU cycle usage also coincide with the same values. When a SSH connection stops being used, the level changes. The change from one to no connection implies a decrease of the creation duration, whereas the deactivation of one SSH connection when two were in used increases the duration. In the first case, the fact that the lifecycle manager is faster without SSH connection is logical. As explained earlier, the verifications are less efficient when using SSH.

The fact that the lifecycle manager is faster when using two remote machine than using one or zero remote machine is more surprising. In this case, the server create the instances on the machine **A** one time in two, and one time in two on the machine **B**. Also, every command (but the instance creation) for one machine uses the same SSH connection. One hypothesis would be that using two machines at once reduces the frequency of usage of the connection. This hypothesis was not furtherly tested, because this behavior is not relevant in this analysis. Indeed, the goal of these graphs is to demonstrate if the application is scalable.

The augmentation of the duration of creation, when ignoring the steps, is constant. To create an

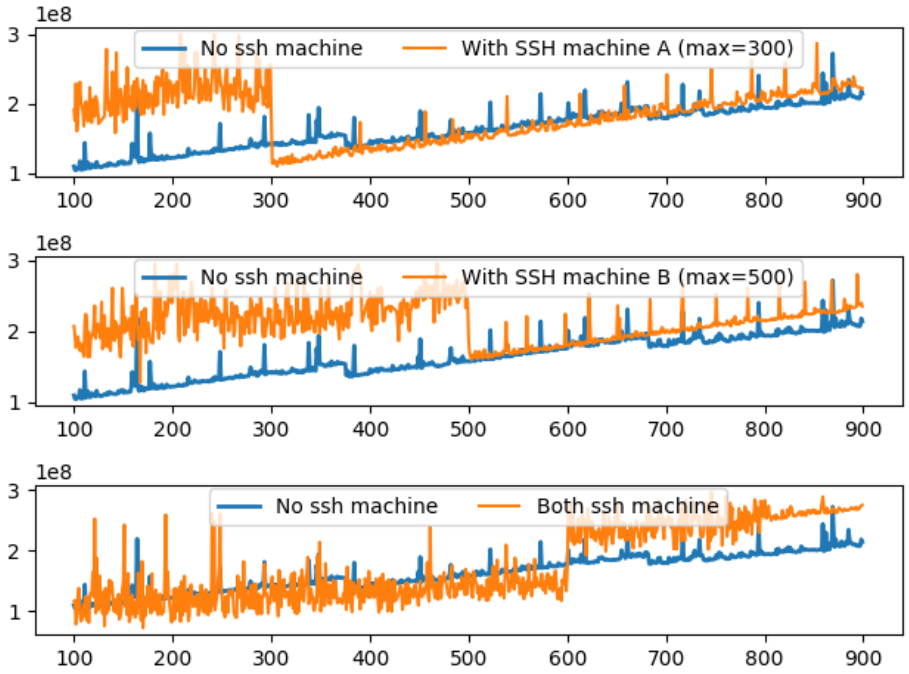

Figure 6.22.: The evolution of the creation duration, using the number of CPU cycles

instance, growing lists have to be read (in particular checking the port is not in use requires to go through the list of instances and the list of port in use by the system). If the number of instances grows, then the time to browse the lists also increases. Thus this increase is not surprising. This augmentation is light, the values double between the first creation and the last, which correspond to an augmentation of 0.0025% per instance.

Since the augmentation of the instance creation is light over 1000 creations, it can be concluded that the application is scalable enough to be used in the case of a Ring Road scenario.

6.3. The requirements completeness

The two previous parts were dedicated to test the functionalities of the application (6.1) and the performances of the application (6.2). With these elements, the requirements which were given on the beginning of this report (3), they can now be analyzed to check the completeness of this project. The table 6.2 summarizes the coverage of the different requirements.

Thus, the lifecycle manager is perfectly usable in a ring road environment, with the condition that the managed software can be called by giving the port as the last argument of the executable.

Requirement	Test	Coverage	Conclusion
/R1/: managing the lifecycle of the instances	6.1.2.1, 6.1.3	Finished	Every asked instance is correctly created; killing them works correctly.
/R2/: getting statistical information about the instances	6.1.1.3	Finished	The statistical information can be retrieved from the server.
/R3/: launching the instance via SSH machines	6.1.2.2, 6.1.2.3	Finished	The instance can be created on a SSH remote machine. The creation respects a maximum number of instances and a priority system.
/R4/: being scalable	6.1.4, 6.2	Finished	The performances of the application allow a usage in a reasonable time of the lifecycle manager. The application is scalable enough to be used in a ring road environment.
/R5/: having an extensible configuration	6.1.1	Finished	The managed software can be changed, several remote machines can be configured, the computed statistical information can be extended.
/R6/: being independent	6.1.1	Perfectible	The lifecycle manager can be embedded in any other software. The managed software can be changed, however the executable has to accept the port as argument and in last parameter.

Table 6.2.: The requirement completeness

7. Summary and outlook

During the introduction, two goals were defined. First, this project intended to extend the existing testing infrastructure in order to test µPCN in a ring road scenario. This architecture was shown in the chapter 4, and outlined three components: the software µPCN, a simulator and a lifecycle manager. The implementation of these components was defined in the chapter 5. The component µPCN received a tiny modification, in order to be able to create several instances on one machine. The lifecycle manager is a new project, and aims to manage the lifecycle of instances of µPCN. The simulator in use is The ONE, it was slightly modified in order to accept communications with µPCN and the lifecycle manager.

The scalability of the solution was the second goal. This part is mainly controlled by the component of the lifecycle manager, which was formally described in chapter 4, whose requirements were defined in chapter 3. Chapter 5.3 shows that a particular attention was paid to the performances of the application. Finally, the evaluation of this project provided in part 6 demonstrate that the API is scalable enough to be used in a bug ring road scenario.

The provided solution fulfills the initial requirements, but could be extended. First of all, the implementation provided for the simulator is only a prototype. For instance it is not possible to have several groundstations communicating with one instance of µPCN at the same time. This point can be improved. It can be interesting to add a new node in the network during the simulation. For the lifecycle manager, this software is more complete. One suitable modification would be stopping to use the port as primary key, an EID could be created instead.

Appendices

A. The format of µPCN messages

A message sent to µPCN shall have a specific format. This chapter will go through its specification. Two main type of messages exists: the commands (as `settime`, `mkgs`, ...) which aims to execute and action the instance of µPCN, and the data messages, which contains data (sending a bundle for instance). All the commands which were implemented are listed in the table A.3.

Every message has the structure defined in the table A.1. Every field but the corpus measures one byte. The values of the fields `BEGIN_DELIMITER`, `BEGIN_MARKER`, `END_DELIMITER` and `COMMAND_SEPARATOR` are defined by µPCN. The constant values can be found in the table A.2.

head	BEGIN_DELIMITER	BEGIN_MARKER	*type*
corpus	...		
foot	END_DELIMITER	COMMAND_SEPARATOR	

Table A.1.: A message format – text in upper case represents fixed data for every message, text in italics represents variant fields.

The field *type* depends on each message type. It is implemented by a single byte representing the command type. The type of the different commands can be found in the table A.3.

Constant name	value
BEGIN_DELIMITER	0x00
BEGIN_MARKER	0xFF
DATA_DELIMITER	':'
END_DELIMITER	0xFF
COMMAND_SEPARATOR	'\n'

Table A.2.: The value of the different constants

The field corpus can be as big as suitable. It is the payload of the message. In the case of a command, the command itself shall be put in this field. In the case of a data-message, it is simply the data.

Command's name	Meaning	type	corpus
mkgs	adds a groundstation to the instance, using its EID, its realibility and its Convergence Layer Adapter (CLA)	0x01	mkgs <EID> <reliability> <cla>
mkct	indicates a new contact to the instance	0x01	mkct <gs> <from> <to> <rate>
mkep	indicates that an endpoint is reachable from a groundstation	0x01	mkep <EID> <EP>
rmep	removes an endpoint from a groundstation	0x01	rmep <EID> <EP>
bundle	Sends a bundle to the instance	0x02	the serialized bundle
settime	Sets the time of the instance	0x06	the time to set
reset	hard resets the instance	0x0B	Empty

Table A.3.: The different messages used in this project

B. Installing and using the lifecycle manager

After describing the core of the application, the installation and utilization of the application will be described. It contains the three following parts: the installation of the lifecycle manager, the utilization of the lifecycle manager as a standalone project, and the utilization of the lifecycle manager combined with a simulator.

B.1. Installation

Installing the remote machine can be very quick. First of all, python3-devel (Fedora), python3-dev (Ubuntu) or an equivalent must be installed. *pip* is also needed, but may be installed at the same time as python.

It could be better to create a virtual environment in order to use this application. It provides an isolation for the application. Some extra modules will be installed later. Using an environment will only install these packages for the application, and will not pollute the remaining of the machine.

To create an environment, enter the command `pyvenv env` in a terminal, at the root of the application, where *env* is the name of the environment, any name can be chosen. Then the environment needs to be activated: `source env/bin/activate`. Once the work has ended, enter `deactivate` to exit the environment.

Then enter the command `make`. This will install every needed dependencies (flask, pyyaml, psutil, paramiko).

Three configuration files need to be filled:

B.1.1. The general configuration file

General settings can be set in the file `general.yml`. A sample example is provided in figure B.2a.

General µPCN information can be set in the object `µPCN`. The path to µPCN can be set in the field `µPCNPath` and the name of the executable can be set in the field `µPCNExecutable`. These two fields allow the remote controller to find µPCN on the local machine. They are also used as default values for other machines.

The default port can also be set in the corresponding field. The remote controller will first try to create an instance at this port. If the port is not available, it will be incremented until finding an available port.

The field `killInstanceOnExit` indicates whether the server shall kill the created instances before being shutdown. It accepts a boolean value. The default value is false.

Every log from the instances of µPCN will be written in the repository with the name `logDirectory`.

Figure B.1.: Description of the configuration files.

```
---
uPCN:
  uPCNDirectory: "/path/to/upcn/
  ↪ build/posix_local/"
  uPCNExecutable: "./upcn"
  defaultPort: 4200
killInstancesOnExit: True
logDirectory: "log/"
...
```

(a) Sample for file general.yml

```
---

VmRSS:
    description: "info"
    cmd: "ps -aux | grep {} | grep
{}"
    arguments: ["pid","ip"]
    cast: str

...
```

(b) Sample for file statistics.yml

```
---
-
  ip: "localhost"
  port:
  user: "user"
  priority: 2
  max: 5
  upcn:
    directory: "/path/to/upcn/build/posix_local/"
    executable: "./upcn"
-
  ip: "127.0.0.1"
  port:
  user: "user"
  priority: 1
  max: 10
  upcn:
    directory: "/path/to/upcn/build/posix_local/"
    executable: "./upcn"
...
```

(c) Sample for file remote.yml

Every path can be provided absolutely or relatively from the root of the application server. However, in the last case, the relative path of µPCN could also be used on remote machines as a default value (see the paragraph B.1.2 for more information).

B.1.2. The configuration file for remote machines

It is possible to specify remote machines so that the remote controller creates the instances using *SSH*. To do so, the user needs to add every machine to the list in `remote.yml`. A sample of this file is presented in figure B.2c. If remote machines are not needed, leave the file empty (but create it).

Every element represents one remote machine. The parameters `ip` and `user` are required. The others can be left empty, or may also not be present at all.

ip This field must contain the IP of the remote machine. It is required.

port This field represents the port needed for a SSH connection. If this field is not set, 22 is used.

user This field specifies the user name to use in order to connect to the remote machine using SSH. It is required.

priority This field represents the priority of the machine. If this field is unset, the default value is 1. If this field is set to a negative value (0 included), the machine will not be used.

max This field represents the maximum number of instances which may be launched on this machine. Once this number is reached, the machine will not be used again. If this field is unset or if the value is negative (0 included), no limit is set for this machine.

upcn As the field in the subsection B.1.1, this is separated in two fields: `directory` and `executable`. They represent the path to µPCN, and the name of the executable. If they are not set, the value set in *general.yml* is used. If only `executable` is set, the default value is used for this field as well.

B.1.3. The statistics configuration file

The file `statistics.yml` represents the different bash commands which can be executed when the user asks information about an instance. A sample of this file is provided in figure B.2b. Every command is represented by a YAML dictionary, containing the following fields:

description This field represents the description of the command, as it will be returned by the server. If this field is not provided, the name of the dictionary is used.

cmd This field represents the bash command in itself. It is possible to insert parameters using python3 formatting mini-language[1]. For basic usage, the first `{}` will be replaced by the first parameter. For more advanced usage, see the official documentation.

arguments This field must be an array. Every element of this array will be passed to the formater for the above string.

cast This field represents the cast of the result of the command. It is also possible to put here every python function which takes one string argument. The use of lambda may be useful here[2].

Once the application has been installed and configured, the correctness of the configuration can be tested by launching the server. The general and the SSH machine configuration files will be tested just after launching the instance.

[1] `https://docs.python.org/3.4/library/string.html#formatstrings`
[2] Checking if a result is equal to three would use this cast: `lambda x: int(x)==3`

B.2. Using the lifecycle manager as a standalone project

This part will present the utilization of the lifecycle manager. Of course, it has to be correctly installed and configured. A tutorial can be found in the previous part.

To launch the lifecycle manager, the user shall first activate the environment, if one had been configured during the first part. If an error appears indicating that modules are missing, it is probably because the environment was not activated. To launch it, the bash command `source <env_name>/bin/activate`, where `<env_name>` is the name of the environment, can be used.

Launching the server can be done with the command `make run`, at the root directory of the project. The launching phase requires some time to be launched, since verifications are launched on the configuration files. If an error happens during this phase, it is probably because the server was misconfigured. Once the sentence ** Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)* appears, the server can be used.

Starting now, the URL of the server (`http://127.0.0.1:5000`) will be omitted. Only the resource will be printed, starting from the `/`.

The different available routes are listed in the figure 5.1. They can also be accessed through the route `/help`. The quick list bellow will summarize the main possibilities:

- Creating an instance can be done through the route `/create` or `/create/at-port/<port>`. The first command creates a new instance, choosing itself the port, while the second tries to use the provided port of fails. A JSON file containing the IP and the port of the socket provided per instance will be returned;
- Creating several instances can be done through the route `/create/several/<number>`, where *number* is the number of instances to create. A JSON file containing the IP and the port for all the created instances will be returned;
- Recovering the statistical information can be done through the route `/<port>` where *port* is the port of the selected instance. A JSON file containing all the required information will be returned;
- Killing the instances can be done through the routes `/kill/<port>`, `/<port>/kill` or `/kill/all`. The two first commands kill the instance running under the specified port, whereas the last kills all the instance (including those not launched by the lifecycle manager).

To kill the server, a terminate signal can be sent.

Glossary

Bundle Protocol The Bundle Protocol aims to provide a message data structure which is compatible with a DTN. It contains information which may be lost during the transfer (date of expedition, sender, ...) and also information to check the integrity of a bundle (length, end of validation, ...). It also tries to minify the amount of used bandwidth. 5

CubeSat CubeSat are nano-satellites. Thanks to their low cost, it is possible for universities to send them into space. To reduce the price, they are very light, thus they can be embedded in other space mission as secondary payload. The OPS-SAT is an European Space Agency version of the CubeSat. 5

DTN DTN stands for Delay Tolerant Network (or Disruption Tolerant Network). It is a communication protocol which supports cut-off, on the contrary of TCP. Instead of aborting the request if the recipient is not reachable, the transitional nodes will store the data until the next node is reachable. 5

RESTFul REST stands for Representational State Transfer. It is an architecture type for web services. To be RESTful, an API must follow some constraints. The most important for the lifecycle manager is the stateless behavior: any command shall be doable in one HTTP request. 29

Ring Road Approach The Ring Road Approach is a telecommunication concept. The idea is to provide an Internet connection using a satellite constellation. The network on the planet is divided into several subnetworks, each having at least one groundstation, which can communicate with the satellite. A node connected to a satellite and to Internet is called a hotspot, whereas a node only connected to a satellite is a coldspot. 5

Bibliography

[1] C. Krupiarz et al. "Using SmallSats and DTN for Communication in Developing Countries". In: *59th International Astronautical Congress*. Glasgow, Scotland, 2008.

[2] *CubeSat Design Specification*. The CubeSat Program. 2014.

[3] K. Fall. "A Delay-Tolerant Network Architecture for Challenged Internets". In: *sigcom.org* (2003).

[4] Keith Scott and Scott C. Burleigh. *Bundle Protocol Specification*. RFC 5050. Nov. 2007. DOI: 10.17487/rfc5050. URL: https://rfc-editor.org/rfc/rfc5050.txt.

[5] M. Feldmann and F. Walter. "µPCN - a Bundle Protocol Implementation for Microcontrollers". In: *International Conference on Wireless Communications & Signal Processing (WCSP)* (2015).

[6] Daniel Ellard et al. *DTN IP Neighbor Discovery (IPND)*. Internet-Draft draft-irtf-dtnrg-ipnd-03. Work in Progress. Internet Engineering Task Force, Nov. 2015. 26 pp. URL: https://tools.ietf.org/html/draft-irtf-dtnrg-ipnd-03.

[7] András Varga and Rudolf Hornig. "An Overview of the OMNeT++ Simulation Environment". In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. Simutools '08. Marseille, France: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, 60:1–60:10. ISBN: 978-963-9799-20-2. URL: http://dl.acm.org/citation.cfm?id=1416222.1416290.

[8] Ari Keränen, Jörg Ott, and Teemu Kärkkäinen. "The ONE Simulator for DTN Protocol Evaluation". In: *SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. Rome, Italy: ICST, 2009. ISBN: 978-963-9799-45-5.

[9] Riccardo Böhm. "Bewertung von DTNRouting protokollen für Ring-Road-Netzwerke". Masterarbeit. Technische Universität Dresden, 2017.

[10] Davies and Doria. *D2.2: Functional Specification for DTN Infrastructure Software*. 2010.

[11] S. Burleigh. "Interplanetary Overlay Network: An Implementation of the DTN Bundle Protocol". In: *2007 4th IEEE Consumer Communications and Networking Conference*. Jan. 2007, pp. 222–226. DOI: 10.1109/CCNC.2007.51.

[12] C. Caini, A. d'Amico, and M. Rodolfi. "DTNperf_3: A further enhanced tool for Delay-/Disruption- Tolerant Networking Performance evaluation". In: *2013 IEEE Global Communications Conference (GLOBECOM)*. Dec. 2013, pp. 3009–3015. DOI: 10.1109/GLOCOM.2013.6831533.