Extending a SAIN Architecture Agent with Active Network Measurements

Auteur : Carlisi, Thomas
Promoteur(s) : Donnet, Benoît
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"
Année académique : 2020-2021
URI/URL : http://hdl.handle.net/2268.2/11524

University of Liège

Master Thesis
Extending a SAIN Architecture Agent with Active Network Measurements

Author:
Thomas Carlisi

Promoter:
Benoit Donnet

2020 - 2021

Contents

Abstract

In a context where Intent-based Networking is in fast evolution, techniques to provide service assurance are becoming more and more important. A project named SAIN, for Service Assurance in Intent-based Networking, was started and is still in the process of standardization. The University of Liège has developed the prototype of an agent of this architecture whose objective is to determine symptoms and health levels of the different sub-services of a network service.

This agent, named DxAgent, was functional but did not yet have any metrics to assess the health of network connections. Therefore, the objective of this thesis is to elaborate the implementation of active measures and thus to be able to assign a health score to the network but also to determine potential symptoms.

The measurement tool that has been chosen is the One-way Active Measurement Protocol (OWAMP), which has the advantage of providing one-way measurements. A python interface of the standard implementation of this protocol was then developed to allow the retrieval of the metrics.

Then this one was integrated to the DxAgent to extend it to new metrics and thus widen the agent's field of view. This extension proved to be a success and the DxAgent gets even closer to its original goal by allowing to discover symptoms proper to network metrics (delay, reordering, link failure, etc).

Finally, a use case has been developed to show how these additions work and how useful they can be. A topology based on Docker containers simulating the network of a company offering a streaming service has been set up. Moreover, a tool allowing to automate parameterized scenarios on this architecture has been developed. The results of the experiments show that the metrics and the detection of various symptoms work well in a near real-life context.

Acknowledgements

Throughout the duration of this master thesis as well as for my years at the university, I have received a great deal of support and assistance.

I would like to acknowledge the professors who accompanied me during these 6 years at the university and gave me all the necessary tools for my future life. I would particularly like to single out my supervisor, Professor Benoit Donnet, who never hesitated to help me and answer my questions throughout the duration of this work but also since the beginning of my journey. Also, I would like to thank Korian Edeline and Justin Iurman who guided me several times in this project.

Then I would like to thank the friends I have made in this university who have been good colleagues for many projects and especially my friend and partner in crime William Collin. Finally, I would like to thank my parents and my family who, unlike me, never doubted I would make it.

Introduction

In a context where network architectures are constantly evolving and diversifying, managing them has become a real challenge. Among these new ways of imagining the network, Intent-based Networking (IBN) has been in full expansion and constant evolution for several years. It is a paradigm that reduces human intervention in the management of the network to its strict minimum. Indeed, a network operator only needs to define an intent, in other words, to define the service he wants the network to produce. The intent is then sufficient for the system to organize itself in such a way as to correspond to the operator's request. Once the orchestration has been set up, the system must continuously verify that the intent is still respected despite potential events in the network in order to automatically readapt the system. This verification is an essential step to achieve an automatic and adaptive system. This is called service assurance and it is the main topic of this thesis.

The Internet Engineering Task Force is currently standardizing a fault detection concept for this type of system. The project is called SAIN, for Service Assurance for Intent-based Networking. SAIN divides a service (e.g. a tunnel between two peers) into several sub-services, i.e. into several parts of the network to be assured (e.g. a peer device). The objective is clear, facilitate the discovery of the root cause of a problem in the network thanks to its partitioning. By determining a health score and potential symptoms for each sub-service, SAIN aims to assure the operation of each sub-service independently. This makes it possible to react at the right place in case of a symptom discovered in a sub-service. This facilitates the automation of the IBN system to assure a service.

SAIN proposes a specific architecture to operate whose most important component is the SAIN Agent. The purpose of this agent is to collect and analyze as much information as possible about a network entity to be provided. The objective is to define a health score and potential symptoms of the sub-services that compose the machine studied by the agent. The University of Liege has developed a prototype of this agent called DxAgent. This implementation was already working correctly when I started my thesis but lacked a crucial source of information, the active measurement of connections between machines in the network. This information being very important to determine the state of a service to be provided involving several machines, it was then proposed to me to implement it.

The active measurement protocol that was chosen in priority is OWAMP (One-way Active Measurement Protocol). It has the advantage of providing one-way measurements. For example, it is possible to compute the delay between two machines in one direction or the other. This specificity of OWAMP makes it a very good tool since it is possible to have different path characteristics depending on the direction of the traffic. It should be noted that this protocol requires that an OWAMP server be present on the machine from which one wants to retrieve the metrics. However, the DxAgent has been developed in python and it exists only one implementation of the DxAgent and it is written in C. So I was asked to develop an interface in python that would communicate with the two components that are the client and server programs of OWAMP.

The interface that was then developed aims to provide functions to achieve everything that OWAMP is capable of providing. This project was developed independently of DxAgent and is publicly available on GitHub. To summarize, it is about configuring the server and managing the inputs and outputs of the topology programs. On the other hand, a scheduler allowing to perform one-way measurements on a number of machines at the same time at a regular interval has been developed. This functionality was primarily designed for the integration of the interface into the DxAgent, which was therefore able to be done smoothly.

The DxAgent has been modified to allow one-way measurements with a number of machines, with a wide range of configurable options. In addition, a probing tool has been added, the ICMP ping. It has the advantage of providing a second source of information, of being easier to access but also of being used as a reference value for the tests that have been carried out in the context of a use case.

Indeed, a use case has been set up. It is a topology representing the network of a small company providing a streaming service. The topology has been designed so that the streaming traffic goes through different paths depending on their direction. This network of 6 entities was simulated using interconnected Docker containers. Several experiments, which were facilitated by the creation of a tool developed in python to execute scenarios in an automatic way were thus carried out with this network. Their objective was to show how the DxAgent was able to react to the events we made the network undergo. Several reasons have motivated the development of use cases. First, it was a good opportunity to show that the new DxAgent metrics work and react well to variations. Secondly, it was an opportunity to adjust different parameters of OWAMP and DxAgent. Finally, it was a question of showing that the work that has been done in this project has practical uses in a case that is close to real life.

The results of the tests have been convincing and very close to what was expected. The DxAgent is now able to continuously measures link characteristics and then recognize variations in the initial state of the network and therefore provide accurate symptoms in case of problems.

This work is divided into 4 chapters that we will briefly present. The objective of the first chapter is to define the context and the theoretical basis of this work. First, we will briefly explain the concepts behind Intent-Based Networking. More specifically, we will explain what characterizes such a system with a slight focus on intent. More importantly in this work, we will show the key steps of the IBN life cycle to highlight an important notion: assurance. Then, it will be an opportunity to explain the SAIN project and how it intends to assure complex network structures. An example of a tunnel service has been integrated into the explanations in order to illustrate the different concepts. Finally this chapter will be concluded by a presentation of the SAIN Agent prototype developed by the University of Liège which is the DxAgent.

The second chapter will present the OWAMP protocol that has been chosen to serve as the active network measurement for DxAgent. We will see that the protocol is divided into two parts: Control and Test. Understanding how these protocols work will be very useful when it comes to analyzing and understanding the results of the use case experiments that will be done. This chapter concludes with the presentation of the complete sequence of actions and messages sent by the protocol.

Chapter 3 aims to explain the implementation that has been done in this work. This includes explanations of what was done but also the questions we asked and the choices we made. We will first see the already existing implementation of OWAMP and the way it was interfaced in python. Then, the extensions that have been integrated into DxAgent will be explained, including the addition of active measures such as OWAMP and traditional ICMP ping. Among other things, we will highlight the way in which continuous measurements have been made while updating the DxAgent metrics. In addition, we will see how these changes can be used and how they will impact the discovery of symptoms of a system.

The fourth and last chapter will present a use case to show the efficiency and usefulness of the new DxAgent inputs in a real life case. Then we will explain how the different entities of the Docker topology that has been imagined have been implemented but also how they have been connected. Then we will see a tool that has been developed in python to allow the automation of parametrized scenario execution using the topology. To finish this chapter, we will highlight the use of the DxAgent through four experiments that will be presented and analyzed.

Chapter 1

Service Assurance for Intent-based Networking

In this chapter, divided into three parts, we will see the context in which this work is integrated. First of all, we will present what Intent-Based Networking is, since this is the type of network targeted in this thesis. Then we will explain how networks using this paradigm assure that their services are working properly. And finally, we will conclude with an existing implementation of service assurance on which I am involved.

1.1 Intent-Based Networking

In short, Intent Based Networking (or IBN) allows a network administrator to formulate the desired state of the network in a high-level manner and have the network orchestration done automatically. This network administration is a new research topic in constant evolution. The first attempts to formalize it by the IETF (Internet Engineering Task Force) date back to 2017 and a last draft [11] was released in February 2021. This last document will be used to explain the different principles of this new concept because there is no standard definition yet. And it is not only in formalization that IBN is new, it is also in practice. For example, CISCO started developing their first IBN system in 2014 and it will take until 2017 for it to be operational. But they are far from stopping there, every year Cisco highlights their advances in their IBN systems. This is a topic that has yet to show its full potential and is therefore constantly evolving. From a marketing point of view, the IBN market exceeded USD 900 million and Global Marketing Insight estimates growth to USD 4.5 billion by 2026 [9].

Device-by-device network administration is no longer a possibility today. For years, the complexity of networks has required more centralized, high-level, and automated configuration. In this context, IBN is trying to provide a solution to configure a network as a whole but also to adapt to all its changes. But of course, this network automation still requires an intention, an input. How should the network behave, what should it look like, this is called the intent. This intent is one of the main points that differentiates IBN from other existing models such as Software-Defined Networking [33] or Policy-Based Management [32] administrations. We are going to see what this intent is really about and what is its characteristics.

1.1.1 Intent and Intent-Based Management

Intent, as defined in the RFC7575 [12] on Autonomic Networks is "an abstract, high-level policy used to operate a network". Even if Intent-Based Management wants to be as autonomic as possible, it requires an intervention from outside. This intervention, this intent, is a statement of what the network should meet and what it is supposed to deliver without specifying how it should be done. The most important point is that an intent should only specify if there is something to be done and not how to do it. Furthermore, an intent should keep a certain data abstraction, operators should not take into account the low level configuration of devices. The IETF draft offers several examples to better understand what an intent is. For the sake of clarity, these are presented in natural language. Here are two examples of intents:

- "Avoid routing networking traffic originating from a given set of endpoints (or associated with a given customer) through a particular vendor's equipment, even if this occurs at the expense of reduced service levels."
- "VPN service must have path protection at all times for all paths."

On the other hand, we can also give an example of a statement that cannot be considered as an intent:

- "Configure a given interface with an IP address."

This type of rule cannot be considered as an intent, there is no data abstraction, it is a simple device configuration. Any form of raw configuration or requests explaining how to get to the desired result are not considered as intent. It is the role of the IBN to translate an intent into a series of configurations. Ideally, this translation should be done independently in each machine and each machine must know its role to comply with the intent. In practice, it is easier if some functions are centralized.

Once the intent concept is integrated, we can put forward the different principles of Intent-Based Networking.

1.1.2 Intent-based system principles

The different principles that will be stated are essential to characterize the intent-based nature of a system.

1. **Single source of Truth (SSoT).** SSoT represents the set of intent expressions that have been validated. Since this represents the desired state of the network, it is easy to compare it to the current state. Thanks to the SSoT, it is possible to know what

actions need to be taken to reach the objective network. Also, it is useful to validate a network that has been modified and therefore to ensure that the objectives have been met, in short, the SSoT is the invariance of the intent.

2. **One-touch but not one-shot.** We can say that the intent-based system is "one-touch" because the user will express his intent and then the system will take care of the rest. However, we cannot say that it is a "one-shot" process because it requires several steps before arriving at a well formed intent. Indeed, the user's expression may contain implicit or imprecise parts and it is up to the system to refine the expression to make it valid. In some cases, the system can propose choices or alternatives to the user to clarify his request. Finally, the system must of course ensure that the initial intent does not have any conflicts.

3. **Autonomy and supervision.** It is important for an intent-based system to offer some flexibility to the user to formulate his intent. One can imagine a web interface but also dictate aloud in natural language with a word recognition tool. Moreover, the system must be as autonomous as possible, it must be able to perform a maximum of tasks and operations without requiring the intervention of a user. But it must also provide the right level of supervision to satisfy the user's needs and report the relevant information.

4. **Learning.** An intent-based system is a learning system. It should be remembered that once the user has given an initial intent, the system must reason. And this reasoning is done by means of learning. This ability to learn is as useful to refine the intent as to optimize the way the intent will be rendered. Moreover, the system can also predict a change in intents or in network conditions, that is why the system is in continuous learning and optimization.

5. **Capability exposure.** It is important that the network is expressive enough in terms of its capabilities, requirements and constraints to be able to respond to the demands of the intent.

6. **Abstract and outcome-driven.** As already stated, the user should only focus on what he wants and not on what has to be done, i.e. on the outcomes.

An intent-based system can also integrate other principles than those described above but we have seen the main ones. We will now move on to the functionalities of intent-based Networking.

1.1.3 Intent-Based Networking functionalities

Intent-based Networking integrates a wide range of functions that can be divided into two categories, which are **fulfilment** and **assurance**, which are themselves separated into different subsets. A representation of these functions and the order in which they are performed is shown in the figure 1.1 below.

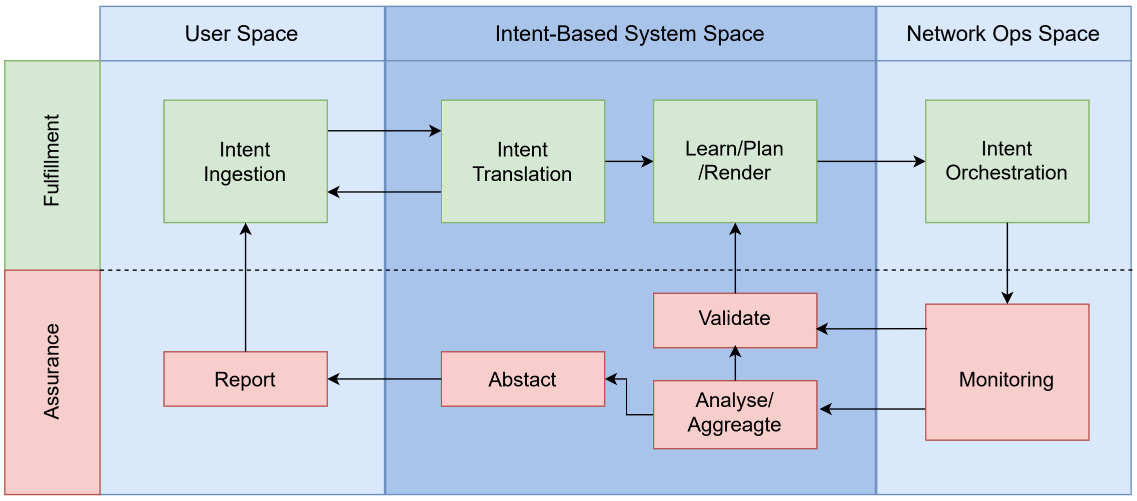


Figure 1.1: Intent-based Networking functions

An explanation of the different concepts of this figure will now be explained starting with the functions themselves. The order in which they are listed represents the logical course of events, although cycles can obviously occur.

- **Intent fulfillment.** These are the functions and interfaces that allow you to retrieve a user's intent and perform the necessary actions for the fulfil. These functions also include the orchestration of configuration operations on the network.
  - **Intent Ingestion.** These are functions to recover the user's intents through interactions with the system to establish a clear intent. Research is underway in this area to make these interactions as natural as possible, so that the system is able to understand the user better and better.
  - **Intent Translation.** This is the step that translates the defined intent into a series of actions and network requests to reach the desired state. Moreover, these functions must be able to learn to choose the best solution, the best heuristic, the best path, ...
  - **Intent Orchestration.** It is simply the actual configuration and provisioning that needs to be orchestrated across the network. This is the previous steps that have determined what needs to be done in this part.
- **Intent Assurance.** These are the functions that allow the user to validate and monitor the network to ensure that the intent is properly complied. These functions also include the analysis of the effectiveness of the fulfillment actions so that the fulfillment process is trained and optimized over time. In addition, assurance is used to verify that "intent drift" does not occur. An intent drift is the fact that the system that evolves over time ends up losing its efficiency to fulfill the intent or worse, do not meet the intent anymore. Assurance functions are actually the ones that interest us most about IBN since this is what my work focuses on.
  - **Monitoring** A first set of assurance functions are used to monitor and observe the network for its events and performance outliers. It is also a question of performing measurements to ensure that the services are working properly. These functions serve as a basis for the assurance process.

- **Intent Compliance Assessment** These are functions that will compare the monitored network behavior with the intent held by the SSoT. These functions continuously assess and validate the effectiveness of the intent fulfillment actions. These functions also observe if an intent drift will not occur over time.
- **Intent Compliance Actions** Once an intent drift occurs or the network behavior is in an inconsistent state, a series of corrections must be applied: this is the purpose of these functions. Alternatively, the system can send an alert to the user to act accordingly. For example, the system could propose a new, slightly modified intent that would no longer pose a problem in the new network state.
- **Abstraction, Aggregation, Reporting** These are the functions that will report to the user the outcome of the Intent Assurance so that the user can relate this information with the intent. The low-level setting statistics retrieved in the previous steps must be up-leveled because the user should only have to deal with high-level concepts. The aggregation and analysis features should serve to provide a report on the intent compliance status but also provide an adequate summary and visualization of the data for human users.

The figure 1.2 provided below better express the life cycle of Intent-based Networking by highlighting the two loops it contains.

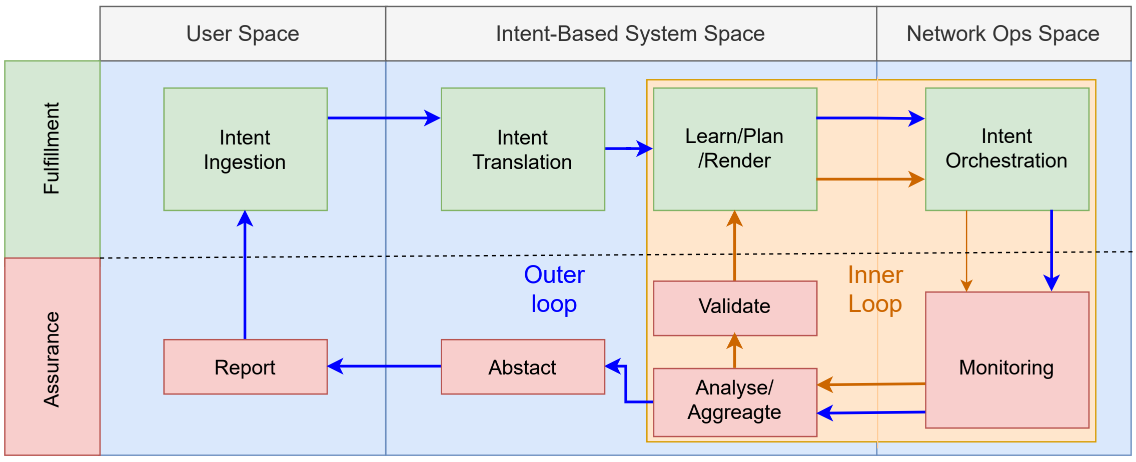


Figure 1.2: Intent-based Networking Life Cycle

We have the **inner** loop which does not intervene in the User Space and which does not involve any human. This loop represents the automatic analysis and validation of the intent based on the analysis of the network operations space. The render functions can then adjust some parameters to reconfigure the network devices if necessary. This loop, in addition to avoid intent-drift, allows the system to continuously improve itself without external input.

Then, the **outer** loop, which extends to the User Space, involves a human since some feedback is sent to him. It is also possible for an operator to adjust the original intent based on this feedback. An intent then has a rather eventful life cycle; it is constantly modified, adjusted and retracted by these two loops to best suit the service it wants to provide.

In fact, this architecture has as many functions allowing to automate the implementation of the intent as functions allowing to ensure that it is well respected. Assurance is an essential part of this new paradigm and that is what this work focuses on.

1.2 Service Assurance for Intent-based Networking

From now on, it is to be expected that companies that want to launch a service on their network may be interested in Intent-based Networking. As we have seen, this makes the work of the network operator much easier, as user-network interactions are minimal. In this context, it is also very important to be able to offer a service assurance that matches the advantages of IBN. Therefore, we will look at a service assurance that is particularly well suited to this type of administration: Service Assurance for Intent-based Networking or SAIN. SAIN is a new project that is still evolving and is not yet standardized. Therefore, the theoretical concepts we will see come mostly from an IETF draft [13].

When a network operator decides on a service (e.g. by proposing an intent to the system), the process of configuring the network is started and the system configuration will be implemented. However, the fact that a configuration is applied does not mean that the service works correctly. It is necessary to monitor both the configuration and the operational data of the service.

Observability has became something essential and well spread as we can see when we look, for example, at the growth of the cloud monitoring market (which was predicted to grow from $723.8 Million in 2016 to reach $1,976.9 Million by 2022 [19]). But it remains difficult for operators to correlate service degradation with the network root cause. Being able to warn the operator of a service slowdown or malfunction is a first step, but it is also important to know the cause. In fact, it is even more useful to know the opposite, i.e., which services will be impacted if a certain network component were to malfunction. For example, "which application would be impacted by an ECMP imbalance?"

To be able to do such things, SAIN works from an **assurance graph**, deduced from the service definition and the network configuration. The root of this directed acyclic graph is the service to be assured. While the children represent the subservices directly dependent on the service. A subservice corresponds to a subpart of the network (e.g. a physical interface) that must be assured. Moreover, these subservice components can themselves have children, in which they depend, representing subservices that are even more precise. In order to better understand graph assurance, we can show a simplistic example for a tunnel service.

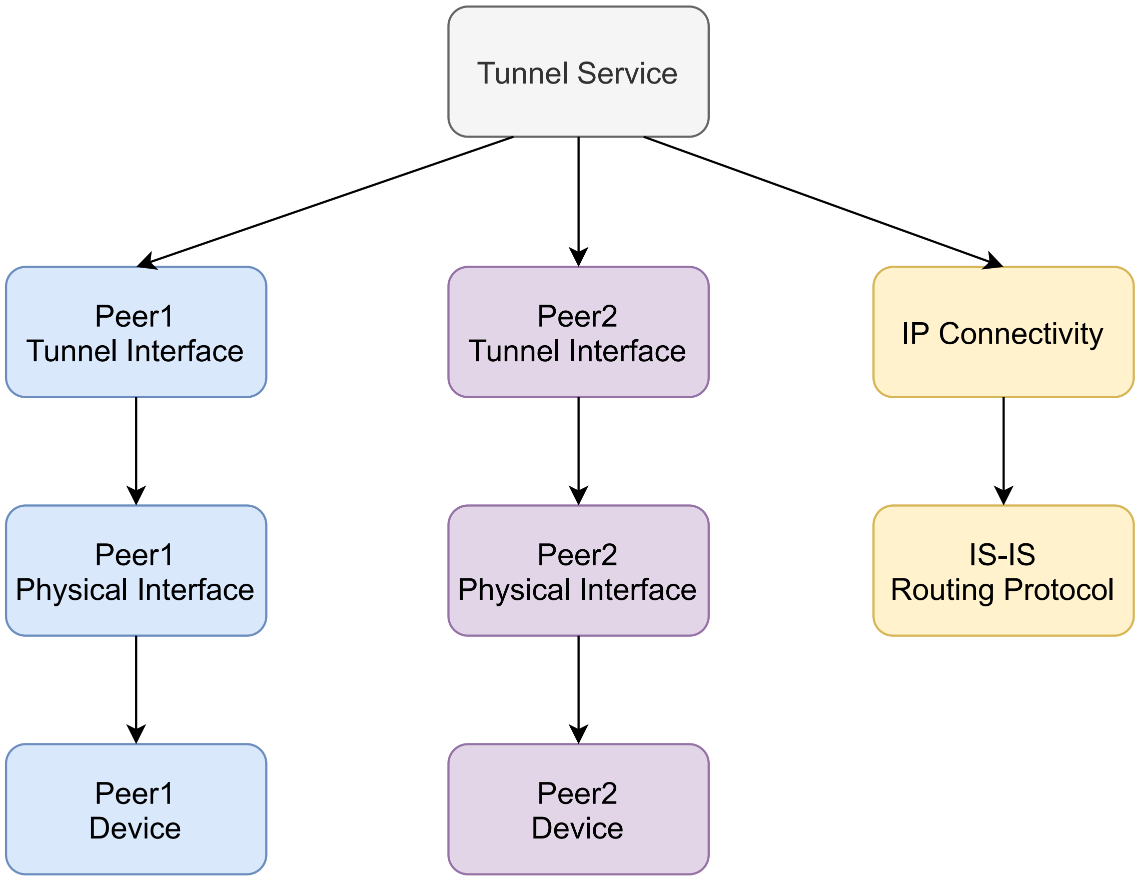


Figure 1.3: Assurance Graph - Tunnel Example

We can see on the figure 1.3 that the Tunnel service depends on the two peer interfaces of the tunnel but also on an IP connectivity which, itself depends on its routing protocol. We can add to this that the peers' interfaces depend on their physical interfaces which are themselves dependent on the devices behind the interface. This example of tunnel service will be used throughout this section dedicated to SAIN.

One main principle of the SAIN architecture is to maintain a correct assurance graph despite possible changes in services or network conditions. The SAIN framework is then able to highlight the problematic component in the graph when a service is degraded. The hierarchy of assurance graph helps to correlate a service degradation with the network root cause. Also, SAIN must be able to deduce the number and type of services that will be impacted by a component problem. This information is very useful because, in addition to giving easily usable information to the operator, it facilitates a possible autonomy of the system. These are the very basic concepts of SAIN and we will now detail its functionalities.

1.2.1 Presentation of SAIN and its architecture

The goal of SAIN is to make sure that a service is working well and if not, to specify what is wrong with it. This is done by means of a **health score** (an integer between 0 and 100) calculated by SAIN for each service instance. If we go back to our tunnel example, we can reduce the IP connectivity health score if the delay between the two peers has been degraded. If the score is not maximal (i.e. different from 100), it means that the service is not fully operational and a symptom is then output to explain this score. The symptom combined with the score is called the health status. The health status is the final product of a series of steps that we will briefly present.

As explained, the service is decomposed into an assurance graph formed by subservices linked by their dependencies. After that, each subservice (i.e. each assurance graph node) is transformed into an **expression graph**. This is a graph that details how to retrieve the metrics of the devices and how to compute the health status of these subservices. We will see in section 1.3 that in practice, it is not necessary to use a graph to retrieve the metrics, we can simply list the metrics we want to observe and the way to retrieve them. For example, the peer device subservice can get the `CPU idle time` metric by reading at `/proc/stat/` file. Then, the expression graphs of the subservices are combined in order to obtain a service expression graph allowing to compute the health status of the service. Finally, a **global computation graph** is built by combining the service expression graphs. The global computation graph simply represents how to combine the metrics of the subservices to provide the health score of these subservice. This global computation graph encodes all the operations necessary to produce the health status of all services from the collected metrics.

Before detailing all steps of SAIN, it is important to present its architecture which is presented in figure 1.4 just below.

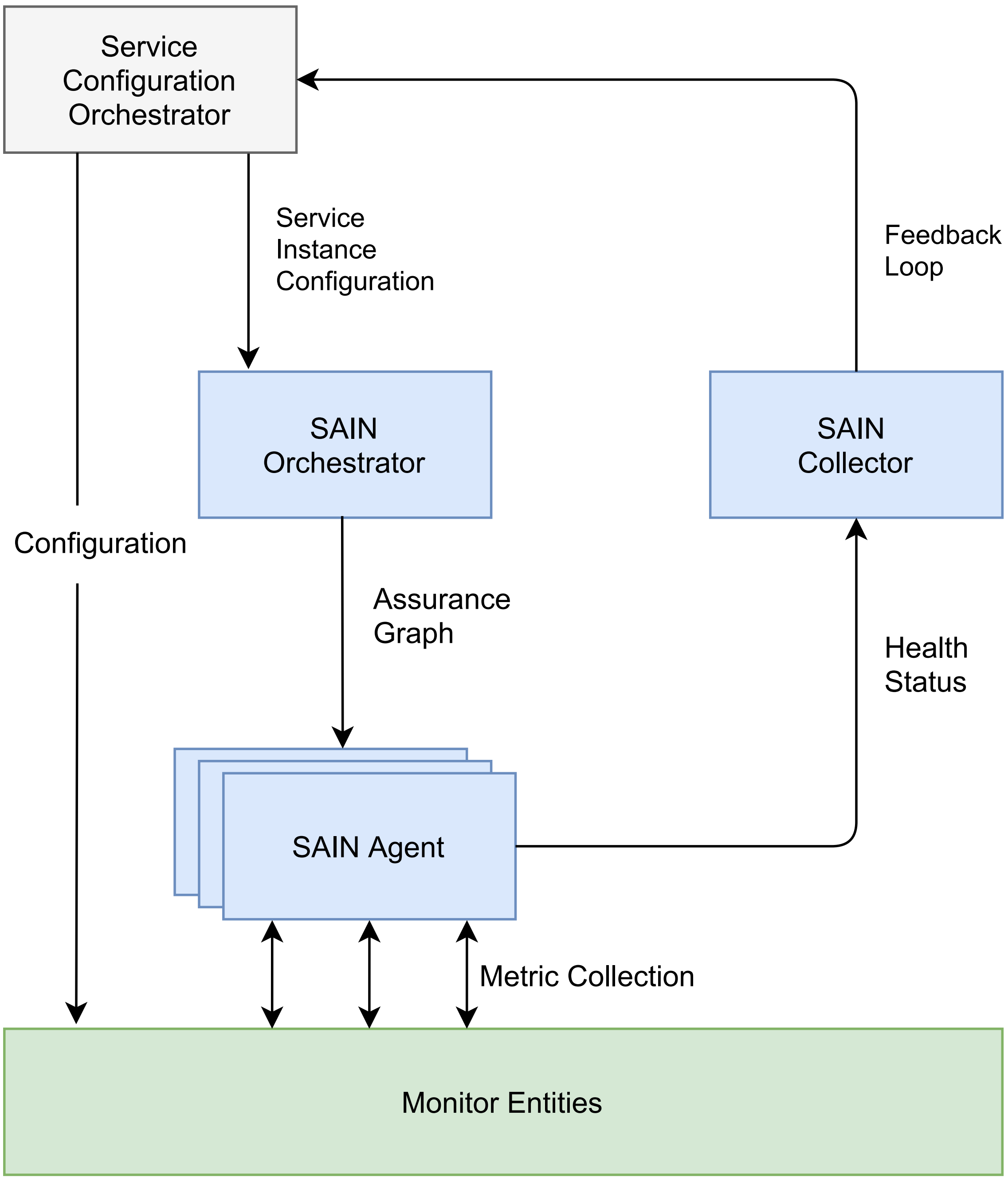


Figure 1.4: SAIN Architecture

SAIN is composed of serveral component which are the following :

- **Service Configuration Orchestrator.** This is the system that implements the configurations to perform the service setup. Typically, it is an Intent-based system.
- **SAIN Orchestrator.** The component of SAIN that retrieves the configurations of service instances and then convert them into an assurance graph.
- **SAIN Agent.** It is a component allowing to communicate with one or several

devices (or with another agent) in order to build the expression graph by performing the necessary computations.

- **SAIN Collector.** This is the part that retrieves the output of the agents to display it in a user friendly form (or process it locally).

From the figure 1.4, we can summarize the sequence of operations. First of all, the service is obviously configured by the Service Configuration Orchestrator on the devices and the configurations in question are sent to the SAIN orchestrator which will deduce the assurance graph. The SAIN agents retrieving the assurance graph must then build the expression graph and calculate (in distributed manner) the health statuses. These statuses that will be retrieved by the collector will be properly displayed to the user. And finally, in order to guarantee a closed automation loop, the SAIN collector must send a feedback to the Service Configuration Orchestrator which may be able to handle the problem.

1.2.2 From Assurance Graph to Expression Graph

As we have seen, a first step in assurance is to decompose services into subservices. When we refer to subservice, it also implicitly includes its assurance. A subservice, in addition to corresponds to a subpart of the network is also characterized by a list of metrics and computations that allow to infer the health status.

The decomposition into subservices is an important step in SAIN, for several reasons. First, it allows to provide a relational picture of a service instance. But also, it allows to separate the service into a series of different domains of expertise. This makes it easier to call the right person or the right tool for the specific subservice that is causing the problem. Finally, it is very likely that a subservice is used by different services, and we can therefore reuse the computations already made for a subservice in the context of another service.

It should be noted that the assurance graph must be maintained in all situations. When a service is added, modified or deleted but also if there is a change of configuration in the network. Moreover, another job that a SAIN orchestrator must provide is to detect what is the intent to ensure. If we go back to our tunnel example, it is indeed necessary that the peers are functional but that is not all. A VPN is expected to have a certain latency and bandwidth requirement and therefore these elements must also be part of the components to be assured. This operation is facilitated by the fact that the Service Configuration Orchestrator and the SAIN orchestrator are most likely combined.

Once the assurance graph has been created, the SAIN agent can now use it to build the graph expression. The first step is to transform each subservice instance into a set of subservice expressions. These expressions take as input metrics and constants and produce as output the status of the subservice (based on some heuristics). Then, for each service instance, the expressions of the subservices on which a service depends are combined in such a way as to build the expression of this service. The way the expressions are combined depends on the type of dependency. We distinguish two types of dependencies: impacting and informational.

- **Impacting:** This is the type of dependency where the score impacts the score of the parent. In this case, the symptoms are taken into account for the parent (sub)service as an impacting reason for the score drop.

- **Informational:** This is the type of dependency where the score does not impact the parent's score. However, the symptoms must be taken into account for the parent as an informational reason.

Finally, a global computation graph is built from the service expressions, it is then the graph encoding all the operations necessary to produce the health status from the metrics. This last step concludes the theoretical explanations on Service Assurance for Intent-based Networking and we are going to see the implementation that has been done and on which I added a new service analyse.

1.3 An implementation of the SAIN Agent

The University of Liège has launched a project to implement one of the component of SAIN. It is the SAIN agent which, it should be remembered, the agent responsible for calculating the health scores by monitoring the devices useful to a service. This implementation project, developed in python, is called DxAgent[1] and is developed by Korian Edeline. His work is available on the following github: `https://github.com/ekorian/dxagent`. This implementation respects the concepts of the draft that has been presented but some choices as well as the implementation of unspecified elements deserve to be studied in more depth.

The first thing to note is that this agent can be used independently of the other SAIN components. Indeed, instead of monitoring only the subservices proposed by the assurance graph sent by the SAIN Orchestaror, the DxAgent retrieves a maximum of data on the device that runs the program. Therefore, this program could very well be used to know the health status of any linux device. One just need to run the program on the machine one want to analyze and a daemon will run to continuously analyze the device information. The recovered data is then used to calculate the health status which can be displayed through two different interfaces.

The operation of the DxAgent program is divided into 3 key steps which are as follows:

1. Retrieve input from several sources
2. Normalize data and build the service dependency graph
3. Determine health statuses from user-defined rules

To simplify the explanation of the implementation in the next sections of this document, we will use the same names as those used by the DxAgent. These three steps are respectively called *Inputs*, *Metrics* and *Rules*.

[1]Dx is an abbreviation for Medial Diagnosis

1.3.1 The three steps

Even though the three steps will be a bit detailed, only the high level operation will be explained. If some implementation details need to be highlighted, they will be explained in the section 3 which is about implementation.

1.3.1.1 Inputs

The first step is to retrieve the data needed to determine the state of the services. This data can be retrieved from both physical and virtual machines. This option is obviously very important since virtualization is more and more present in today's networks. As for the recovered data, it comes from various sources. For example, a certain amount of data is extracted from the linux folder managed by the kernel: `/proc`. Like, for example, information about the disks, the state of the system, the available network interfaces, etc. Netlink is also a good source of information when it comes to network activities. Other sources of information are used by the program but the goal is not to provide an exhaustive list. However, it is interesting to mention that a new input will be added in the context of this thesis: OWAMP. Indeed, details on the integration of this input will be revealed during the section 3.

The purpose of this crucial step is to recover a lot of raw data from the computer. This data can be for example the number of packets received on an interface, the temperature of the computer components, the CPU usage, the current idle or sleeping process number, etc. At the end of this step, we find ourselves with a large amount of raw data and it is the objective of the second step to bring order to it.

1.3.1.2 Metrics

Once the information is retrieved, it is possible to determine the set of subservices running on the machine. For example, the list of active network interfaces is retrieved in the first step, so we can determine that each interface is a subservice. The same applies to the list of processes, storage disks, etc. Of course, as presented in the theoretical concepts, the dependencies must also be determined. We have, for example, the `CPUS` subservice being the parent node of all the `CPUx` subservices associated with each CPU of the machine (with `x` being the id associated with a CPU in linux). The dependency graph of the subservices is then built during this step. We can notice a difference with SAIN which is that the dependency graph is retrieved by a discovery of the DxAgent (i.e. the SAIN agent). While in the SAIN draft, we have seen that the Assurance Graph (which is similar to the graph built here) is built and sent by the SAIN Orchestrator to the Sain Agent. Moreover, the DxAgent does not need to explicitly recreate an Expression Graph. Here, we have a single graph representing the dependencies between subservices and the health scores are calculated in each node using rules. We can see an example of a dependency graph that DxAgent could build on a machine with two CPUs, two storage disks and two network interfaces in the figure 1.5 below. This is actually a part of the graph, the node `Bare Metal` (representing the fact that it is not a virtual machine) has other subservices in the original program graph.

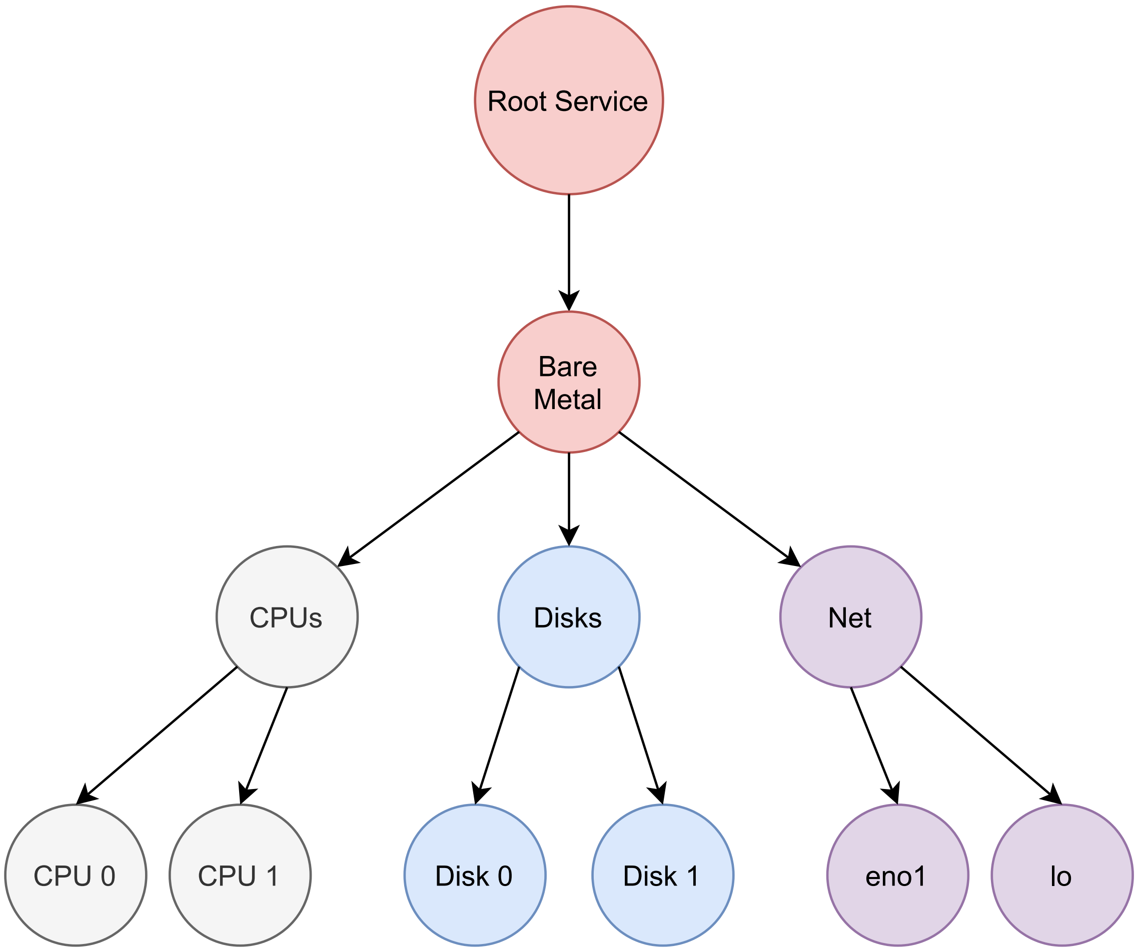


Figure 1.5: Example of a Dependency Graph

This second step of the DxAgent is also characterized by the normalization of the extracted data. It is important to gather data from different sources under the same variable and the same unit. For example, the number of bytes sent by an interface is retrieved from different sources depending on whether it is a physical or virtual machine. Since the next step is to create rules from the metrics, we only need one variable, in other words, one single metric that we will call `rx_bytes` for example. This example can be found in a DxAgent csv file:

```
rx_bytes,if,int,MB
```

This means that this metric belongs to the subservice `if` (for interface), that it is an integer and is expressed in Megabytes.

1.3.1.3 Rules

Finally, once the service dependency graph and the metrics have been retrieved, it is possible to calculate the scores and find potential symptoms. The way to determine these elements is to use user-defined rules written in a config file (with a csv format). These rules, having a rather high level of abstraction, allow to examine the metrics to ensure the normal functioning of the monitored machine. A rule is composed of 4 elements:

1. The Symptom Name.
2. The path to the subservice in the graph.
3. The Severity. There are two possible values : Red (score decrease of 50%) or Orange (score decrease of 10%).
4. The condition to be evaluated

Here ia an example of rule that has been defined in the project:

```
"Low Fan Speed",/node/bm/sensors/sensor,Red,input_fanspeed<100
```

This means that a "Low Fan Speed" symptom will be triggered on a "sensor" subservice with a "Red" severity if the fan speed is below 100 RPM.

Two other examples:

```
"No free memory available",/node/bm/mem,Orange,free<50
"No free memory available for 1 min",/node/bm/mem,Red,1min(free)<50
```

Here we check if there is still memory available in a subsevice "mem" (a RAM). The severity will be "Orange" when there is no more memory available but if it lasts for at least 1 minute, it will be "Red". We can also notice the use of the 1min() function. It is indeed possible to use several such constructs to define the rules. This construct of course wants to make sure that the condition is true for 1 minute.

These rules can be easily created or modified to meet some network operator requirements while keeping a high level of abstraction. Once the calculated scores of the subservices are at the bottom of the tree, they must be propagated to their parent node according to the dependency link (informational or impacting). Each node, from the leaf to the root service, has then a health score (and a symptom if the score is not maximal).

The three steps that we have seen are continuously performed by the DxAgent daemon in such a way as to guarantee the status of the services.

1.3.2 The interfaces

The DxAgent project is actually composed of 3 programs: DxAgent, DxTop and DxWeb. The architecture of the project is presented on the figure 1.6 below.

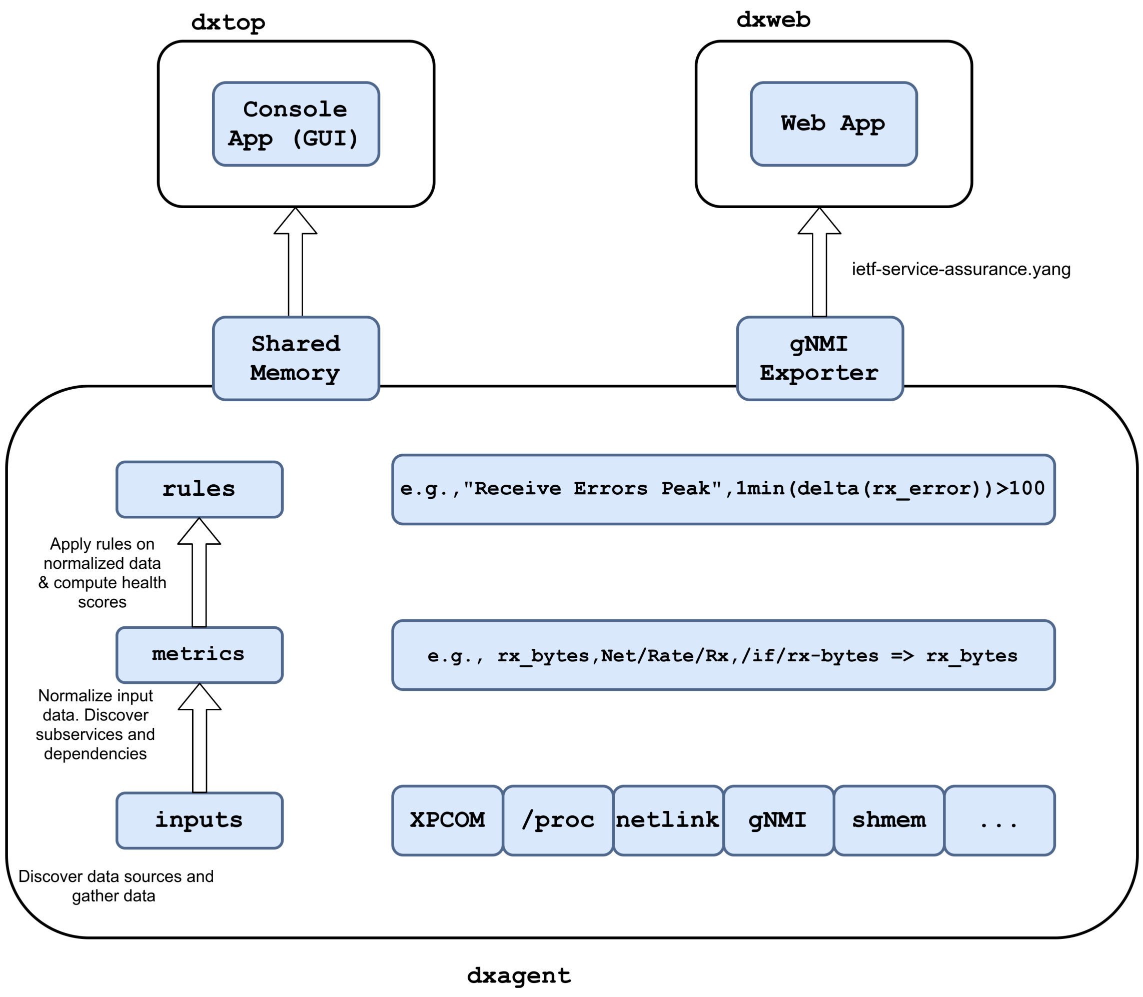
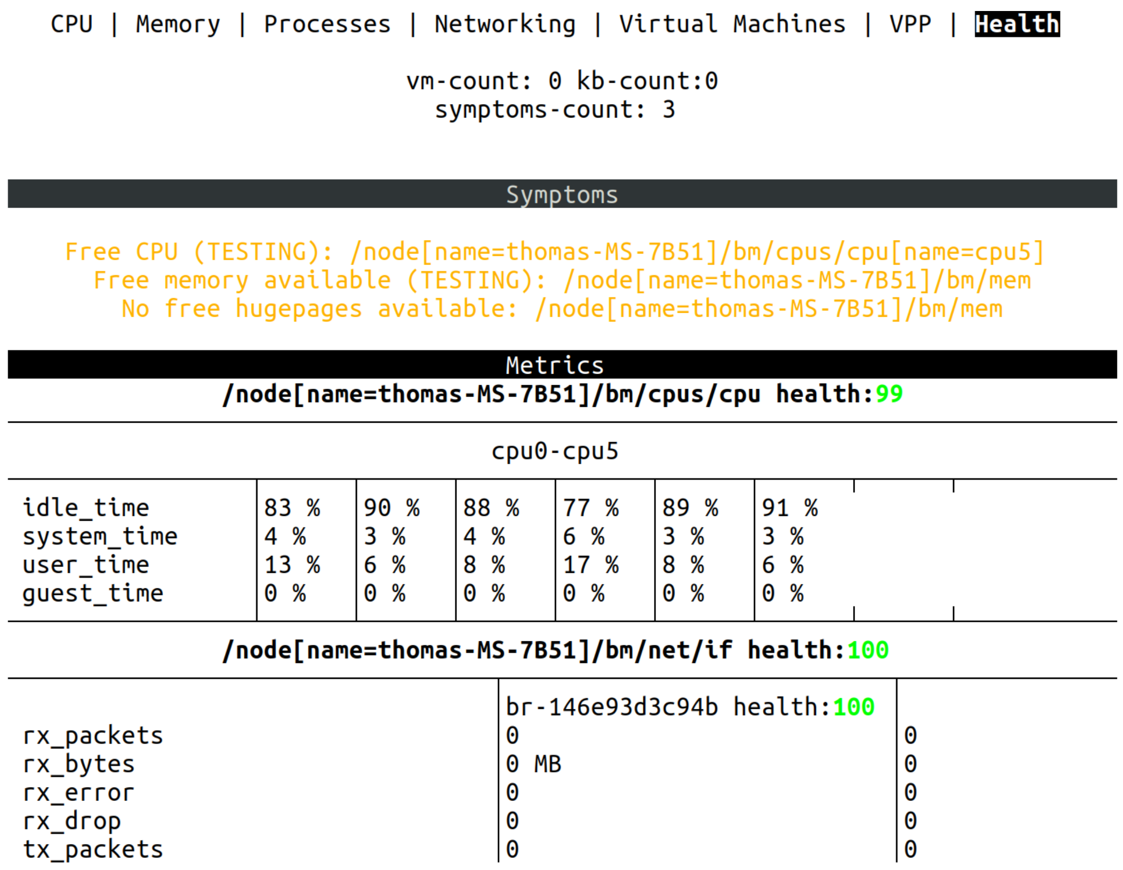

dxtop
Console App (GUI)
dxweb
Web App
ietf-service-assurance.yang
Shared Memory
gNMI Exporter
rules
e.g.,"Receive Errors Peak",1min(delta(rx_error))>100
Apply rules on normalized data & compute health scores
metrics
e.g., rx_bytes,Net/Rate/Rx,/if/rx-bytes => rx_bytes
Normalize input data. Discover subservices and dependencies
inputs
XPCOM
/proc
netlink
gNMI
shmem
...
Discover data sources and gather data
dxagent

Figure 1.6: DxAgent Project Architecture [7]

Since the DxAgent runs in the background, it cannot be used as a user interface. That is why two interfaces (DxTop and DxWeb) have been implemented.

DxTop uses shared memory to communicate with the DxAgent to get useful data to display. You can see a representation of this program running on my machine:



```
CPU | Memory | Processes | Networking | Virtual Machines | VPP | Health

                      vm-count: 0 kb-count:0
                        symptoms-count: 3

                             Symptoms

  Free CPU (TESTING): /node[name=thomas-MS-7B51]/bm/cpus/cpu[name=cpu5]
    Free memory available (TESTING): /node[name=thomas-MS-7B51]/bm/mem
      No free hugepages available: /node[name=thomas-MS-7B51]/bm/mem

                             Metrics
          /node[name=thomas-MS-7B51]/bm/cpus/cpu health:99

                            cpu0-cpu5

 idle_time         | 83 % | 90 % | 88 % | 77 % | 89 % | 91 % |      |
 system_time       | 4 %  | 3 %  | 4 %  | 6 %  | 3 %  | 3 %  |      |
 user_time         | 13 % | 6 %  | 8 %  | 17 % | 8 %  | 6 %  |      |
 guest_time        | 0 %  | 0 %  | 0 %  | 0 %  | 0 %  | 0 %  |      |

          /node[name=thomas-MS-7B51]/bm/net/if health:100

                                  | br-146e93d3c94b health:100 |
 rx_packets                       | 0                          | 0
 rx_bytes                         | 0 MB                       | 0
 rx_error                         | 0                          | 0
 rx_drop                          | 0                          | 0
 tx_packets                       | 0                          | 0
```

Figure 1.7: DxTop - Health tab opened

You can see above the figure 1.7 several tabs, these, except the last one, display the metrics of the corresponding subservices. The last tab, Health, displays first the symptoms of the monitored device. On my machine we can see that there are three symptoms of orange severity specific to the memory. Below, we have the list of subservices with their health score and a list of metrics (incomplete) associated. However, this interface does not offer a view of the dependency graph as a whole.

This is why, in addition, a second interface has been implemented to allow a more graphical view, it is called DxWeb. It is a web interface that displays the dependency graph as well as the health scores and symptoms. Here is an example of view that we can observe on the web application:

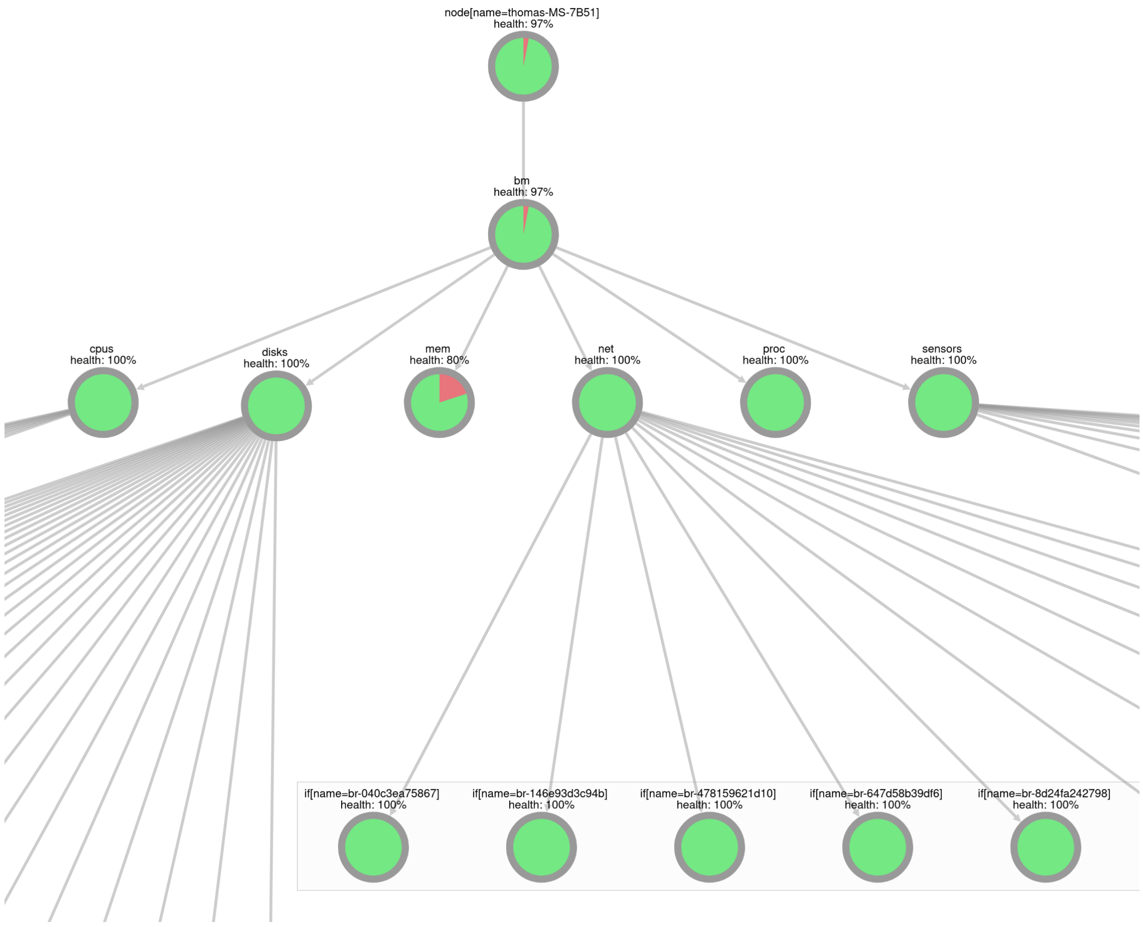


Figure 1.8: DxWeb

We can see from the figure 1.8 that there is a node for each subservice. If it is colored, then the subservice is present on the machine, if it is grayed, then there is no corresponding subservice. The green portion of the nodes indicates the percentage of health of the subservice. It is possible to click on the nodes to know the symptoms of the subservices with a non-maximum health score.

1.3.3 Limitations

The SAIN arhitecture agent that is DxAgent is still in the prototype state and it still has some limitations. One obvious limitation, which is the focus of this thesis, is the lack of active measurements from the network perspective. It should be noted that we say that a measure is active if it creates or modifies traffic on the network. And it is thus, for the moment, impossible to measure the state of health of network links between machines (from a same IBN for example) using DxAgent.

The next step is to implement in the DxAgent inputs related to active network measurements. The inputs which will be recovered are those produced by the OWAMP protocol which we will detail in the following chapter.

Chapter 2

OWAMP Presentation

An idea that has been imagined to improve the DxAgent is to add active network measurements to establish the health of links with other network machines. The measurements that have been chosen are those collected by the OWAMP protocol.

First, we will present what OWAMP is and why it is used. Then we will see the standard architecture of this protocol. Afterwards, the two sub-protocols that compose OWAMP will be explained separately. Finally, we will finish this presentation by showing the set of messages that are exchanged during a normal ping made with OWAMP.

2.1 Overview

OWAMP is a protocol standardized in 2006 in the RFC 4656 [10] by the Internet Engineering Task Force. Its objective is to measure certain unidirectional characteristics such as one way packet delay and loss (defined respectively in RFC 2679 [25] and 2680 [26]). The advantage of these measures compared, for example, to the round-trip delay is that it is not uncommon for congestion to occur in only one direction. Therefore, the information given by these measures is more precise and interesting than those offered by a classic ping. These measures defined by the IETF IP Performance Metrics (IPPM) [23] needed to be computed in a standard way, this is the reason why this protocol was created. And as time sources became more and more present and accessible, it was possible to calculate more accurate measures than those that existed previously.

In addition to its main objective, the team behind this protocol has set other goals:

- The measurement should be hard to detect in order to minimize interference with intermediaries in the middle of the network. The test traffic is indeed difficult to detect because it is a simple UDP stream between ports that have been negotiated beforehand and that does not contain any static content. If we add to this that packets have no fixed size and that we can encrypt the traffic, this protocol is as invisible as possible.
- Another guideline that the team behind OWAMP has given to themselves is to be able to secure the traffic to prevent unauthorized accesses or man-in-the-middle-

attack that would falsify the results.. Therefore, it is possible to authenticate and/or encrypt control and test messages.

- Finally, a last objective was to separate as much as possible the control and testing functionalities.

2.2 The OWAMP Topology

As it has just been stated, OWAMP is in fact composed of two sub-protocols: OWAMP-Control and OWAMP-Tests which have two distinct roles:

The first one (OWAMP-control) aims at managing the tests, which means the initialization, the initiation or the closing of the test session. But it also aims to provide the results when the test session is over. The exchanges of this protocol are done through a TCP connection opened on port 861. The messages of this sub-protocol are only transmitted before and after OWAMP-test session (with the exception of early stop sessions message).

The second one (OWAMP-test) which is layered over UDP is used to send test packets along the Internet path. These packets will then be used as data to measure the quality of the link.

In addition to the separation of the protocol, OWAMP involves several actors whose different roles are as follows:

- Session Sender : the endpoint sending the test stream.
- Session Receiver : the endpoint receiving the test stream.
- Server[1] : the server that manages the OWAMP-test session, it also handles the configurations for each session and can return the results.
- Control Client : the entity initiating the request for a new measurement with the desired configuration.
- Fetch Client : the entity requesting the results from the server.

All these roles can be played by different hosts or a host can have several roles. Therefore, there are many choices in the configuration of the topology and we will see two of them.

The first topology represented in figure 2.1 gives the most genericity and control, each role is played by exactly one host:

[1]When the word "server" is used with a capital letter in this report, we refer to this role in particular.

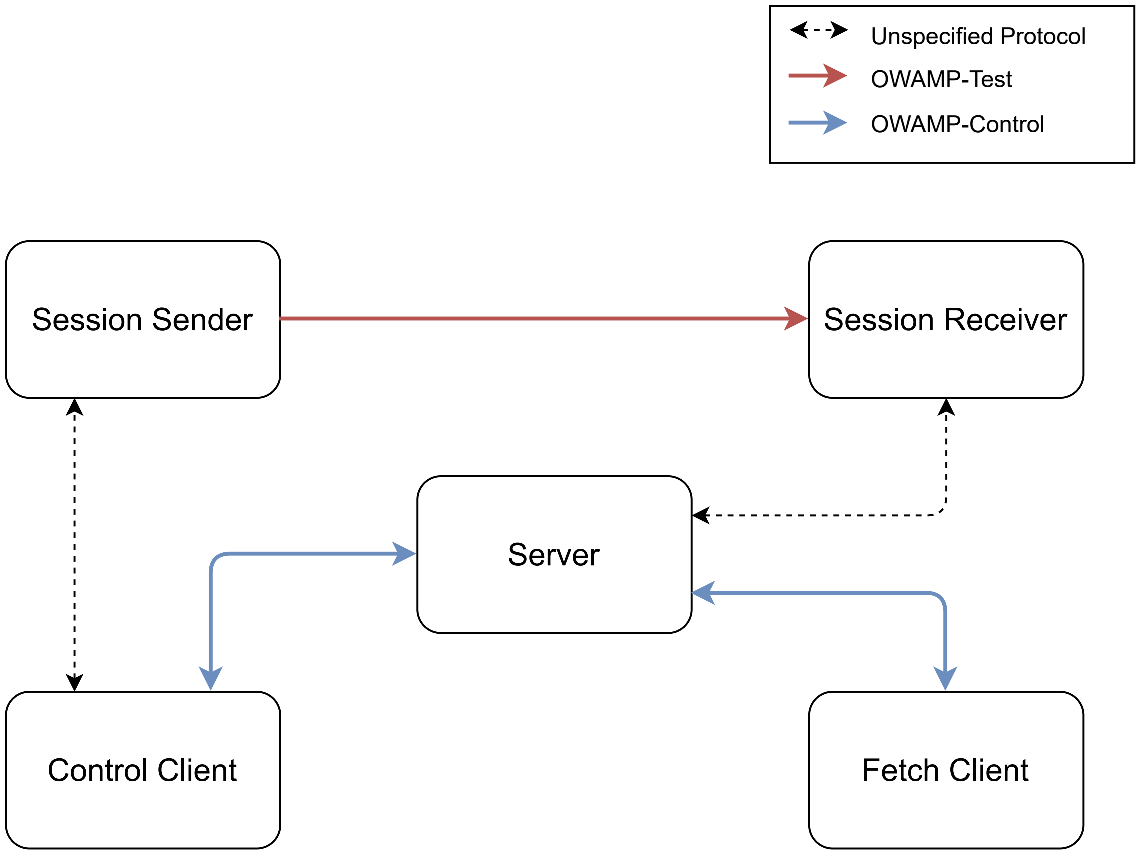


Figure 2.1: Topology 1

The problem with the configuration above is that it is difficult to use in practice. Even if it is possible to imagine the implementation of this kind of topology in a closed network, we have to admit it is far from the simplicity of a classic ping. In addition, some exchanges use unspecified protocols and will therefore use proprietary protocols which requires more elaboration and development. It is much more convenient to have a simple client-server topology : the client host is playing the role of the `Control Client`, `Session Sender` and `Fetch Client` while the server is the `Server` and the `Session Receiver`. A representation of this topology is presented below in figure 2.2.

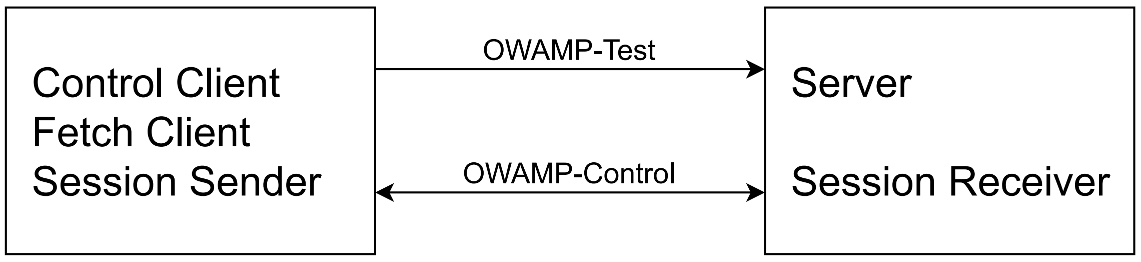


Figure 2.2: Topology 2

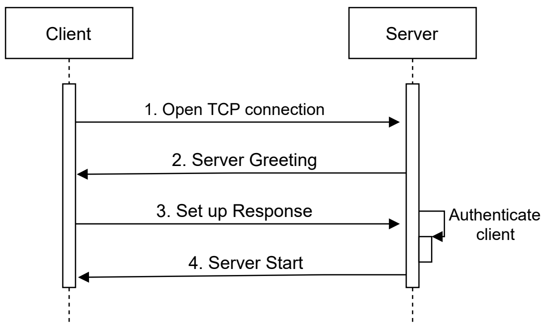
From now on, we will continue to think with this topology in mind because it is easier to grasp, especially since it is the only one that has been implemented so far. And indeed, it is this topology that will be used and integrated in the SAIN project.

2.3 OWAMP-Control

The aim of this section is not to expose all the details of this protocol, only the functional base case will be explained. There are several stopping mechanisms at different stages of the protocol to cancel operations, but also security considerations that will not be discussed. In addition, since the content of the packets can be quite complex, they will not be detailed either. The objective here is to synthesize the functioning of this protocol by listing the different messages that can be sent and their meanings. This reasoning will remain the same for the test protocol that we will later develop.

Before being able to exchange control commands, it is necessary to establish a connection between the client and the server. These first exchanges are also intended to authenticate the client, but also to exchange the information necessary to encrypt the messages according to the server setup. Authentication as well as encryption are not mandatory, the client indicates the mode it wants (open, authentication, encryption) in these first messages and the server will respond favorably if it accepts this mode in its configuration. Note that client authentication and encryption are based on a shared secret in the form of a passphrase and the encryption is done using a key derived from the shared secret.

These exchanges are actually composed of 4 messages which are the following:



- 1. Client open a tcp connection
- 2. Authentication Challenge
- 3. Mode chosen and auth proof
- 4. Setup done.

Figure 2.3: OWAMP connection setup

If the connection setup was successful, then depending on the chosen mode, all the following messages will be authenticated by HMAC and/or encrypted using AES.

From now on, the test session will be done by means of control commands from the client and the server. The client can send 4 different commands which are as follows:

- Request-session : This message is aimed at indicating all the desired parameters for a test stream. This includes a lot of information, the most important being, of course the IP addresses (in version 4 or 6) and the ports chosen for the sender and receiver. Then there are some test parameters like the number of packets to send, the chosen schedule and the DSCP value that can be selected. During the same OWAMP Control connection, it is possible to perform several different tests (i.e several pings), so a session ID is also communicated. In addition, a timeout indicating the time before a packet is considered lost is filled in. Finally, the client has the possibility to indicate when the test should start (this parameter is named start-time).

- Start-session : After sending one or more requests, the client can start the tests using this message. If the requests have been accepted, the server must start the test phase immediately after or at the start-time indicated by the client.
- Stop-session : This message, that should be sent after the test stream is completed, aims to end the test session. The server will answer with an acknowledgement containing all the sequence numbers of the packets that have been skipped[2] if there are any. Skipped packets can happen for example if the start-session is sent too late or a stop-session has been sent prematurely.
- Fetch-session : This message which aims to retrieve the results of the stream test will only be answered if the session has ended successfully. It must indicate a range of packets, these are the packets from which the client wants to get the results of the test (it is possible to request the whole test). If the tests went well, the server must respond with a record per packet containing this information:
  - Send Timestamp
  - Receive Timestamp
  - Send Error Estimate
  - Receive Error Estimate

With the information retrieved using the Fetch-session command, it is very easy to calculate the one way delay (as well as its estimate error[3]).

The server has a dedicated acknowledgment message for each of the client's messages. The only command it can trigger is the Stop-session message to terminate the session prematurely.

2.4 OWAMP-Test

The messages of this protocol are sent through UDP by the ports negotiated during the control phase (Request-session message). The encryption mode used in the test messages will be the one inherited by the controllers (client and server) during the connection-setup. In addition, the encryption and integrity mode remains the same as the control protocol, i.e. an HMAC in each packet and an AES encryption of the packets.

We have seen that the schedule can be decided when requesting the server for a test. Before going further into this new protocol, it is better to specify the nature of this schedule. It is indeed possible to decide, in addition to the time of the first packet sending (start-time), when to send the other packets of the stream. A schedule has a number of time slots where each slot represents a delay to wait before sending a packet. Each slot has a type and a parameter. The first and simplest type is a fixed quantity, in this case the parameter is simply the time to wait. The second type is an exponentially distributed pseudo-random quantity, in this case the parameter is the mean value. When all the slots

[2]In this case, "skipped" means "not sent by the sender"

[3]which is not defined and thus implementation dependant

of the schedule have been exhausted (i.e. a number of packets equal to the number of slots have been sent), one just has to start the schedule again from the beginning, it's a cycle.

To have a better view, let's show an example of a schedule using only fixed values of 2, 1 and 1 (the unit of time is not important to understand the example) with 9 packets to send. The figure 2.4 below illustrate this example.

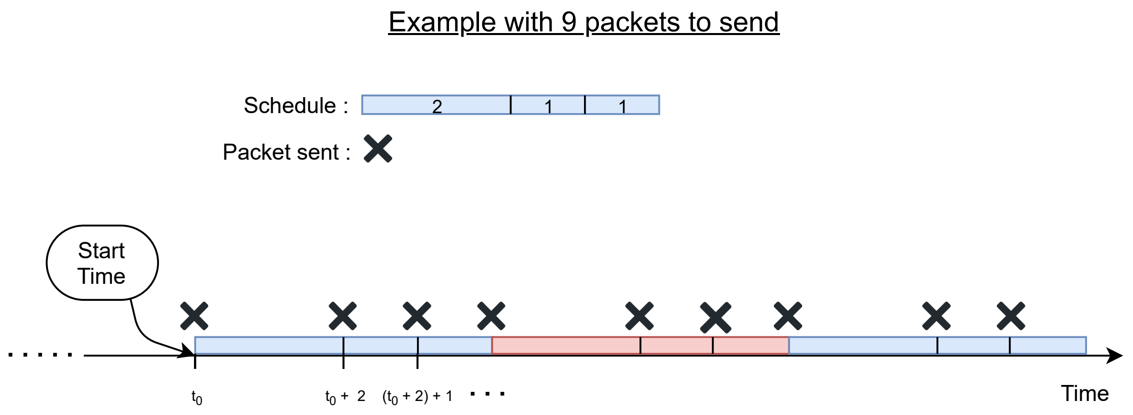


Figure 2.4: Schedule Example

It is important to mention that the algorithm generating the exponential delays has been determined by the RFC in such a way that it always gives the same delay for a given mean, whatever the implementation. In this way, the sender and receiver know exactly when the packets will be sent, which is the very basis of this protocol.

Knowing this, the test protocol is very simple, the sender sends a packet at the time announced in the schedule with an estimation of the error compared to the sending. The receiver knows when to receive it and will timestamp and save it as it is. It should be noted that there are several cases where the packet will be dropped, especially if it is sent at least one timeout too early or too late. And it is indeed possible to send it "too early" from the receiver's point of view if the clocks of the two entities have a synchronization difference greater than the one-way delay. In this case, the data is saved as it is and it is the role of the Fetch-Client to use this data as it wants. This protocol is not intended to calculate the statistics itself but it is the Fetch-Client that will compute what it needs (one way maximum/minimum/median delay, reordering, packet loss, etc) from the raw data.

2.5 Complete Example

To conclude this chapter and to have a better overview of this protocol, a complete example will be shown. To do so, all the messages exchanged by the two OWAMP sub-protocols during a one-way ping will be gathered and put in order to create an example of the complete process. For the sake of comprehension, we put ourselves in the case of a ping without error with a basic client-server topology. The figure 2.5 below is a diagram representing the different actions of an OWAMP ping example.

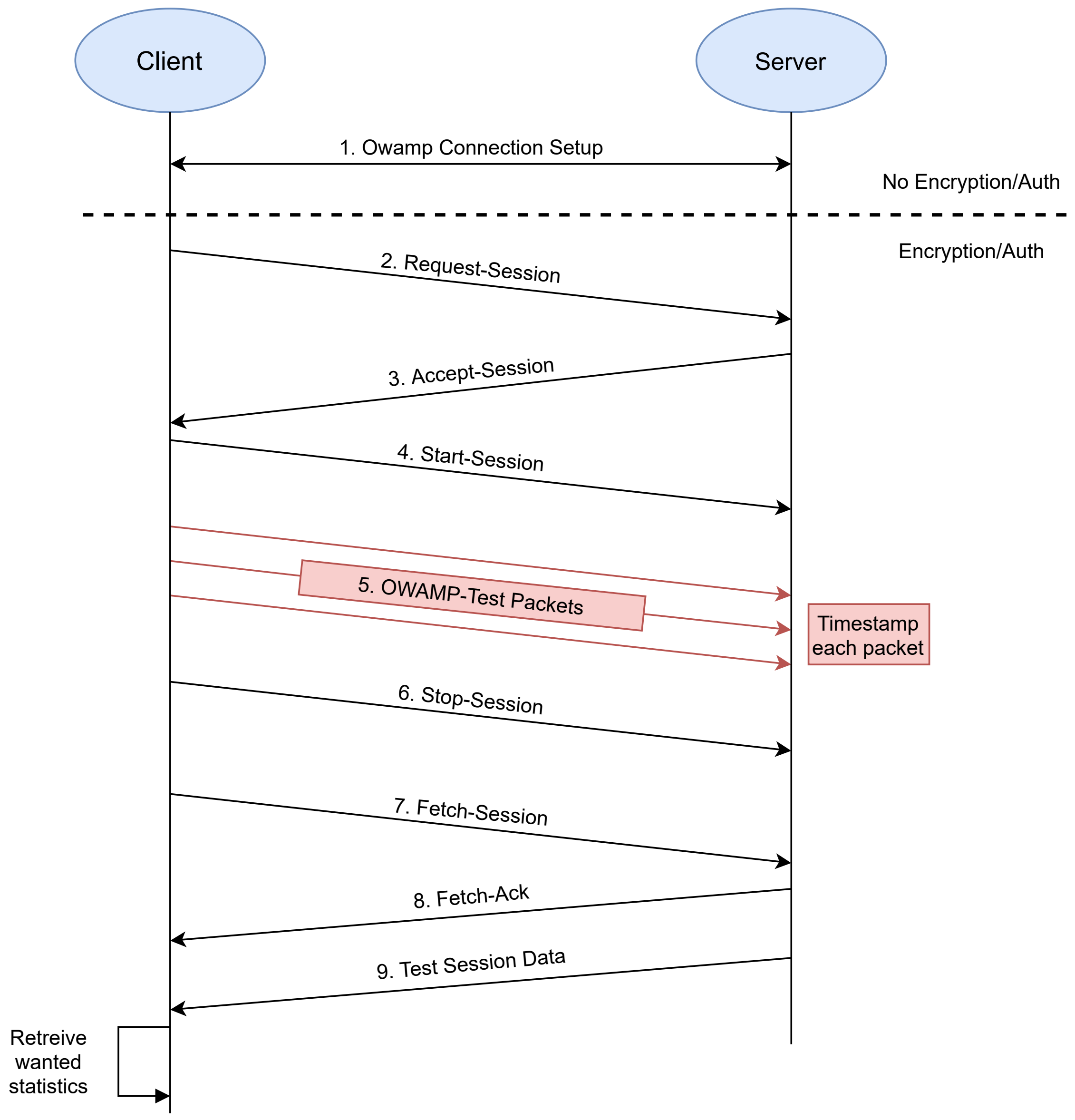


Figure 2.5: Complete Example

In order to differentiate the messages coming from OWAMP-test on the figure 2.5, they are represented in red while those from OWAMP-Control are in black. The sequence of actions is as follows:

1. The client and server setup the connection and decide on the encryption and authentication modes (see fig 2.3 for more details)
2. The client requests a ping with the parameters it wants for the test
3. The server accepts with a simple Ack
4. The client sends a message to start the test session
5. The test packets are sent at the time specified by the schedule decided in point 2. Moreover each packet is timestamped by the server.

6. The client sends a message that the test is finished and tells the server which packets were actually sent.

7. The client ask for the test results.

8. The server replies with an Ack (specifying the number of packets received and the number of potential holes in the sending process)

9. The test data is sent to the client, including the receive timestamp of each test packet received by the server. The client can then deduce the statistics it wants by comparing the send timestamp and the receive timestamp like one-way delay, loss, reordering, etc.

Chapter 3

OWAMP Interface and Integration in Dx-agent

This chapter is dedicated to the implementation work that has been done. First, we will see the implementation that has been used for OWAMP. Then we will see the interface of this implementation that has been developed in the framework of this project. Finally we will see how the new network metrics were added to the DxAgent

3.1 Existing Implementation

After integrating the concepts of the OWAMP protocol, a usable implementation had to be found to be able to integrate it into the SAIN project. Only one open-source implementation [20] can be found on the net, the one developed by the authors of the RFC of these protocols. This implementation, written in C, is maintained and distributed within the perfSonar [29] project (a network measurement toolkit managed by the internet2 community [27]). The problem with this implementation is that it is not directly usable in the SAIN project which is developed in python. That's why I made an interface in python to be able to launch and use the topology of the implementation. But before presenting this interface, a brief presentation of the internet2 implementation is necessary.

3.1.1 The topology

The topology chosen by internet2 is the one presented in figure 2.2. It is therefore the standard client-server topology that we are used to. Indeed, the interest of this topology is to make the use of this tool as simple as possible, even if it means less flexibility. While the protocols allow many possibilities, this implementation can only measure the link between the client and the server. The use of this implementation is then very close to the traditional ping. To use it, a host has to run the server, it is then able to receive requests from clients that want to get the network statistics between them and the server.

The project that can be found on github [8] is particularly complex, once the code is compiled, it is 10 programs that are built (the server, the client, statistics tools, etc).

But we are going to focus on only two of them: owampd and owping. Owampd is the server program while owping is the client that makes the request to the server. These programs are highly configurable and we will see some of their possibilities. But first, let's take a look at these programs' architecture. As can be seen in the figure 3.1, the owampd program has been developed as a classic accept/fork daemon. The master daemon listens for new connections and manages the resources of the child processes. When a connection arrives, the master daemon forks a child, which will take care of the control exchanges of the OWAMP-Control protocol. Then, the owampd child process and the owping process will fork a child process (the test endpoints) which will in turn exchange the test packets of OWAMP-test protocol.

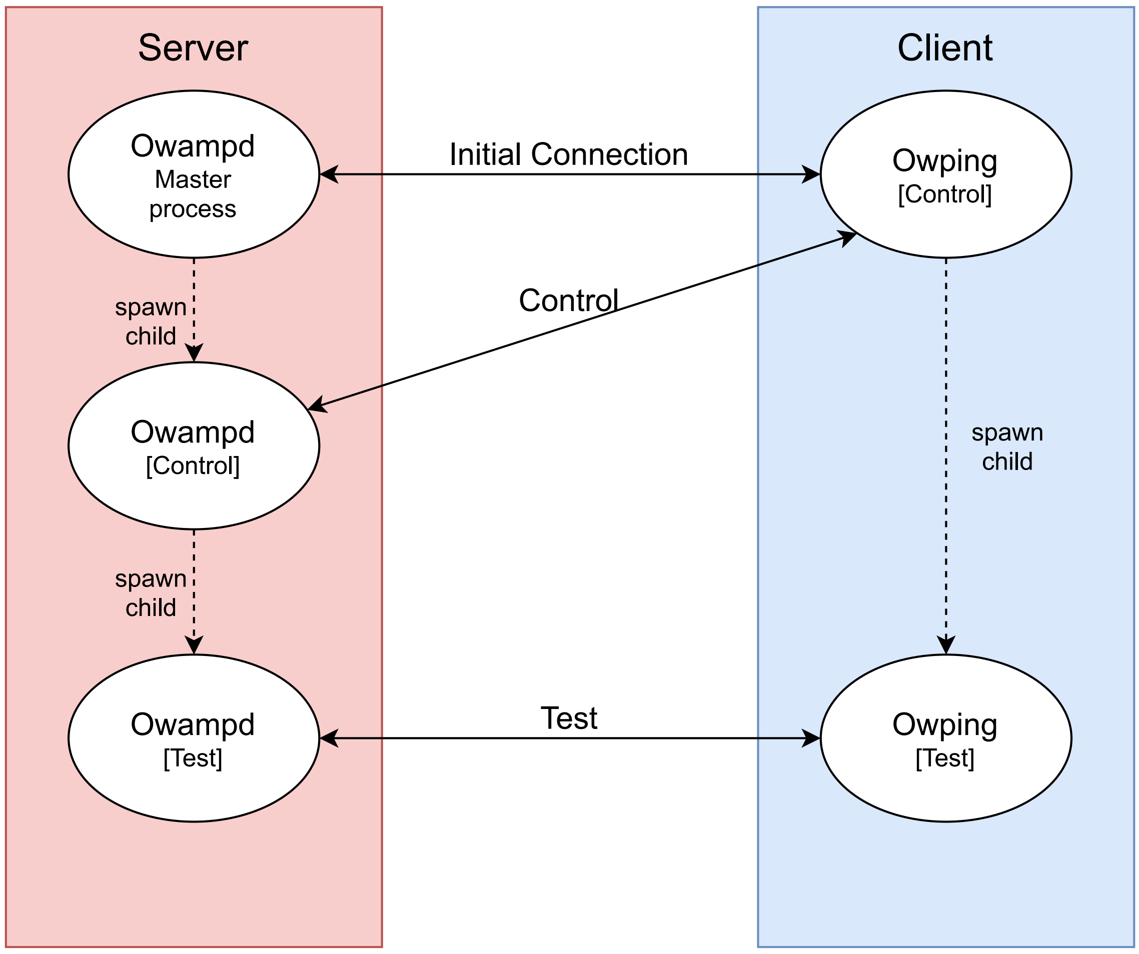


Figure 3.1: Owamp implementation architecture

3.1.1.1 The server: owampd

Before starting the server, it must be configured, which is done by modifying several files in the installation folder. We are not going to present each of the options that can be configured but we will only expose the most important ones. First of all, this implementation does not forget a key point of the OWAMP protocol: authentication and encryption. It is indeed possible, by means of policies, to create limit classes. These classes, that supports inheritance, can have several attributes, such as the authentication mode, bandwidth, maximum disk space, etc. Once a limit class has been established, it must be assigned to either users or network subnets. For specific users, a passphrase must be registered and only a client (owping) having the passphrase will be able to authenticate and therefore be assigned to this class. In this way you can for example, create a class for administrators assigned to user Joe:

```
limit root with \
            bandwidth=900m, \
            disk=2g, \
            allow_open_mode=off

assign user joe root
```

Or a class rejecting any access assigned to the "bad" subnet 192.168.1.0/24:

```
limit jail with \
         parent=root, \
         bandwidth=1, \ # bandwidth = 1 is similar to rejection
         disk=1, \
         allow_open_mode=off

assign net 192.168.1.0/24 jail
```

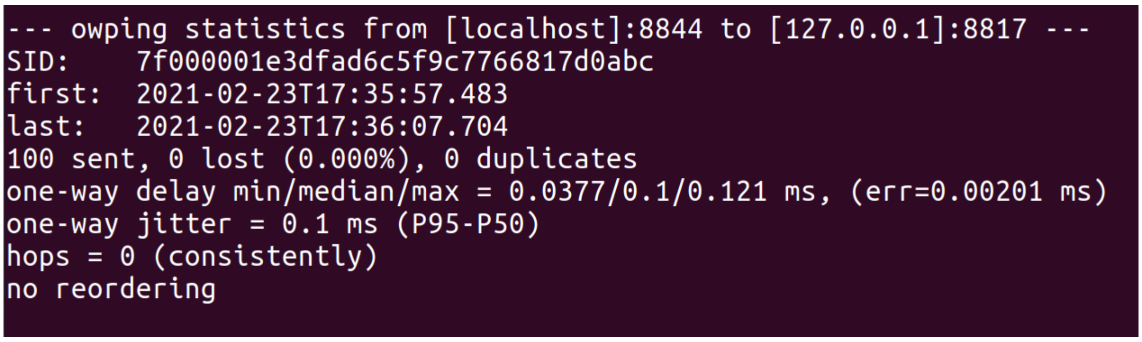
This creation of limit classes allows a high degree of flexibility which can be very useful.

In addition to the attributes that can be attached to the limit classes, other important server configurations are possible and some even mandatory with every new installation on a host. For example: the user running the program, the ports used for control and testing, timeouts, delays, minimal authentication mode, etc.

3.1.1.2 The client: owping

Among all the options that can be modified to launch the owping program, the most important are those that influence the type of tests that will be carried out. For example, it is possible to choose the number of packets for a test session, the DHSCP value, the stream test schedule, the packet size, the timeout, etc. Of course, the user can also be specified to be assigned to a limit class of the target server.

Once all options have been chosen, the test will run and return a series of statistics in the form of the desired output. We can see an example on figure 3.2 of an output that I retrieved by doing a test on the loopback address.



```
--- owping statistics from [localhost]:8844 to [127.0.0.1]:8817 ---
SID:    7f000001e3dfad6c5f9c7766817d0abc
first:  2021-02-23T17:35:57.483
last:   2021-02-23T17:36:07.704
100 sent, 0 lost (0.000%), 0 duplicates
one-way delay min/median/max = 0.0377/0.1/0.121 ms, (err=0.00201 ms)
one-way jitter = 0.1 ms (P95-P50)
hops = 0 (consistently)
no reordering
```

Figure 3.2: Owping output

It is therefore possible to recover, among other things, the number of packets sent, lost, or duplicated. But above all the one-way delay (minimum, median and maximum) obtained as well as an estimate of its error. Moreover, the number of hops can be retrieved thanks to the TTL (set to 255 at the time of sending) as well as the information of a possible reordering.

3.2 Development of an API

3.2.1 Presentation

Instead of just using the programs of the internet2 implementation directly in the SAIN project, it was more interesting to create a python API that could handle most of the uses of OWAMP. This library would therefore aim to be flexible enough as the OWAMP implementation is, and that anyone wishing to use this protocol in a python project could quickly use it. For this reason my work is available on github at : `https://github.com/tcarlisi/owamp_wrapper`

The interest of such an API would be, besides making this protocol more available, to be easy to integrate into the SAIN project. However, this kind of realization requires to take up several challenges which are the following:

- This API must be highly configurable, it would be unfortunate to lose all the flexibility offered by the protocol and its implementation.
- It also requires thinking about a set of functions that is both understandable and, above all, sufficient. This API should not undergo any modification in order to be used and should therefore handle the most possible use cases.
- Good exception handling is also very important for anyone wishing to use the API. Therefore, a set of custom exceptions must be set up to make it easier to understand potential problems. The fact that the API handles independent programs makes error handling particularly complicated.

The realized project is thus a python program made up of several classes whose one is on public display: the API itself. Even if the heart of this work is an interface, it is not wrong to call it a program because a main class is provided in the project to present a use case. Before explaining the interest of these classes and how they work, a class diagram is available below on figure 3.3 to simplify the understanding of the different class interactions.

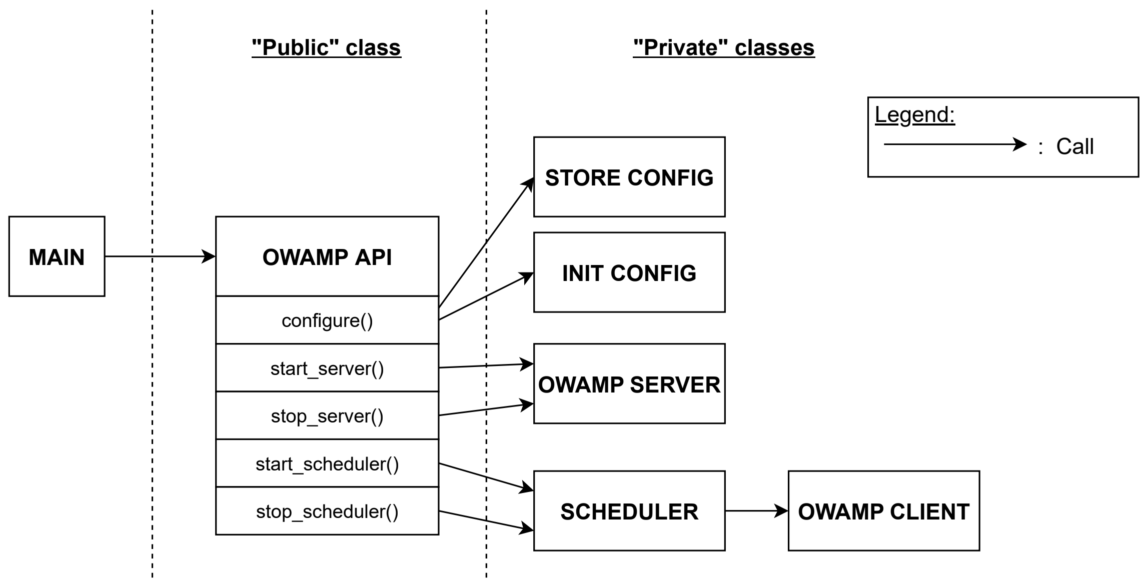


Figure 3.3: Class Diagram of the OWAMP API project

In order to make this diagram as clear as possible, only the API functions are displayed. Moreover, to better understand the classes used by the API, the arrows starting from the API class start from the functions that specifically use the targeted private classes. In reality, the notion of public and private here has nothing to do with the concepts of object-oriented programming but simply indicates that a user of this project does not need to use anything other than the API class.

To summarize the functionalities of this API, it is possible to configure an OWAMP server, to launch it (and close it) but also to send pings (on a recurring basis if needed) and to retrieve statistics.

3.2.2 Explanation

We will now proceed to an explanation of the different components of the program starting from the functions of the interface.

Configure()

As we can see in the class diagram, the first function of the api, `configure()`, uses two classes (`init_config` and `store_config`) and this makes sense because this function has two roles. The first one is to retrieve the configuration data of the API user while the second one is to reintroduce these options in the code and in the OWAMP configuration files.

Indeed, the first step is essential if we want to create an interface that is totally detached from the intial C implementation. For this reason, the interface is completely usable without having to understand anything about how to configure the OWAMP implementation, which is a good thing since the original configuration is not practical to use. Indeed, just to be able to launch the server for example, it is necessary to modify several files in several different folders but also to launch another program to create a password file. This implementation lacks accessibility and one of the objectives of this api is to remedy this. Of course, the user configuration retrieval goes through a series of input checks before being stored in a data structure used by several other classes.

Then, the next step is to modify the implementation files to match the needs of the interface user. This step therefore involves, among other things, rewriting OWAMP files to modify some directives.

start_server()

Once the configuration data has been retrieved and the OWAMP implementation folder has been configured, it is now possible to launch the server. This consists in launching the owampd program with the right parameters. The standard python subprocesses [6] library was then very useful because it allows to run a program. Before continuing, it may be interesting to quickly present the use of this library (because it is used several times in this project) by means of lines retrieved from my code:

```
process = Popen(shlex.split(cmd), stdout=PIPE, stderr=PIPE)
stdout, stderr = process.communicate()
exit_code = process.wait()
```

The first line is intended to create a process from the shell-like command line `cmd`[1] but also to indicate that we want to be able to retrieve the standard output and error output. The second line asks to communicate with the process in order to read the standard outputs. And finally the last line aims to wait for the end of the process execution and returns the exit code of the program.

We might think that waiting for the server to finish executing is a very bad idea because it would block the program entirely, but it is not. In fact, owampd is a bit particular because it will itself launch another process in background (which will be the server), then it will write the pid of this server in a file and will close. This is done very quickly and it is therefore interesting to retrieve, in addition to the outputs, the exit code in order to be sure of the right execution of the server starting. Finally, closing the server is quite simple because we know the process pid and we can kill it whenever we want (with a call of the `stop_server` function).

[1]In this case, this is the command to start owampd with the right parameters

start_scheduler()

Another possibility of this API is to be able to ping a list of hosts on a recurring basis. For example, a user can configure a list of four host addresses and a frequency of 1 ping every 20 seconds. And so a ping will be sent to all 4 addresses at the same time every 20 seconds. This kind of use can be very useful when you want to create a monitoring program like the SAIN project. Of course, it is also possible to ping a single host on a one-off basis.

To implement this type of operation, the API uses a job scheduler called apscheduler [1]. This highly configurable library allows to schedule a job at a certain time of the day or at a certain frequency. In our case, it is the second use that interests us. It could be interesting to present this library at the same time as showing the different choices that have been made for its use.

The apscheduler has 4 components:

- schedulers
- job stores
- triggers
- executors

The scheduler is the interface with which the application developer will interact. It gathers all the components of apscheduler and there are several types of them. The main ones are the `BlockingScheduler` and the `Backgroundschedduler`. The first one can be useful if it is the only thing that has to run in the program, but this is not our case. So the second type will be used, this one is not blocking because it runs in parallel with the calling program.

The job store is what stores jobs as its name implies. Again, there are several types of job stores, but the one chosen is the default one. Indeed, there are job stores that allow one to keep the status of the job even after the scheduler has closed or crashed, but this was not necessary for the use we have in this project.

Triggers are what contain the scheduling logic. In our case, the chosen trigger is simply a trigger that is activated at a fixed interval (typically seconds).

The executors are the ones who take care of running the jobs. They typically submit a job that has been triggered to a thread or process pool. Once the job is completed, the executor can notify the scheduler whether or not the job was successful (and can even give the return value of the job). It is possible to choose between an executor based on thread pool or process pool. Since our job is to simply run a ping program and wait for the end of its execution to recover the return value, a thread pool were sufficient. Process pool could have been useful to take advantage of several CPU cores in the case of intensive use of CPU operation but here the job spends most of the time waiting.

As explained, it is possible to be notified when a job has been completed and even to know if it worked. To take advantage of this feature, the API function `start_scheduler` takes as arguments a callback function when a ping is successful and another one when an error occurred. This leaves a certain amount of leeway to the users. For example, the success callback can be used to store or display the ping results. While the failure callback can be used to raise a custom error or a warning for example.

Now it is time to discuss about the job itself which consists in pinging (using OWAMP ping style of course) a specified address. There is not much to add about the way the ping is launched because it is the same as for the server. So, we launch the owping program with the right parameters and we get the standard outputs and the return value of the program. However, a notable difference with the server program is that the owping program only stops when the ping is finished, which can take several seconds. And since we want to recover the return value, we have to wait for the end of the execution. But this is not a problem because the waiting is done inside a job that happens to be a thread. Therefore, even if the program execution is blocking, it is only this particular thread that will be blocked and thus it does not cause any waiting to the application. Then, once the execution of the program is finished, the standard output of the process owping is parsed to fill a data structure available with the API. This structure is then returned as the return value of the job and can therefore be used directly in the callback function that can be customized by the user.

3.2.3 Installation and usage

The installation and the use of the API are explained in the github. Moreover an installation script is provided to setup the directories, install the different components and perform the necessary steps for authentication. I took a lot of time to understand how to launch OWAMP and play with the parameters (especially for authentication), so I wanted to take special care of the ease of use for this python interface.

3.3 Integration in the DxAgent

We now come to the central subject of our work that consists in extending the field of vision of DxAgent. The git repository containing all the commits done for the integration is the following: `https://gitlab.uliege.be/Thomas.Carlisi/dxagent_owamp`. Indeed, this project already analyses a myriad of inputs but it is not yet exhaustive. Therefore, it was proposed to me to improve its ability to detect anomalies by adding active measurements from the network point of view. There are of course several types of protocols allowing to realize active measurements but the choice of OWAMP was a natural choice. We will analyze in the use case (presented in chapter 4) the differences that this one can have with other active measurement protocols. In the meantime, we can still highlight its main advantage which is, as we recall, the measurement of one-way metrics. Indeed, in a context of Intent Based Networking (or SDN), having the possibility to discriminate the quality of a connection in one direction or another can have a certain interest. Indeed, some services might require a more severe minimum acceptable delay in one direction of the link and not in the other. Therefore, the DxAgent can use different health rules for the two directions of the same link.

Before going into the implementation details of the OWAMP integration in the Dx-Agent, it would be wise to illustrate the use of their combination. As explained, the implementation of OWAMP requires the use of a client and a server. Therefore, these two components had to be integrated. Therefore, the DxAgent integrates both a server and the client functionalities, i.e. the sending of measurement requests. The figure 3.4 below presents an example of different possible uses with this combination of OWAMP and DxAgent

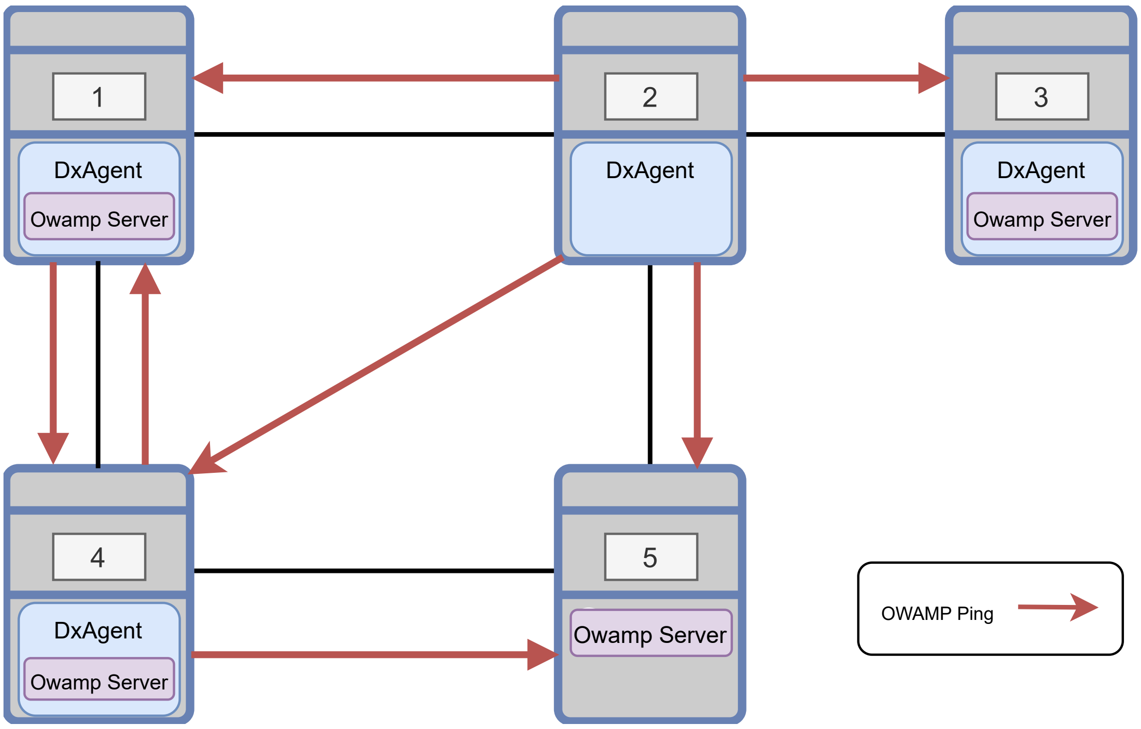


Figure 3.4: OWAMP and DxAgent Combination

As it is possible to see on the figure 3.4, the use of the server as well as the sending of request is completely optional. It can be seen on machines 2 and 5, it is entirely possible to have a server running without DxAgent or a DxAgent not integrating an OWAMP server. This freedom of use leaves a network operator free to choose the way in which the DxAgents will communicate with each other. The arrows in red on the figure show simply which machines decide to ping which other machine(s), it is not about network communication. The most basic use case is represented by machines 1 and 4 running DxAgent with the OWAMP server. They ping each other and are therefore both able to retrieve information about their link characteristics (delay, loss, etc) at any time. Moreover, we remind that the interface that has been developed in the frame of this work allows to send a ping to several servers at the same time from the same machine (as machine 2 does). For these reasons, it is possible to imagine all sorts of topologies and uses.

3.3.1 The DxAgent Implementation

As already explained, the DxAgent project was realized by a third party at the University of Liege. A first step that I had to realize was to understand the scope of this project. I had to understand the whole project and all the code by myself before I could imagine any modification. This crucial step proved to be quite difficult, it takes time to get into a code of this scale with very few initial clues. In the same way that I had to understand this project to integrate OWAMP, it is important to briefly present this implementation before presenting my additions. Of course, only the essential elements I worked on will be presented (some classes and features not really relevant of the DxAgent will not be presented).

The easiest way to understand and present a project of this size is to present the different classes and the way they fit together. A diagram of the different classes that will be presented is available in the following figure 3.5.

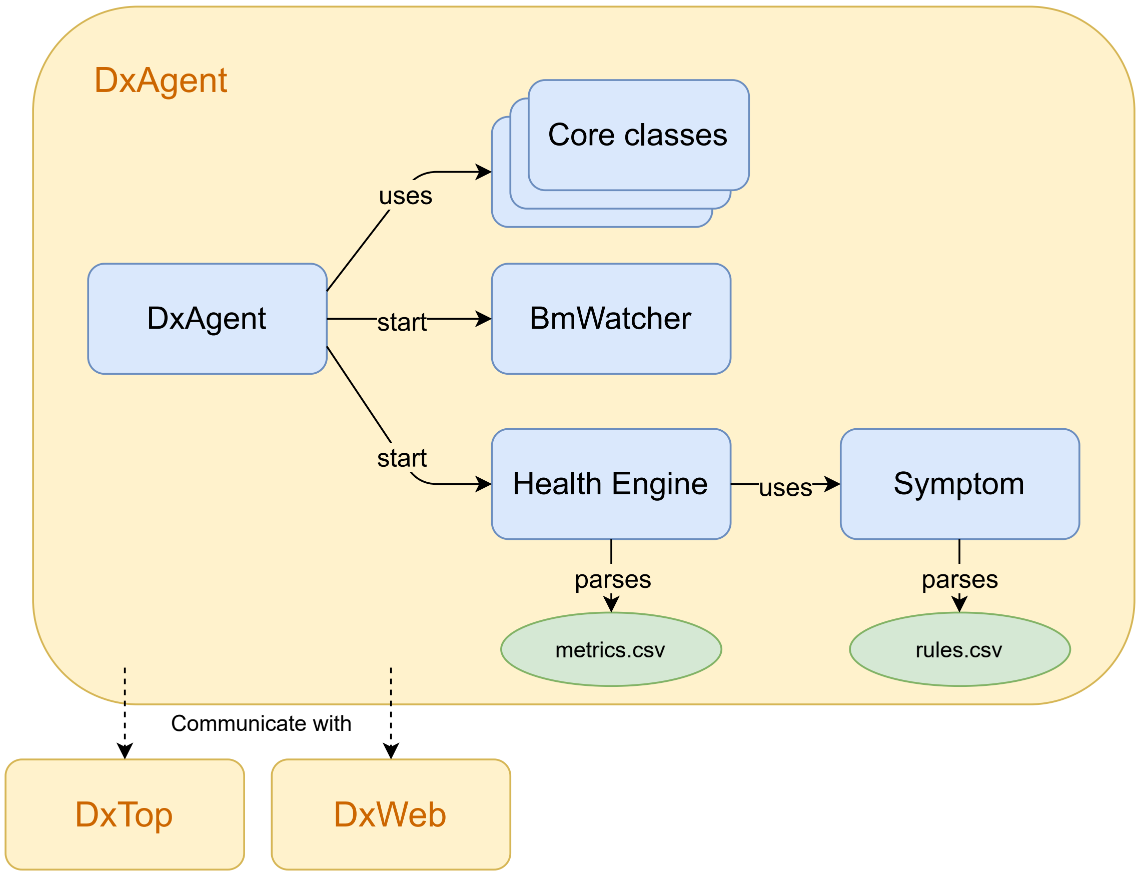


Figure 3.5: DxAgent class diagram

Without going into the implementation details, it may be interesting to briefly explain the interest of these classes.

- **DxAgent:** This is the main class that will contain all the actions (comparable to a main). Mostly, it is about launching the input analysis (by starting the Bm Watcher) and then initiating the transformation into metric and the creation of symptoms (by starting the Health Engine). Once the sequence of actions `input` → `metric` → `rule` that we have already presented is finished, they are scheduled again after a period of 3 seconds and so on. By the way, it should be noted that if the DxAgent is configured to launch the OWAMP server, it will be done in this class before the scheduled process is started.
- **Core classes:** These are the classes that allow the DxAgent to be launched as a daemon as well as the definition of certain data structures. The most important

one is the ringbuffer. Indeed all inputs and their metrics equivalent are stored in a dictionary of ringbuffer dictionary. The first dictionary refering to the type of input to retrieve and the second key refers to the input. The value of this embedded dictionaries is the ringbuffer of the data collected for the input. The purpose of the ringbuffer is to keep in memory the old values of the inputs/metrics to be able to analyze them through time. Moreover, the core classes include a class allowing to interact with the inputs/outputs of the program. This class allowed me, for example, to link in a single configuration file the options of DxAgent and OWAMP.

- **BmWatcher:** This is the class that allows one to retrieve the baremetal inputs (i.e. the inputs of the physical machines). The `input()` function of this class, which is called every 3 seconds, retrieves data from interfaces, CPUs, memory, etc. It is mostly about reading files (contained in /proc, /sys, etc.) or retrieving data by means of programs (interfaced by python libraries) like `ethtool` [31] for example.
- **HealthEngine:** The interest of this class is twofold, first of all to select and transforms the inputs to be normalized in metric. Note that all inputs are not necessarily transformed into metrics, some inputs are just interesting to recover the name of an interface, the time of the data recovery, etc. In reality, the set of metrics, retrieved from metrics.csv is almost four times smaller than the set of inputs. Secondly, the purpose of this class is to create the dependency graph or to update it. It is then a matter of determining which subservices are active but also of determining potential changes. For example, it is possible that an interface is deactivated after the launch of DxAgent, so the update function must remove the node linked to the subservice in question. It is also this class that initiates the analysis of the rules defined in a `health.csv` (parsed by the `Symptom` class) to determine a potential change in health score.
- **DxTop and DxWeb:** These two interfaces are programs that can communicate with an instance of the DxAgent. I had to modify them slightly in order to introduce the input, metrics and health data specific to OWAMP. Moreover I modified DxTop so that it can better manage the display when the number of CPU is not an exponent of 2 (which was the case of my personal computer).

3.3.2 Integrate OWAMP as Input/Metric/Rule

Now that the structure of the program is known, it is easier to show the additions that have been made. To do this, we will proceed by separating the explanation into three parts, which are Input, Metric and Rule.

3.3.2.1 Input

The first of the three actions that occurs in the DxAgent consists, as we recall, in recovering the inputs of the machine under study, which will be used to evaluate the state of health of a subservice. In our case, the subservice that we want to integrate is the communication between the machine that is being monitored and other machines (of a same or different subnet). This subservice is interesting because it is not rare for an IBN (or SDN) to have network requirements within its network. The objective of the study of this subservice is to recover network link information such as reachability, delay, packet loss, etc. In our case, we retrieve all this data by means of OWAMP pings, we just have to find a way to send them to the DxAgent.

Handle the schedulers

We should remind that the input recovery is scheduled at a frequency fixed by the DxAgent. Once the cycle of the three actions of the DxAgent is completed (input/metric/rule), it is rescheduled at a fixed number of seconds later (3 by default). Moreover, as we have seen in the API, OWAMP pings are also scheduled by a fixed frequency scheduler (which can be modified in config file). However, this scheduler will take place at the set frequency, no matter how long it takes to complete the action. The difference is clearer when we look at the figure 3.6 that will be presented below. We clearly see that the time of next input retrieval depends of the time of the current retrieval.

The fact that we have two different schedulers in different processes creates some complexity but is also essential. Indeed, it leaves a lot of flexibility on when and how often the pings will be executed. Furthermore, it would be unreasonable to think that the pings are performed in the same thread as the DxAgent. Since pings take a long time to complete, this would considerably increase the input retrieval. For this reason, the pings are executed in a thread pool managed by the scheduler. The figure 3.6 below shows an example of these two schedulers if they are set to 3 seconds for the DxAgent and 7 seconds for the OWAMP pings. In the example, the number of different addresses to ping is set to 3, so there will be three threads in the pool.

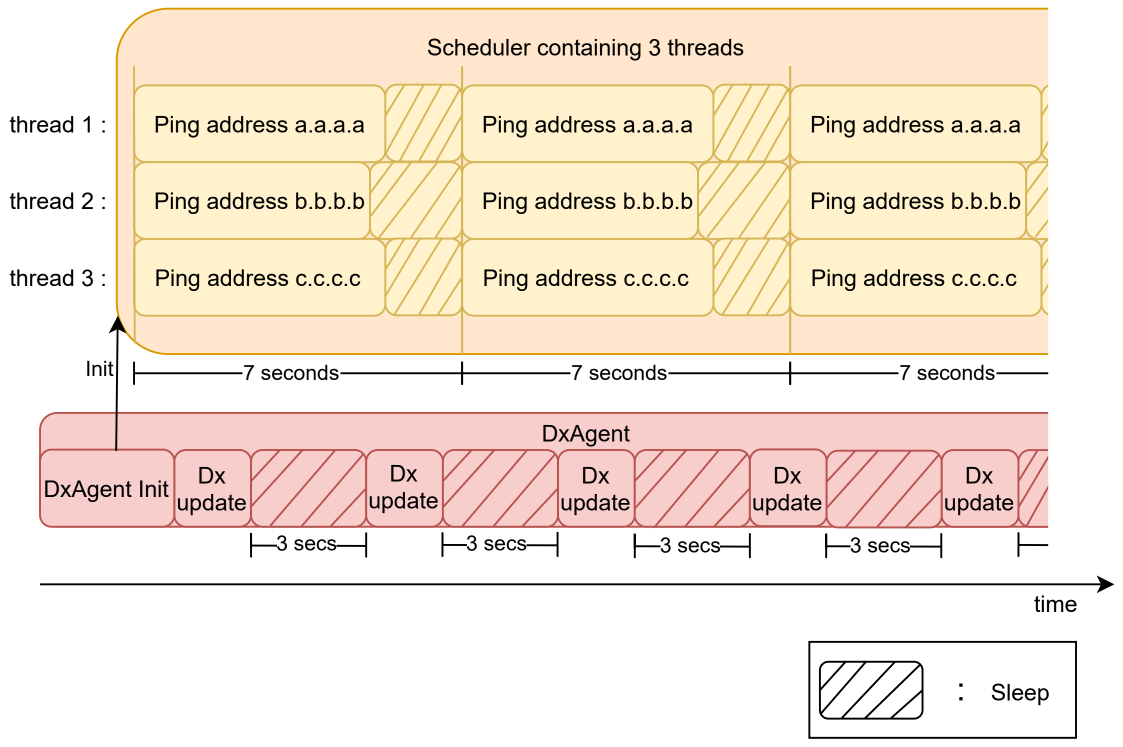


Figure 3.6: Schedulers organization

On the figure 3.6, is represented in red the course of DxAgent course of actions over time. We notice that it starts with an initialization phase in which the OWAMP ping scheduler is started. Then, once the operations allowing to update the state of health of the machine is finished, the DxAgent passes by a phase of waiting of 3 seconds then starts again an update and so on. What is represented in yellow corresponds to the OWAMP scheduler which starts every 7 seconds a new ping in each of its 3 threads, whatever the time taken by the previous ones. Note that it is preferable to choose a frequency that allows a complete ping to be executed, otherwise the ping will be delayed and the next pings will become desynchronized. If there is desynchronization of the pings, this does not pose any problems to the programs, but it is preferable for the user that the DxAgent recovers the information from all the pings during the same input update. We can see that these two programs are completely independent, so we have to find a way to communicate between them.

Possible pings scenarios

Before explaining how the DxAgent retrieves information from OWAMP, it is important to present all the scenarios that can happen with the pings scheduler.

- It is possible that the DxAgent does not require any ping, in this case, it is obviously not necessary to start a scheduler or to retrieve input specific to OWAMP
- The DxAgent can target one or several addresses to ping. It is then necessary to differentiate and recover the information specific to each address.
- It is possible, and even almost certain that the first input retrieval takes place before the end of the first ping, in this case, the display in the DxTop as well as the creation of links requires a certain vigilance.
- An address may be accessible for a while but the ping stops in the middle because, for example, the OWAMP server is stopped. The OWAMP program will not detect this as an anomaly, so it is up to the DxAgent to manage the recovered values and to handle them as best it can (notably by ignoring them if they have not had time to send and receive a single packet).
- If an address is no longer available, one must indicate the user and also manage the various inputs which no longer make sense (delay, los, etc.).
- An address which was no longer accessible becomes accessible again. This case does not pose a problem given the way in which the OWAMP input retrieval has been implemented.

It was necessary to think about all these possibilities during the whole integration, as well for the input retrieval, as for the creation of the dependency graph or for the display on DxTop.

Input retrieval by file reading/writing

The input retrieval itself is done by means of a file that will contain the ping OWAMP information. The callback function, contained in the DxAgent, sent to the OWAMP API scheduler will ask the scheduler to start writing the output to a file (one file per different addresses to ping). In this way, when an input retrieval is scheduled by the DxAgent, it is enough to read this file. The choice of read/write files to associate the DxAgent and OWAMP was natural. First of all, thanks to the way it was implemented, it is not necessary to synchronize the two programs in any way. In addition, the DxAgent gets almost all of the input by means of file reading. Using any other method, such as shared memory or message passing, would not only complicate the procedure by adding synchronization concerns, but would be inconsistent with the way DxAgent works. In this way, the process is simple and does not require any modification of the OWAMP API, which was developed independently from DxAgent.

In order to better understand how the two schedulers are organized and how the data retrieval takes place, a diagram is shown below in figure 3.7

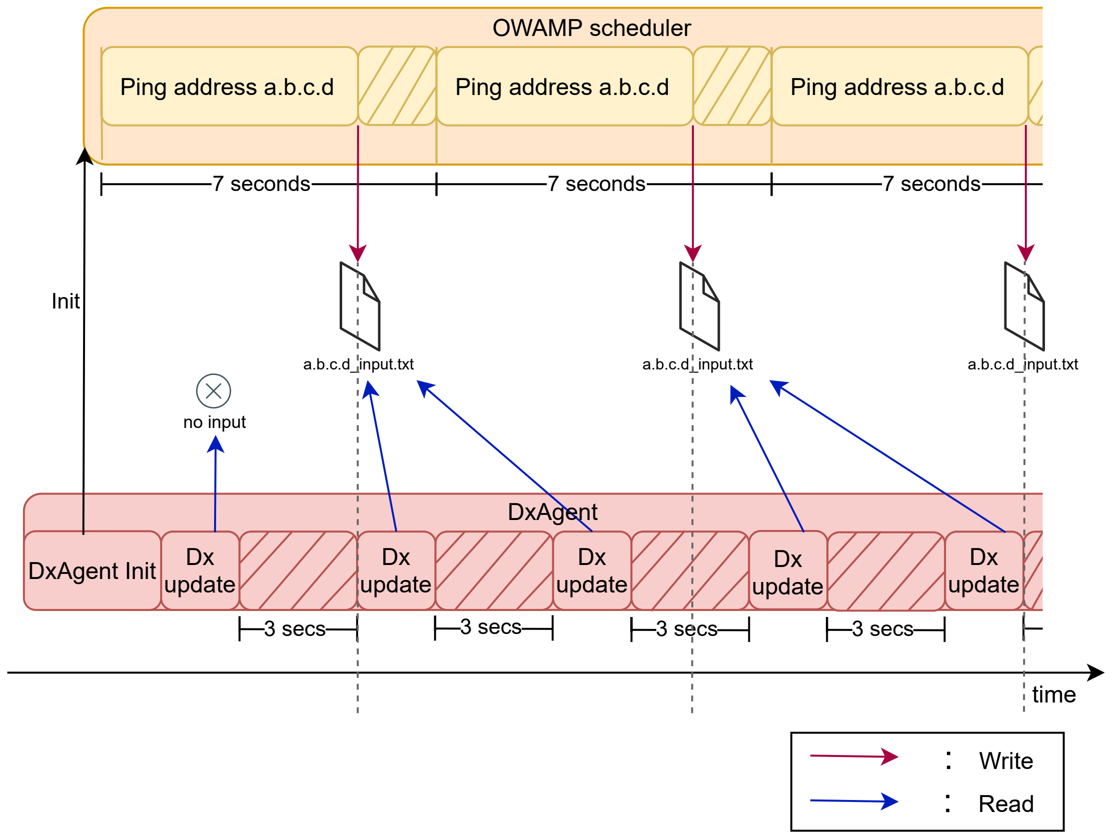


Figure 3.7: Input Retrieval using file reading/writing

The example shown in figure 3.7 represents only one thread (and thus only one address to ping) but the reasoning remains the same if one increases the number of threads. The file writing takes place as soon as the ping is finished. While the DxAgent tries to read this file at each update. If one exists, the values will be updated if it is a new ping. If there are several updates before a new file is written, as in the example, the values will remain the same in the DxAgent until the next write. It is noted that the creation of the ping-related subservice is only created the first time the file exists. This technique allows a high flexibility in the choice of frequencies of the two schedulers. One can also imagine to perform 3 pings between two updates of the DxAgent (only one will be read in this case) or on the contrary only one ping per day. To summarize this operation, when a new value is ready, this is the one that will be read at the next DxAgent update.

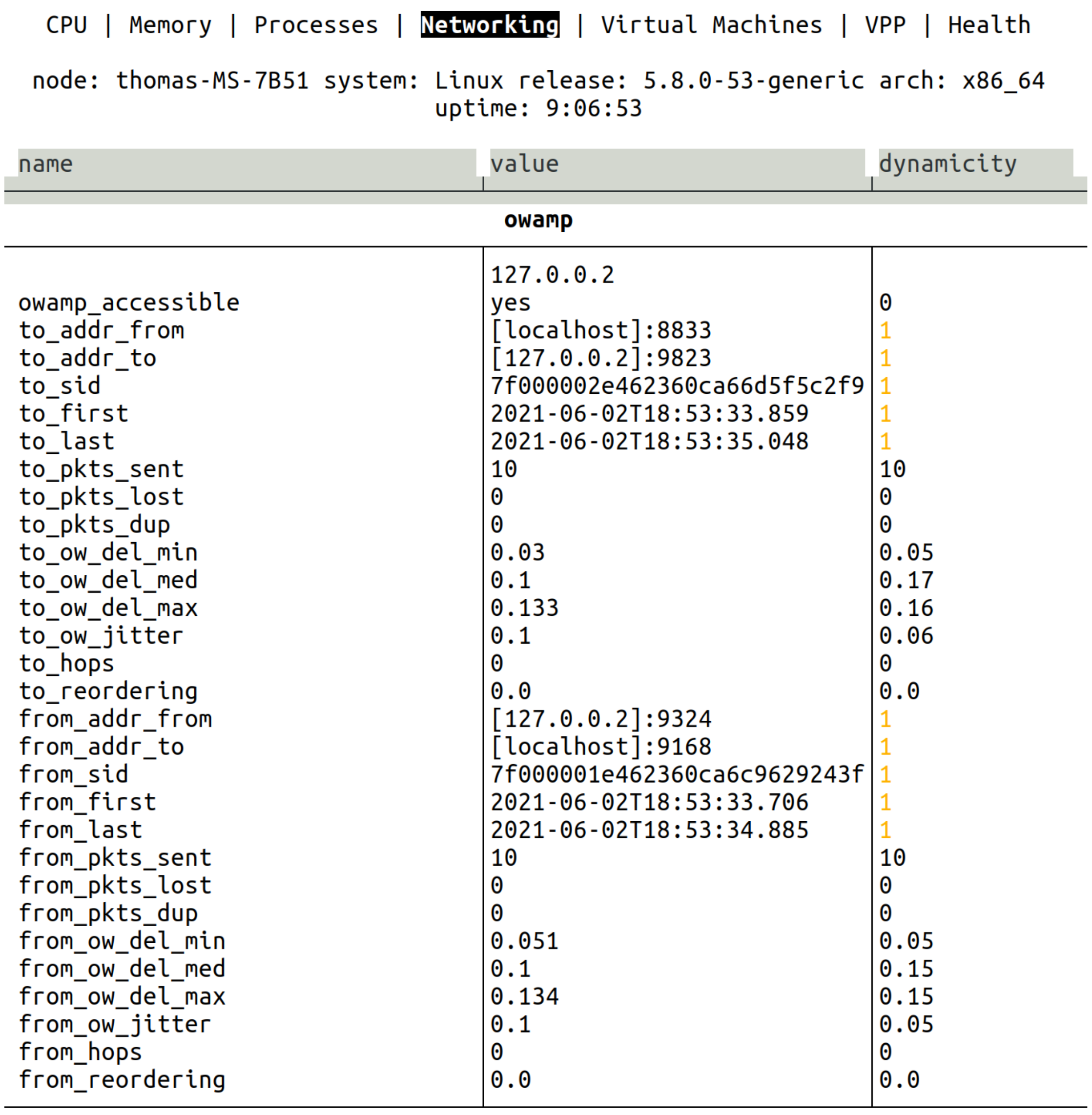
However, there is a problem to be aware of with regard to reading and writing. Indeed, we have to make sure what would happen if the reading of the DxAgent took place at the same time as a writing by the OWAMP scheduler. The chosen way to remedy to it is the use of atomic writing. There is a python library named `python-atomicwrites` [30] able to realize this type of writing. The way it works is rather simple. The writing takes place in a temporary file and when the writing is finished, the temporary file is renamed by the file one wants to overwrite. And, as mentioned in the official documentation, the system call `rename` is atomic: "If newpath already exists, it will be atomically replaced, so that there is no point at which another process attempting to access newpath will find it missing. However, there will probably be a window in which both oldpath and newpath refer to the file being renamed" [17]. We can therefore deduce that the DxAgent will read either the old file or the new one and not a file with indeterminate content.

But the problem is not over yet, we have to make sure what happens when the OWAMP input file is renamed (and thus replaced) during a write. Actually, this concern is solved by the way the `open` system call works: "A call to open() creates a new open file description, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags [...]. A file descriptor is a reference to an open file description; this reference is unaffected if pathname is subsequently removed or modified to refer to a different file" [16]. Therefore, if a write of new contents occurs during a read, DxAgent will continue to read the old file. This one is only unlinked and will really be deleted when calling the `close` system call: "if the file descriptor was the last reference to a file which has been removed using unlink(2), the file is deleted" [15].

The display in DxTop

Finally, once the data has been retrieved, it must be entered into an input dictionary so that it can be analyzed in the next step.

At the same time, the dictionary has been added (by means of a shareable buffer) to the Network category of DxTop thinking of all possible scenarios again. Indeed, the refreshment of the DxTop display being also scheduled every 3 seconds, it was necessary to think about several particular cases (as the first ping done late compared to the dxagent scheduler, the ping stopped in the middle, etc). A representation of the new integrated inputs displayed by the DxTop is present on the figure 3.8 below. It should be noted that the accessibility input is not really an input that OWAMP gives but rather an input that can be inferred from the output of the ping program.



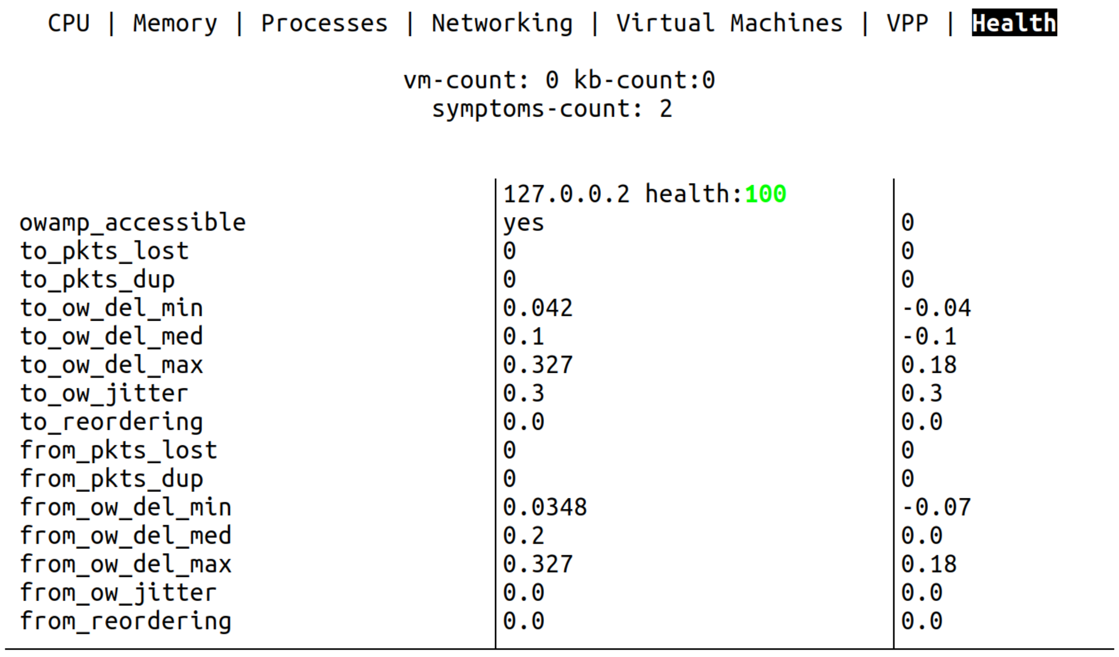
```
CPU | Memory | Processes | Networking | Virtual Machines | VPP | Health

node: thomas-MS-7B51 system: Linux release: 5.8.0-53-generic arch: x86_64
                          uptime: 9:06:53

name                              value                          dynamicity
                                  owamp
                                  127.0.0.2
owamp_accessible                  yes                            0
to_addr_from                      [localhost]:8833               1
to_addr_to                        [127.0.0.2]:9823               1
to_sid                            7f000002e462360ca66d5f5c2f9    1
to_first                          2021-06-02T18:53:33.859        1
to_last                           2021-06-02T18:53:35.048        1
to_pkts_sent                      10                             10
to_pkts_lost                      0                              0
to_pkts_dup                       0                              0
to_ow_del_min                     0.03                           0.05
to_ow_del_med                     0.1                            0.17
to_ow_del_max                     0.133                          0.16
to_ow_jitter                      0.1                            0.06
to_hops                           0                              0
to_reordering                     0.0                            0.0
from_addr_from                    [127.0.0.2]:9324               1
from_addr_to                      [localhost]:9168               1
from_sid                          7f000001e462360ca6c9629243f    1
from_first                        2021-06-02T18:53:33.706        1
from_last                         2021-06-02T18:53:34.885        1
from_pkts_sent                    10                             10
from_pkts_lost                    0                              0
from_pkts_dup                     0                              0
from_ow_del_min                   0.051                          0.05
from_ow_del_med                   0.1                            0.15
from_ow_del_max                   0.134                          0.15
from_ow_jitter                    0.1                            0.05
from_hops                         0                              0
from_reordering                   0.0                            0.0
```

Figure 3.8: DxTop - OWAMP Inputs

The address 127.0.0.2 is the one for which we see the link inputs, one just need to scroll down to see the inputs for other addresses. The inputs starting with the prefix **to** refer to the values calculated by OWAMP from the monitored machine to the target address. While the inputs with the prefix **from** refer to the information specific to the return path.

3.3.2.2 Metric

As explained in section 1.3.1.2, this step consists in selecting and normalizing the inputs of the previous step. In my case, I don't need to normalize the data because they all come from the same input source i.e. the OWAMP API. The selection of the metrics among the inputs was rather simple, once the timestamps and addresses were removed from the inputs, only important information for the health of the link remained. We can, for example, see the list of metrics chosen from a screenshot of the DxTop program showing the values of these metrics for a loopback address of my computer on the following figure 3.9.



```
CPU | Memory | Processes | Networking | Virtual Machines | VPP | Health

                          vm-count: 0 kb-count:0
                            symptoms-count: 2

                                 |127.0.0.2 health:100      |
 owamp_accessible                |yes                       |0
 to_pkts_lost                    |0                         |0
 to_pkts_dup                     |0                         |0
 to_ow_del_min                   |0.042                     |-0.04
 to_ow_del_med                   |0.1                       |-0.1
 to_ow_del_max                   |0.327                     |0.18
 to_ow_jitter                    |0.3                       |0.3
 to_reordering                   |0.0                       |0.0
 from_pkts_lost                  |0                         |0
 from_pkts_dup                   |0                         |0
 from_ow_del_min                 |0.0348                    |-0.07
 from_ow_del_med                 |0.2                       |0.0
 from_ow_del_max                 |0.327                     |0.18
 from_ow_jitter                  |0.0                       |0.0
 from_reordering                 |0.0                       |0.0
```

Figure 3.9: DxTop - OWAMP Metrics

A second operation that takes place during this step is the update of the dependency graph, i.e. adding or removing nodes from the graph. It was obvious that I should add the metrics of OWAMP in a subservice of the `net` subservice. Before the integration of OWAMP, the `net` subservice was only composed of `if` (for 'interface') subservices, collecting, for each interface, data specific to it. We can see an example of this graph from DxWeb with the interfaces (eno1 and lo) of my computer on the figure 3.11.

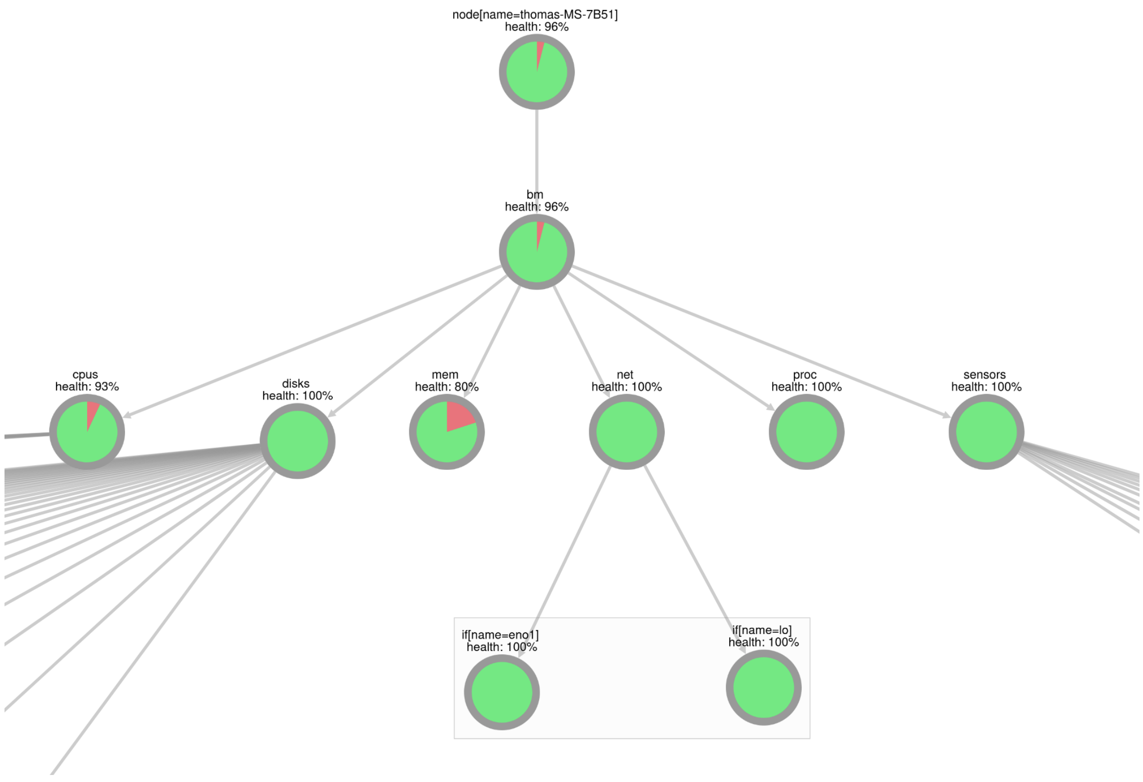


Figure 3.10: DxWeb - net node example

However, I thought that it would not be interesting to add the inputs that I integrate to these `if` subservices because it is rather the health of a communication between two machines we want to establish regardless of the interface. Therefore, a `link` subservice is created for each address whose connection status we want to analyze. This subservice is completely optional. In fact, as long as no ping information (whether it is a hit or not) is retrieved, no `link` node is added to the graph. We can see an example of this graph when we want to establish the health score with three target addresses.

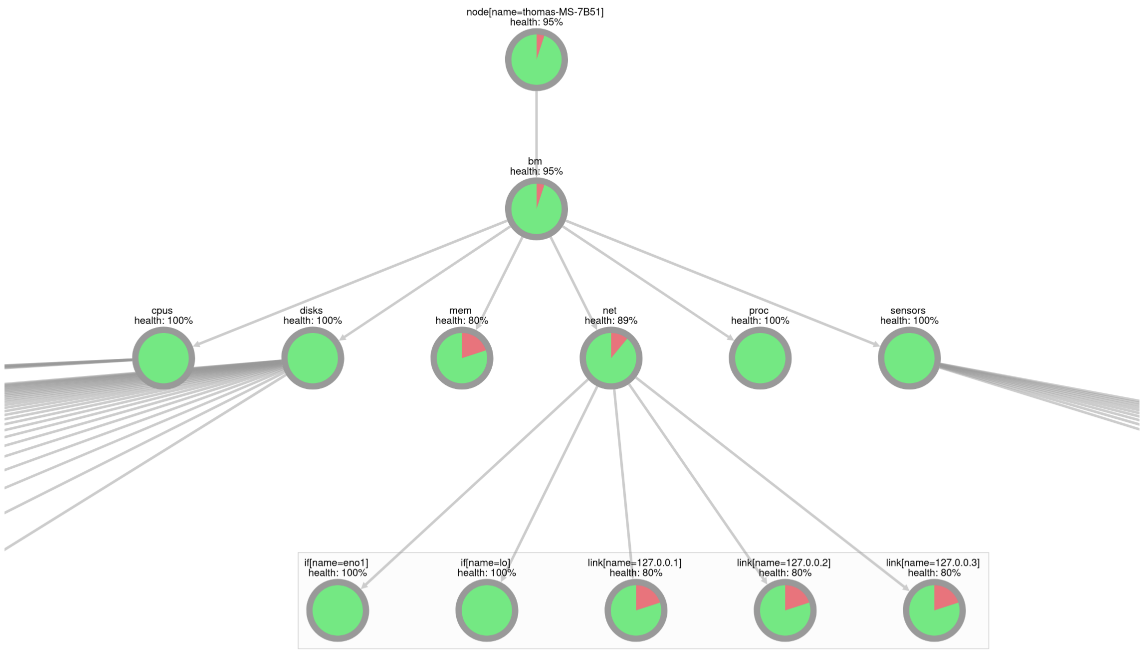


Figure 3.11: DxWeb - Net nodes examples with three links

We can see on the figure 3.11 that the links have a health of 80%, which is normal given the rules that were in force at the time of the screenshot. Moreover, we can notice that the name of these 3 link instances represents the addresses targeted by the pings.

3.3.2.3 Rule

This last step in the Dxagent process is not really something that needs to be implemented because it is up to the network user to know what he wants. Indeed, this last step consists in choosing a set of rules allowing to establish the health of the links and to introduce them in a csv file. Thus, the rules that we want to apply depend largely on the infrastructure that we have and the services that we want to provide. However, it would be interesting to present different ways of creating rules with the metrics I have integrated.

Let's just remember that the rules have a format similar to the operations useful for conditions that can be performed in Python (==, <, >=, etc). There are also constructs which ensure that a condition is true for a given time :`1min()` or `5min()`. And finally, one can embed a metric of the function `dynamicity` so that it is the dynamicity that is analyzed (i.e. the average of the variable through time). Finally, the gravity color (red or orange) must be associated with it. Here is a list of rules that we can imagine for the OWAMP metrics:

- The first rule is essential and rather obvious, it is simply to ensure the reachability of the target address. We could assign an orange severity if it occurs for less than five minute and a red serverity when it exceeds 5 minutes.

```
Address not reachable : owamp_accessible == "no"
Address not reachable (>=5 min) : 5min(owamp_accessible == "no")
```

- We can also indicate an anomaly when there is at least one case of duplicated or lost packets on the path **to** the target machine. We can also make sure that there is no reordering on this path.

```
Path TO : Packet loss : to_pkts_lost > 0
Path TO : Packet duplication : to_pkts_dup > 0
Path TO : Packet reordering : to_reordering == "true"
```

- We could also assure, for example, that there is never a delay on the path **from** the target machine greater than 1 second otherwise a symptom of red severity will be released.

```
Path FROM : Maximum delay exceeded : from_ow_del_max > 1.0
```

- But it is obviously also possible to create rather complex rules. For example, one can imagine that one wants the sum of the averages of the median one-way delays in **both** directions to be greater than 2 seconds for at least 5 minutes.

```
5min(dynamicity(to_ow_del_med) + dynamicity(from_ow_del_med)) > 2
```

Of course, these are only examples, but it could allow a network operator to be inspired by these few ideas in order to realize his own rules meeting his desired criteria.

3.3.3 Discussion about OWAMP

After having learned about the OWAMP protocol and how to integrate it with DxAgent, I think it is important to come back to the strong and weak points of this measurement tool.

First of all, I think it is important to remind how useful it can be to get different delays and information depending on the direction of the packets (given a possible assymetry in the network). The accuracy of the protocol and its "one-way" aspect make it a remarkable tool. Another important point of this protocol is the ability to authenticate ping requests and thus to allow pings only to authorized users. In the case of DxAgent, it is enough to share a passphrase between the different OWAMP entities of an infrastructure, the developed API takes care of the authentication.

However, the obligation to have a server on a machine one wants to target makes the number of accessible machines very small. Especially since OWAMP is far from being present on all machines. Furthermore it is possible that the ports required for OWAMP-Control (in TCP) and OWAMP-test (in UDP) are filtered and will therefore create even more complications. If a network operator decides to have one OWAMP server per entity in his network and does not need information from other machines, this will suffice. But if they need to know the delay or reachability of an external machine (e.g. an external cloud server), OWAMP will not be able to meet this need. That's why I decided to add another probing tool to DxAgent: the ICMP ping.

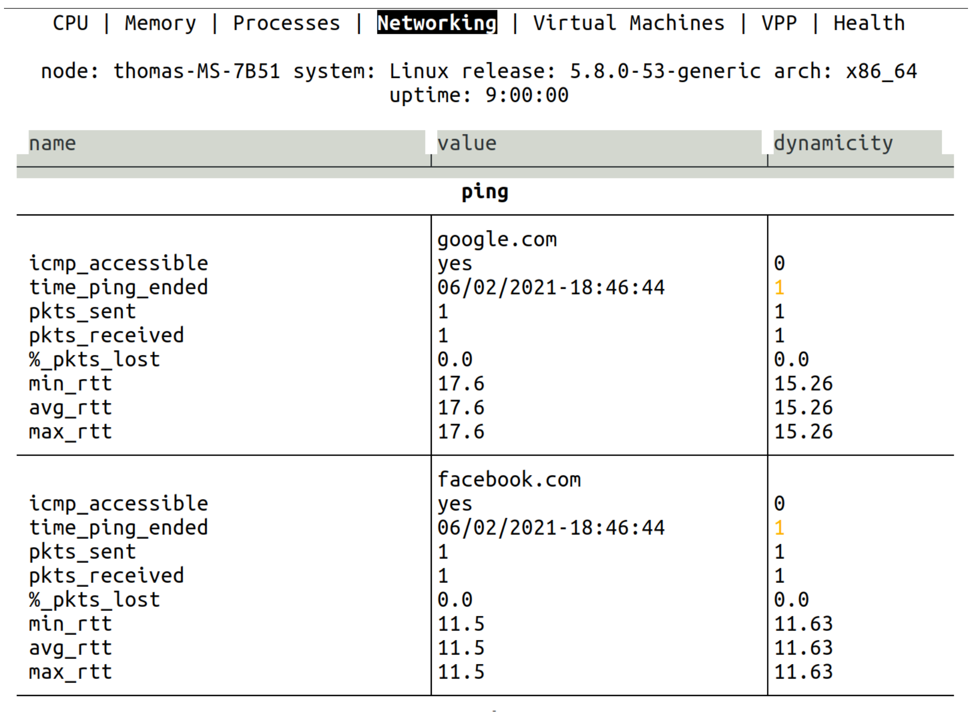
3.3.4 Integration of ICMP pings in the DxAgent

In addition to being a good complement to OWAMP, as just explained, ICMP ping will be a good candidate for comparison with OWAMP.

Since all the steps required to integrate OWAMP into DxAgent have already been presented, it is not necessary to explain them again in the case of ICMP ping. The method used for ICMP pinging is almost the same as for OWAMP except that it has been directly integrated into DxAgent without the need to implement an independent API. Instead, it would be more interesting to explain the few small differences from the implementation point of view.

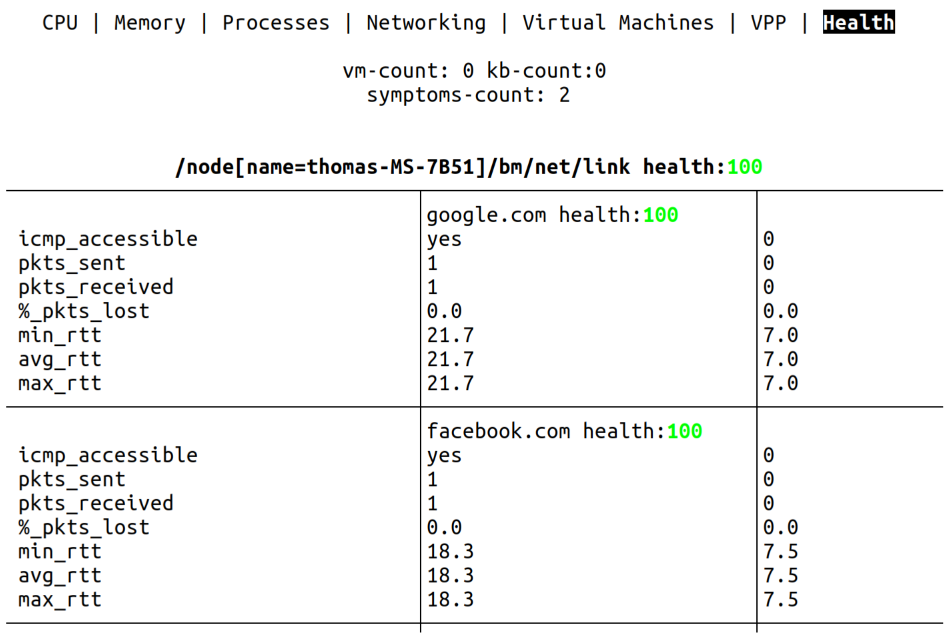
The fping program [15] is used to perform the pings, it is a program able to ping several addresses at the same time (with a round-robin principle) and to give the result of all the pings in stdout. The number of options in the program is quite large and I chose the output options carefully to get the statistics. Once again, I leave the choice of the probing options (number of probes, size, timeout, ping interval, etc) to the user in the configuration file.

And so, to integrate it to the DxAgent, I launch a thread from the DxAgent which is actually a scheduler that will execute the fping process and write the data atomically and periodically into a file. By repeating the steps already done for the integration of OWAMP, we can retrieve the list of inputs (figure 3.12) as well as the list of metrics (figure 3.13) which are available just below:



```
 CPU | Memory | Processes | Networking | Virtual Machines | VPP | Health

node: thomas-MS-7B51 system: Linux release: 5.8.0-53-generic arch: x86_64
                          uptime: 9:00:00

name                               value                   dynamicity
                               ping
                                   google.com
icmp_accessible                    yes                     0
time_ping_ended                    06/02/2021-18:46:44     1
pkts_sent                          1                       1
pkts_received                      1                       1
%_pkts_lost                        0.0                     0.0
min_rtt                            17.6                    15.26
avg_rtt                            17.6                    15.26
max_rtt                            17.6                    15.26

                                   facebook.com
icmp_accessible                    yes                     0
time_ping_ended                    06/02/2021-18:46:44     1
pkts_sent                          1                       1
pkts_received                      1                       1
%_pkts_lost                        0.0                     0.0
min_rtt                            11.5                    11.63
avg_rtt                            11.5                    11.63
max_rtt                            11.5                    11.63
```

Figure 3.12: DxTop - ICMP Inputs



```
 CPU | Memory | Processes | Networking | Virtual Machines | VPP | Health

                     vm-count: 0 kb-count:0
                       symptoms-count: 2

        /node[name=thomas-MS-7B51]/bm/net/link health:100

                                   google.com health:100
icmp_accessible                    yes                     0
pkts_sent                          1                       0
pkts_received                      1                       0
%_pkts_lost                        0.0                     0.0
min_rtt                            21.7                    7.0
avg_rtt                            21.7                    7.0
max_rtt                            21.7                    7.0

                                   facebook.com health:100
icmp_accessible                    yes                     0
pkts_sent                          1                       0
pkts_received                      1                       0
%_pkts_lost                        0.0                     0.0
min_rtt                            18.3                    7.5
avg_rtt                            18.3                    7.5
max_rtt                            18.3                    7.5
```

Figure 3.13: DxTop - ICMP Metrics

It may be interesting to note how these pings fit into the DxAgent dependency graph. In fact, these are also embedded in the `link` subservice, just like OWAMP. Of course, it is quite possible to ping both OWAMP and ICMP at the same time, so the subservice will contain both sets of metrics. This is interesting because it is for example possible to have data redundancy. Let's imagine that a machine is both probed by OWAMP and ICMP, we can then make sure that a machine is not reachable if neither of the pings can access it. It is possible that the OWAMP ping does not hit because the server is off, and therefore the ICMP ping can still reach it. But the opposite is also possible, if for example the OWAMP server is well accessible but the ICMP echo request packets are blocked by a firewall. The rule below could be more reliable than having only OWAMP or ICMP pings:

```
    Address not reachable :  (owamp_accessible == "no") and (icmp_accessible == "no")
```

We will see even better in the next chapter, which deals with a use case, the interest of having these two different sources of information.

3.3.5 Testing and Limitations

The interface as well as the integration into the OWAMP DxAgent was extensively tested manually during the whole process. But a tool I developed for the use case we will present in the next chapter allowed me to observe and fix several bugs. This tool allows to automate DxAgent scenarios in a rather complex topology. So I just had to realize different scenarios varying the different parameters of the program to test its efficiency in a case as close to reality as possible. The way it works will be well explained in the next section.

Chapter 4

Use Case Demonstration

In this chapter, we will show how the integration of OWAMP with DxAgent works through a use case. This use case will show how and why DxAgent combined with OWAMP can be useful in a real and practical case. We will see that it can be very well integrated into a network architecture in order to quickly diagnose anomalies. Moreover, this use case will be the occasion to test the program in order to test a large part of its different uses. By varying the parameters of the program in the face of different simulations of anomalies, we will be able to ensure that it works properly. All the work realized for this use case is present on the following github repository: `https://gitlab.uliege.be/Thomas.Carlisi/dxagent_use_case`

4.1 Use Case Presentation

The use case that has been chosen is that of a small network architecture offering a service to its customers. This service could be anything, but let's take the case of a video streaming service for the example. We are then in the case of a company that has to provide video streaming to its customers and therefore has certain network requirements to meet. The way in which the service has been set up does not change the use case but we can imagine that it is an Intent-based Networking architecture and that the streaming service has been set up using an intent. Therefore, we would like to show that the DxAgent can observe if an intent drift is present or about to appear by analyzing the network continuously. That is, to determine if the intent that was decided upon is not compromised due to a change in network conditions. The network topology has been defined in order to show several interesting cases and it is presented on the following figure 4.1.

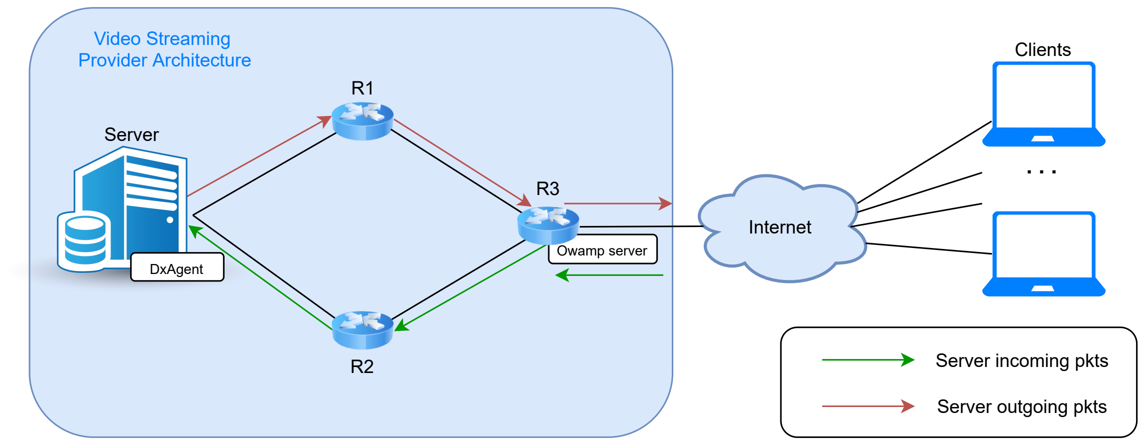


Figure 4.1: Streaming service Topology

The first thing that is important to note in relation to this figure is that the entities present in the architecture of the streaming provider could represent only a subset of the entities of the network enterprise. The goal here is not to create a complete topology or one that is completely consistent with reality, but to show a use case that will highlight the main asset of OWAMP, which is the retrieving of one-way metrics. This is why the diamond formation composed of the three routers (R1, R2 and R3) and the server has been imagined. It can also be noticed that the server (or the data center to see the bigger picture) that provides streaming to the client integrates a DxAgent. It is indeed in this entity that the measurements will be initiated and retrieved. While an OWAMP server is present in the router at the exit of the service provider network to analyze the traffic between this router and the server.

It would be interesting to imagine the intent that could have been involved for such a service in order to have what we want to assure. In this use case, the intent, expressed in common language, would be for example the following:

> "One wants a connection between the server and R3 such that outgoing packets go through a different path than incoming packets and whose outgoing one-way delay does not exceed 50 ms".

For this reason, Figure 4.1 shows a topology where outgoing packets pass through R1 and incoming packets pass through R2. Furthermore, we notice that the intent example that was given shows a delay requirement only in the direction of packets leaving the server. This might make sense because the service provider wants to ensure that video packets are received quickly while it is potentially less important that the delay for client acknowledgement packets be as low. Of course, this is just an example and we will see in the demonstration that it is possible to assure other parameters with DxAgent and OWAMP such as reordering for example. Moreover, we can note that the delay between the server and the client is not observed and this is reasonable. The intent and its assurance only concern what can be managed, i.e. the internal network entities. If the delay deteriorates significantly once the packets are outside the internal network, this is not something the service provider can influence.

We will therefore see through several experiments the different use cases of the active network metrics recovered by DxAgent. But first it is interesting to present the way this topology and these experiments have been set up and implemented.

4.2 Use Case Setup

The implementation of the topology is done by means of docker containers [24] (one container per machine) that communicate with docker bridges [4] that are created and managed by Docker-Compose [5]. It seems important to briefly summarize these technologies in order to better understand how they could be used to implement the topology.

4.2.1 Creation of Docker Containers

First of all, a `Docker image` is a standalone software package that allows one to run an application with everything it needs. This includes all libraries (system or not), programs, code, parameters required to run the application. Then, when this image is run on a Docker Engine, we say that it is a `Docker container`. Without going into details, the Docker Engine is the technology allowing to run in a consistent way the containers on different platforms. The main interest of Docker is therefore to be able to deploy an application using a container without having to worry about all the problems of dependencies depending on the platform. One just needs to create an image that contains everything one needs to run an application and this application can then be run on a number of different systems in a consistent way and without additional installation.

For my part, I use it above all to simulate the different entities of my topology. The use I make of it allows me first to simulate a network of several machines on a single physical machine. And secondly, given the consistent nature of Docker, it allows me to create a use case and experiments that are very easily reproducible. Moreover, in addition to being easy to reproduce, the experiments are easy to automate.

The specification of the Docker image that will be built is done through a Dockerfile. The purpose of this file is to state the different steps necessary to launch the application. The first step is the choice of the base image, it is a basic software package allowing not to have to specify the whole of the dependencies (in particular the very many needs at system level). In my case, I chose the Ubuntu image but any other version of Linux was possible (to run the DxAgent). From this image, one just has to install some programs (fping, iproute, etc) and libraries necessary for DxAgent and experiments. Then, it is enough to inform in the Dockerfile the files of the code of DxAgent so that it is copied and then to launch the application.

So I had to specify one Dockerfile per machine in my topology. The routers are not really routers but simply linux machines configured to forward packets and whose application is simply a program that waits indefinitely. However the "router" at the exit of the network must also launch an OWAMP server (configured by my python API). The fact that it is a simulation of a linux computer and not a router makes this possible. In addition, a series of modifications had to be applied to the DxAgent so that it could run in a container. The DxAgent reading some files from `/proc` which are not part of the Ubuntu image of Docker, it was necessary to remove some inputs. Other little adjustments had to be made so that the DxAgent could work inside a container. To provide an example of a Dockerfile, here is a simple one used to launch the router being at the exit of the network (the one that integrates OWAMP):

```
FROM ubuntu:20.04

COPY owamp /app/
COPY run.sh /app/
COPY close_server.sh /app/

WORKDIR /app

RUN apt-get update -y && \
    apt-get install -y net-tools && \
    apt-get install -y iproute2 && \
    apt-get install -y traceroute && \
    apt-get install -y iptables

CMD /bin/bash run.sh
```

The Dockerfile keywords are explicit and it is quite easy to understand what is going on. One just has to note that instead of launching the OWAMP server, it is a `run.sh` script that is launched. Indeed, the server being a daemon, once started by the script, one has to wait indefinitely so that the container does not stop its execution.

4.2.2 Connecting the Docker Containers

Being able to launch all the containers is a first step but then you have to be able to make them communicate with each other and using the right routes. To do this, I used Docker-Compose, a tool able to configure (by means of a yaml [34] file) then launch several containers at the same time. Among what can be configured with Docker-Compose, the most important for this project is the Docker Networks. The way the network is managed by Docker is rather particular and it seems important to explain it briefly.

Docker's networking subsystem is pluggable using drivers. Among the different drivers proposed by Docker, the one that best suited my use case is the bridge. A bridge network allows containers connected to the same bridge to communicate while it provides isolation from containers that are not connected to the bridge. The docker bridge driver will take care of setting up the rules on the host machine to allow the connection and/or isolation of containers. By default, docker containers are connected to the default bridge network, and therefore are all connected to each other. The following figure 4.2 shows the default behavior of the network if we launch 3 containers.

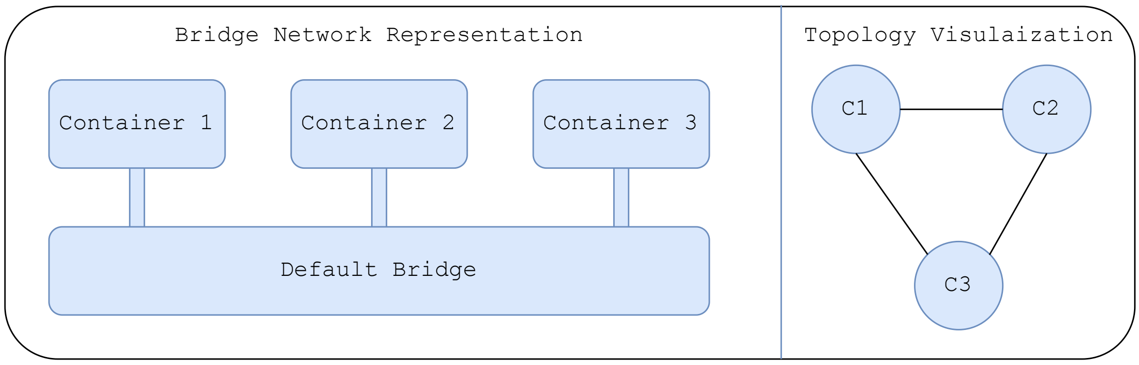


Figure 4.2: Bridge Default behavior

However, given the topology that is targeted for this use case, we want to isolate the containers 2 by 2 in order to simulate the links between the different entities of the topology. The following figure 4.3 shows what we would like to have with three containers which can, for example, represent the link between the server, R1, R2 R3 of figure 4.1.

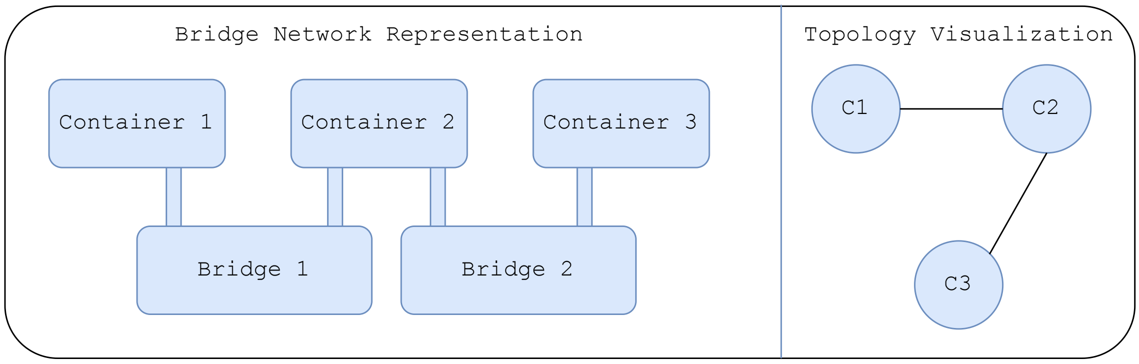


Figure 4.3: Bridge wanted behavior

Since it is possible to create user-defined docker bridges, it is sufficient to create a bridge for each link to connect the containers 2 to 2. Therefore, if a container must be connected to three others for example, it will have three interfaces, one for each of the three bridges connecting it to the three containers. Moreover, it is possible to determine the IP addresses of each interface of the containers and thus have a relatively high control over the topology that we want to achieve.

The topology that has been realized is not really the one that has been presented, especially because the clients are not present because they would not bring anything to the experiments that will be done. Moreover, two routers have been added to simplify the network configuration which is already not obvious on Docker. These two routers aim to avoid a problem that was present with OWAMP and ICMP pings. This problem is that, due to the initial topology and the mandatory asymmetry of the path between outgoing and incoming packets, OWAMP/ICMP packets were sent by one interface and received by another, which prevented pings from taking place. A simple solution was to add routers so that the server and the outgoing router sent and received packets over a single interface. There are more elegant solutions to this problem (such as using loopback addresses), but in addition to being simple to implement, this solution allows more flexibility in the choice of experiments to be performed (because there are more links). Once the bridges have been created, the IP addresses chosen and the routing tables configured, we find ourselves with the topology shown in the following figure 4.4.

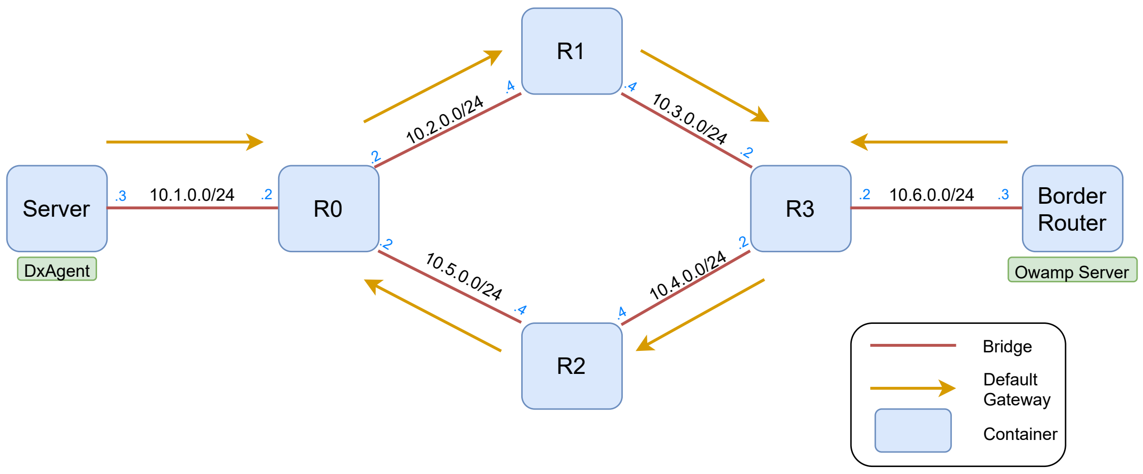


Figure 4.4: The realised docker topology

From now on, we will always use this topology and its names as a reference for the following because this is the one that has been realized in practice. So we have one bridge per pair of containers that must be connected. Moreover, the bridge subnets and the interface addresses have been determined in such a way that they are not chosen randomly but are always consistent. It is quite easy to convince oneself that defining a default gateway per interface is sufficient to set up the desired routing. Indeed, when a container is not directly connected to another by a bridge, it must always reach the other containers by a single path, given the asymmetry that we want to achieve. Remember that the objective is to analyze the path between the server and the border router, i.e. the router that is directly connected to the Internet. In the end, the path that the OWAMP and ICMP packets will take is the following:

Server → R0 → R1 → R3 → Border Router

Then,

Border Router → R3 → R2 → R0 → Server

To see how this was done in practice, let's take the example of R3 communicating with R1, R2 and the Border Router. The part dedicated to R3 in the yaml file for docker-compose is the following:

```
r3:
    depends_on:
      - server
      - r0
      - r1
      - r2
    container_name: r3
    build: ./r3/
    sysctls:
      - net.ipv4.ip_forward=1
    networks:
      r1-r3:
        ipv4_address: 10.3.0.2
      r3-br:
        ipv4_address: 10.6.0.2
      r3-r2:
        ipv4_address: 10.4.0.2
```

The dependencies are simply used to choose the order in which the containers will be started, the `build` directive indicates the folder in which R3's Dockerfile and the files it needs are contained. The `sysctls` directive is used to configure R3 to forward ipv4 packets, which is required to simulate a router. And finally, the `networks` refer to the docker bridges that have been created beforehand with the addresses assigned to each of its bridges (i.e. its interfaces). Then we can see the creation of the networks associated with it and the change of the default gateway below:

```
### Before images built

docker network create --subnet 10.3.0.0/24 r1-r3
docker network create --subnet 10.6.0.0/24 r3-br
docker network create --subnet 10.4.0.0/24 r3-r2

### After containers started

# r3 redirects to r2
docker exec --privileged r3 route delete default
docker exec --privileged r3 route add default gw 10.4.0.4
```

It should be noted that the change of route is not done at the creation of the docker image but after the creation of the container because it is an operation that is only possible by extending the privileges of the container. This is why we execute the `route` command on a container already started with the `--privileged` option. The main part of the operations that allowed to make the different containers communicate have been presented, we will quickly see how the experiments could be automated.

4.2.3 Automation of experiments

As stated, using Docker makes it easy to automate the startup of the topology and the flow of actions. But before one can automate anything, it is important to be able to retrieve data. The DxAgent has two graphical interfaces that give real time results but these results are difficult to use. Moreover, the containers are launched in background, so we need another way to retrieve the metrics and symptoms from the DxAgent in an easily usable format. To do this the DxAgent has been modified to produce a csv file, at each update of the program (every 3 seconds), giving the timestamp of the update, all the metrics and symptoms that can be returned from the OWAMP and ICMP inputs.

Once it is possible to get the data from the DxAgent, it is possible to imagine the automation of the experiments. To do this, a python script has been written. The script takes as a parameter of the program a scenario, that is to say a series of events that must take place in the different containers, and it returns the output that was delivered by the DxAgent during the whole scenario. In addition to the scenario, a series of parameters can be entered in a configuration file and the script will take care of modifying certain variables of the DxAgent and OWAMP/ICMP parameters using preconfigured jinja2 [28] templates. It is therefore possible to create all sorts of scenarios by playing with all possible parameters easily (without needing to understand anything about DxAgent, Docker, etc). Once the parameters have been modified in the DxAgent files, the script will take care of creating the Docker networks, creating the images, starting the containers, configuring the routes and link delays and then starting the bash scenario entered as the program parameter. Once the scenario is finished, the output is copied from the server container to the host machine and the networks and containers are stopped and deleted.

Let's imagine a very simple scenario, written in bash, that waits for 30 seconds, then changes the delay of one link and waits for another 30 seconds:

```bash
sleep 30

docker exec --privileged r1 tc qdisc \
    replace dev eth1 root netem delay 50ms

sleep 30
```

Let's take advantage of this script to explain a tool that will be widely used in the experiments that will be presented: `tc qdisc` [18]. This linux command allows to change the queuing discipline of the packet that will be transmitted. In our case, we use `netem` [22] which offers Network Emulation features such as changing the delay, loss percentage, reordering percentage, duplicates percentage, etc. Thus, the command presented above will force R1 to wait 100ms before sending its packets.

Then, one just has to modify the configuration file created specifically for the use case and then launch the python script with the above scenario as parameter. And the topology will be started, the scenario executed and a csv file will be generated with the symptoms and metrics of the DxAgent collected during the 60 seconds of the scenario. All this being done in background, the user investment has been reduced to its strict minimum to make it simple to perform a whole series of experiments that are easily reproducible.

4.3 The Demonstration

It is now possible to perform all kinds of experiments to test the different parameters of DxAgent, OWAMP and ICMP. Furthermore, we can demonstrate that the inputs that have been added to the DxAgent work properly and can react to different situations that may be useful in a real case. For each experiment we will perform, the scenario, the objective that is aimed at, the chosen parameters and the DxAgent rules used will be exposed. Given the number of events that are applied to the network and the large list of inputs added to the DxAgent, the number of experiments to be performed is immensely large. Four experiments showing the most important and interesting aspects to be discussed have therefore been carefully chosen.

It is important to first present the initial state which is the same for each experiment. The 6 entities of the topology with their links and routes which were presented on the figure 4.4 are started. And a delay of 2ms is emulated (thanks to netem) on each and every link. The original delay is very short and unpredictable because everything takes place on the same machine, therefore a delay of 2 ms has been set. And thus, if del_{x_y} represents the one way delay from `x` to `y`, the expected (minimum) delay between the server and the border router is the following (since the packets go through 4 links in both directions):

$$del_{S\_BR} = del_{Br\_S} = 4 \times 2 = 8ms$$

Interval between pings

Before presenting the different experiments, there is a point that it is important to discuss, it is the maximum time to realize a ping in its entirety. Indeed, it is important to take this time into account because a ping must be performed in its entirety for the result to be recovered by the DxAgent. Moreover, it is important to know this time in order to decide on the interval between each complete ping. It is important to remember now the difference between the time of the OWAMP API scheduler between each ping and the schedule to define for the ping. The first one represents the time between each complete ping while the second one corresponds to the time between each packet sent within the same session, i.e. the same ping. Let's now analyze this time for OWAMP and ICMP.

For OWAMP, in the implementation that has been made, it starts the test phase (i.e. the ping) with a certain delay, this is called the delay start. This delay is useful to make sure that the test session starts after the setup protocol is completed. This configurable delay is minimum 1 second (plus 2 to 3 RTTs), we will leave it at this default value for the experiments. In addition to this, it is obvious that we have to wait for the number of test packets multiplied by the time interval between each packet sent, which is called the schedule. Finally, the OWAMP protocol waits for a time equal to the configurable timeout before ending the session. Of course, this is only an estimate, one should also take into account the time for the OWAMP-Control session to be completed as well as the writing of the results. But this estimate is sufficient to choose the adjustable interval between each complete ping to be performed.

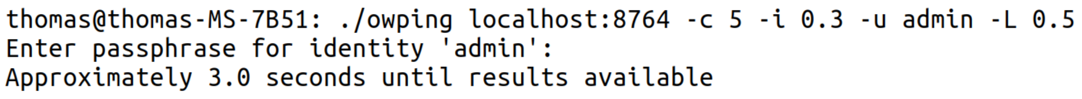
So we have the following result:

$$\text{time}_{ping} \approx \text{delay start} + \text{timeout} + \text{pkts number} \times \text{schedule}$$

Considering that in our case the two to three RTTs of the delay start is insignificant (in the use case, it is between 32 and 48 ms). If we perform a ping with a timeout of 0.5 seconds, a number of packets equal to 5 and a schedule of 0.3, we have :

$$\text{time}_{ping} \approx 1 + 0.5 + 5 \times 0.3 = 3s$$

It is easy to check this result by testing these parameters with the owping program because it gives an estimate of the ping time in output. Here is the result predicted by owping on figure 4.5:



```
thomas@thomas-MS-7B51: ./owping localhost:8764 -c 5 -i 0.3 -u admin -L 0.5
Enter passphrase for identity 'admin':
Approximately 3.0 seconds until results available
```

Figure 4.5: owping example estimate

Of course, this is only an estimate, but it allows us to choose the minimum interval between each ping. In the example we gave, we could simply take a margin of error of 0.5 seconds and therefore choose an interval of 3.5 seconds between each ping. Obviously, we could choose 20 seconds, 100 seconds, or even one ping per day, depending on the use we want to make of DxAgent. For the experiments to follow, we will always take an interval close to the time to perform a complete ping in order to perform a maximum of new measurements in a minimum of time. Remember that the shorter the interval between pings, the more often the DxAgent will retrieve new values during its updates. Moreover, a short ping interval allows to give values as close as possible to real time. It is obvious that if DxAgent updates its metrics every 3 seconds and if a ping is scheduled every hour, DxAgent will at some point have values that are almost an hour old.

In addition, the time to ping with ICMP is similar to the one that can be done with OWAMP. The difference is that there is no mandatory 1 second start delay. Now that the problem of the interval between pings has been discussed, we can explain experiment 1 which is directly linked to it.

A word on parameters

It is possible to influence several parameters of OWAMP and ICMP ping but those influence weakly the experiment to be interesting to highlight. Indeed, the first experiment was tested by varying all the parameters (schedule, packet number, packet size, timeout) but it makes almost no change. By taking values that make sense, there is almost no variation in the delay measurements nor in the DxAgent symptom observations. However, when taking nonsensical (absurdly large) values for the parameters such as sending 100 packets per second of 10,000 bytes for 10 seconds, we observe a little change in the median delay from 16.5 to 16.7 ms. Since these values are sensless for a simple ping and the effect is still very small, these parameters will not be varied in the experiments. Therefore, we will use a arbitrary but reasonable parameter values (close to the default values of the ping protocols) for all experiments which are the following:

- schedule = 0.1 seconds
- packet size = 100 bytes
- timeout = 0.5 seconds

Only the number of packets per ping will be adjusted so that packets are continuously sent during the whole scheduler interval.

4.3.1 Experiment 1 : Delay and ping interval

The first experiment aims at demonstrating that the DxAgent reacts well to delay changes on the network but also to show the importance of the choice of the time interval between pings. We would like to observe how this interval can influence the observations of the DxAgent.

Presentation

The scenario that will be studied in this first experiment is represented in the following figure 4.6:

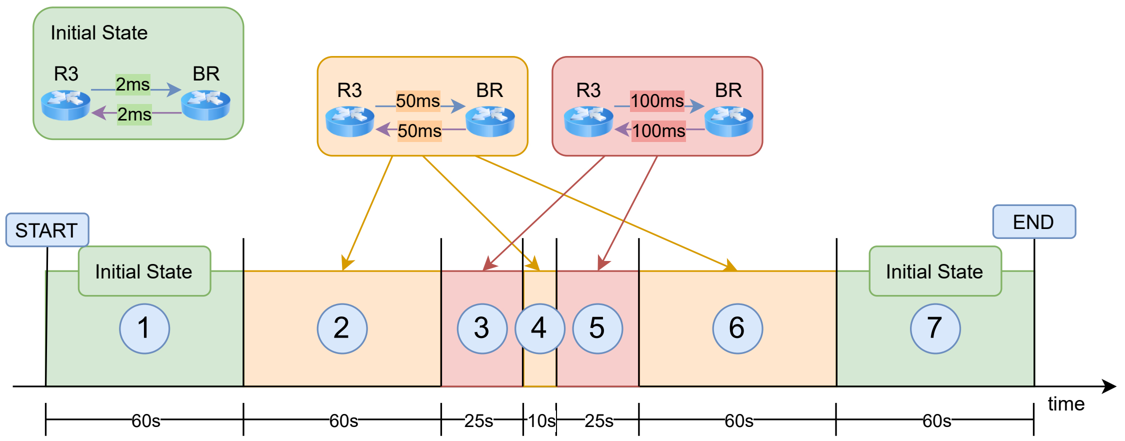


Figure 4.6: Scenario 1

The states 1 and 7 correspond to the delay of the initial state (i.e. 2ms for each link). For states 2, 4 and 6, the link transmission delay between R3 and the border router (BR) is replaced by 50 in both directions. While states 3 and 5 correspond to an even greater delay rise of 100ms in both directions. The choice of this scenario is motivated by the fact that the DxAgent should able to determine a higher or lower delay rise and also to detect the return to normal which should be materialized by an absence of symptom. Moreover the 3 fast changes represented by states 3, 4 and 5 have for objective to see how a long ping interval could manage fast changes in the network

The DxAgent rules that will be used are the following:

```
Owamp rtt > 100 : (from_ow_del_max+to_ow_del_max <=200) and
                  (from_ow_del_max+to_ow_del_max >100)
Owamp rtt > 200 : from_ow_del_max+to_ow_del_max >200
Icmp rtt > 100 : (max_rtt >100) and (max_rtt <=200)
Icmp rtt > 200 : max_rtt >200
```

It is simply a matter of returning a symptom of orange severity when the maximum RTT[1] of a ping (OWAMP or ICMP) exceeds 100 ms and of red severity if it exceeds 200ms.

Results

The first action that was done was to measure when a symptom was detected by the DxAGent and what its nature was. This measurement, which uses a ping interval of 3 seconds, was carried out with OWAMP and then ICMP and gave vigorously equal results. For this reason, we will only show the OWAMP result. The objective of this first measurement is to show that the DxAgent reacts well to network changes. Here is a graph made from the csv output file that was generated during the experiment:

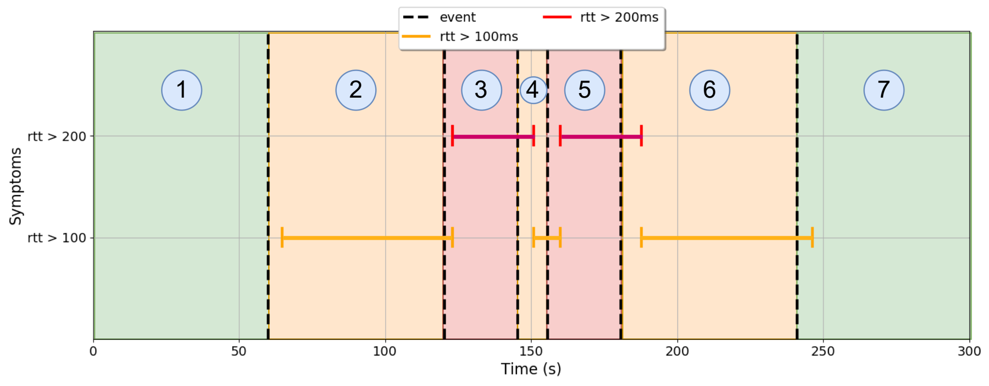


Figure 4.7: Symptoms - experiment 1 (OWAMP)

What can be directly observed on figure 4.7 is that the symptoms are indeed observed when changes of delay take place. However, we can notice that the symptoms are discovered with a small delay. This result is obvious, the DxAgent performs updates of its metrics every 3 seconds and OWAMP independently performs measurements every 3 seconds. Therefore, there will always be a small delay between the change in the network and the moment when the DxAgent notices it. For the same reason, the DxAgent stops reporting a symptom for a while when it should not. These results were quite expected and satisfactory, the DxAgent reacts well to changes in the network in a real case.

In addition to having carried out this experiment with 3 second intervals, it was also carried out with other values which are 6, 9 and 12. The objective was to see how this

[1]For OWAMP, the RTT is the sum of the one way delay in both path directions

interval can modify the discovery by the DxAgent of changes in the network. We are sure that when there is a change of delay, the associated symptom (orange or red severity) is returned by the DxAgent. It would therefore be more interesting to show the round trip time measured by the two measurement tools. Let's take the case of OWAMP first:

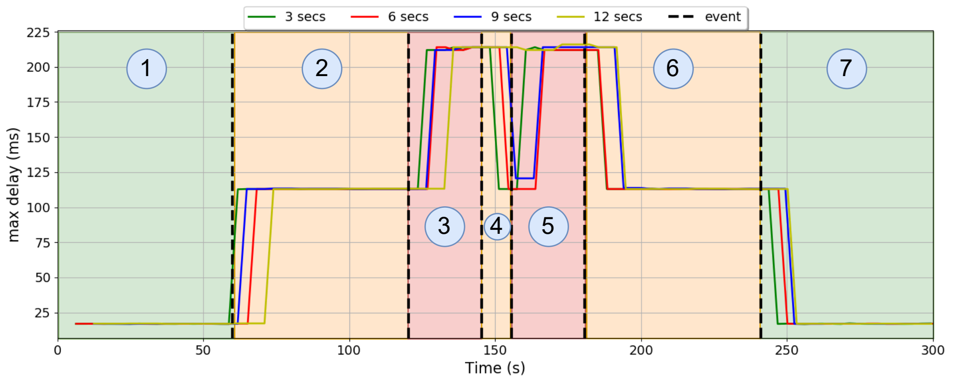


Figure 4.8: Maximum delay - experiment 1 (OWAMP)

If we observe on the graph presented on figure 4.8 a single curve, for example the green curve (3 seconds between pings), we notice as expected that the maximum delay undergoes an increase or a decrease at each network event with a short delay. The three approximate RTT values that can be extracted from the observed measurements are: 17, 113 and 213 ms.

First of all, the initial delay of 17ms was expected, indeed, the packets between the server and the border router pass through 4 hops in one direction and 4 in the other. Since the transmission delay of these 8 entities is delayed by 2 ms, the minimum delay is 16 ms. But this is of course not the only one delay to take into account (propagation, processing, queuing delay must also be considered), so an RTT of 17 seconds is coherent.

Then, the transmission delay of two of these 8 links has been replaced by 50, so we have $6 \times 2 + 2 \times 50 = 112$. Once again, within one millisecond of delay, we have a reasonable result. Finally, when the transmission delay is changed from 50 to 100 for both links, we have $6 \times 2 + 2 \times 100 = 212$ (and therefore close to 213).

What we can then observe on figure 4.8 is the difference between the 4 curves. The bigger the ping interval, the more delay there will be to detect an anomaly. Even worse, when the change lasts only 10 seconds (state 4), an interval of 12 seconds does not even detect the delay modification. Since this is the maximum delay, only one value close to 213 ms is measured during the 12 seconds of probing for the maximum delay to be close to this value. A first solution to this problem would be to use the median delay instead, but this would not solve the problem entirely. It is obvious that if two network changes take place during the same probing, the result will not be consistent. The second and better solution is then to choose an interval between pings as short as possible (close or equal to the interval of the DxAgent updates) when we want to observe short perturbations in time. From now on, all experiments will be executed using a three second interval.

If we observe the same measurements for the same experiment but in the case of traditional pings, we only notice one difference:

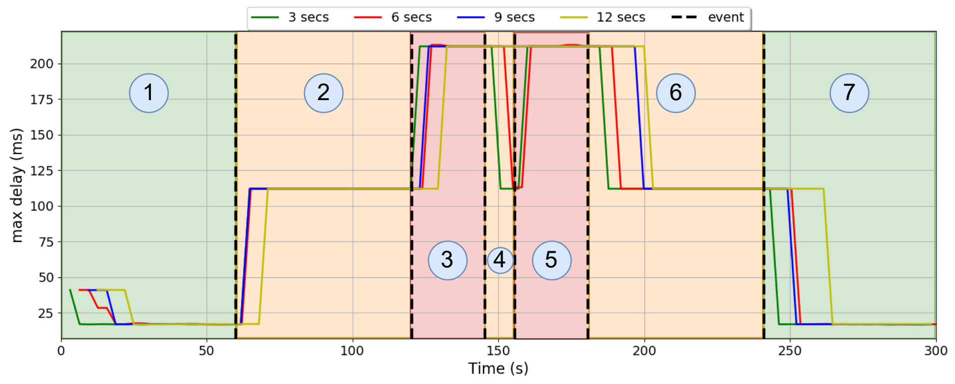


Figure 4.9: Maximum delay - experiment 1 (ICMP)

This difference, that is clearly visible on figure 4.9 is the presence of higher maximum delays in the case of the first measurement. This due to the facts that the docker entities does not know MAC addresses of their neighbors and must then send Address Resolution Protocol (ARP) requests to resolve the destination IP addresses to a destination MAC addresses in order to properly forward the packet. The reason this does not affect OWAMP is that these ARP requests occur at the time of the TCP connection setup (OWAMP-Control) and therefore do not create any delay for the test packets.

4.3.2 Experiment 2 : Duplicates and reordering

The second experiment consists in checking other OWAMP metrics to make sure they are working properly. This involves checking that the DxAgent is able to determine when duplicate or reordering is taking place.

Presentation

The scenario which is then set up is the one presented on the figure 4.10 below:

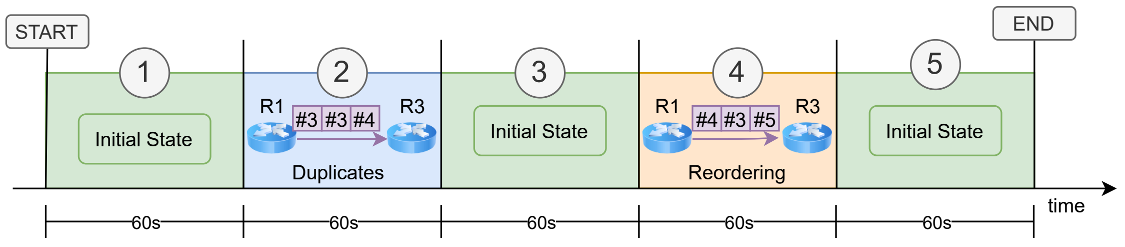


Figure 4.10: Scenario 2

States 1, 3 and 5 simply correspond to the initial case (no reordering or duplicate), they serve as base cases. One minute after the start of the experiment, every second packet from R1 is duplicated, this is state 2. Moreover, after the third minute, every second packet is delayed by 100 ms in order to create reordering. It should be noted that these modifications in the behavior of R1 were once again performed with the tc qdisc tool combined with netem.

The DxAgent rules that will be used are the following:

```
Owamp Duplication : to_pkts_dup>0
Owamp Reordering : to_reordering>0
```

These two rules ensure that if at least one packet is duplicated or reordered, the corresponding symptom will be reported by the DxAgent. It is noted that, R1 being on the path from the server to the border router, it is the one-way duplicate and reordering of this direction that is observed.

Results

Once the experience has been performed and the timestamp of the symptoms retrieved, they can be displayed in a graph which is presented in figure 4.11 below.

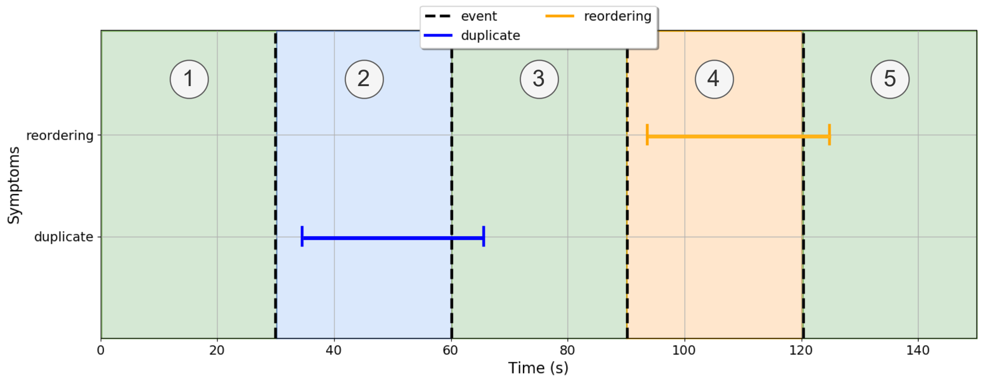


Figure 4.11: Symptoms - experiment 2 (OWAMP)

The purpose of the experiment was to demonstrate that the DxAgent enhanced with OWAMP input is indeed able to determine when reordering or duplication occurs. The graph above shows the expected behavior, although again a slight delay in problem discovery and resolution occurs.

4.3.3 Experiment 3 : Link failure discovery

The third experiment aims to demonstrate that we can use the new DxAgent inputs to find out if links are dead. Since we have two sources, it is possible to verify that a link is dead in two different ways. Multiplying different sources allows us to better define the reason of a problem and we will see why.

Presentation

The scenario that will be analyzed, putting to the test both pings is the following:

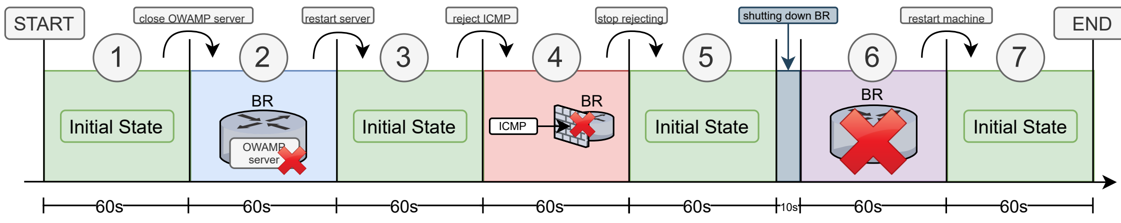


Figure 4.12: Scenario 3

As before, states 1, 3, 5 and 7 are intended to serve as a reference state between the various anomalies that we will introduce into the network. The first link failure (state 2) is simply the shutdown of the OWAMP server located on the Border Router before being turned on again 60 seconds later. The second anomaly (state 4) is to test the ICMP ping for 60 seconds by simply blocking the `echo-request` packets at the firewall. The last anomaly (state 6) is to shut down the container representing the border router. Shutting it down takes some time (close to 10s); the real time has been calculated and is visible on the graph 4.13.

The DxAgent rules used for this scenario are rather explicit, they determine when a ping fails to reach the target machine:

```
Owamp - not reachable : owamp_accessible=="no"
Icmp - not reachable : icmp_accessible=="no"
```

Results

By running this test on the docker topology and putting on a graph the recovered information, we obtain the following:

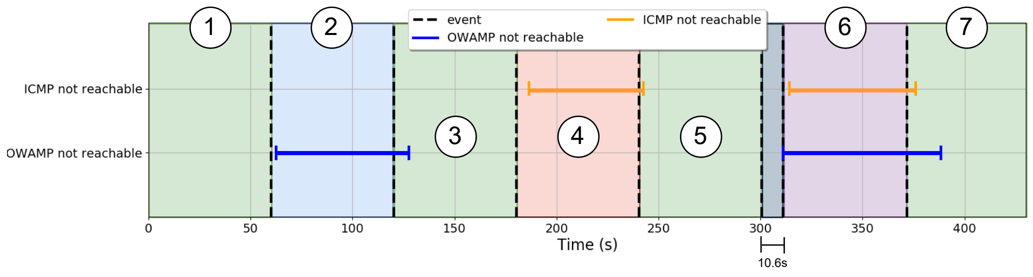


Figure 4.13: Symptoms - experiment 3 (OWAMP and ICMP)

As expected, when the OWAMP server is off, OWAMP pings do not reach their destination while ICMP pings do not encounter any problems. But when the echo-request is blocked, it is the opposite. Finally, the most interesting case to analyze is when the router is turned off.

First of all and fortunately, neither of the two pings reach their destination. This result is interesting because by multiplying the sources we can establish a better vision of the network that we want to diagnose. Indeed, we have shown that, for example, cutting the OWAMP server is enough to make the machine inaccessible for OWAMP. However, it is still accessible, so having a second source is a good idea. We could then improve our set of rules in this way:

```
Owamp - not reachable : owamp_accessible=="no" and icmp_accessible=="yes"
Icmp - not reachable : icmp_accessible=="no" and owamp_accessible=="yes"
Machine not reachable : owamp_accessible=="no" and icmp_accessible=="no"
```

Of course, the fact that the machine is not accessible for both protocols does not necessarily mean that the machine is off, but we are getting closer to the truth than with only one source. Moreover, we notice that OWAMP observe more quickly that the machine has been turned off and takes more time to realize that it has started up compared to traditional pings. My intuition is that the OWAMP server should be shut down before the machine is turned off (due to the shut down process) and that the server should turn back on some time after the machine is turned on.

4.3.4 Experiment 4 : The Interest of one-way metrics

The purpose of the last experiment is to show the interest of OWAMP and the reason why it was chosen in priority. By comparing the OWAMP results of this fourth scenario with those of the traditional ping, we will show the value of retrieving one-way metrics.

Presentation

The figure below shows the scenario that was used for the fourth and final experiment.

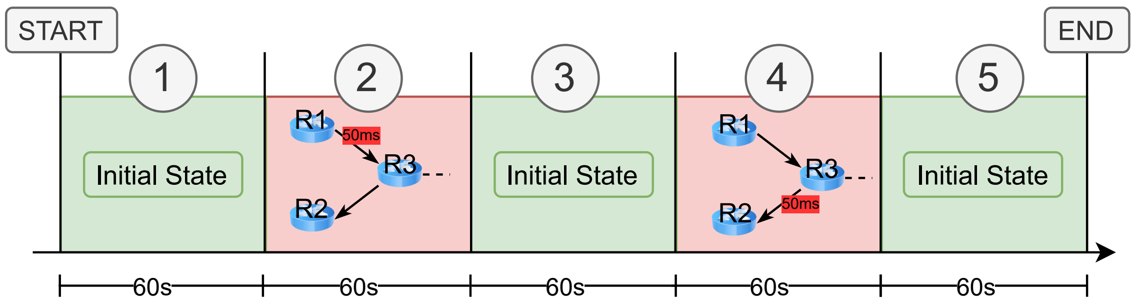


Figure 4.14: Scenario 4

This scenario takes advantage of the asymmetric topology, which was actually made that way for this reason, to add delay in one direction of the path and then in the other. The figure 4.15 provided below shows the docker topology in a more fundamental way than the one already presented. We can easily see that the packets leaving from R1 belong to the path starting at the server and arriving at the border router, let's call this path the **to** path. While those leaving from R2 belong to the return path, let's call this path **from**.

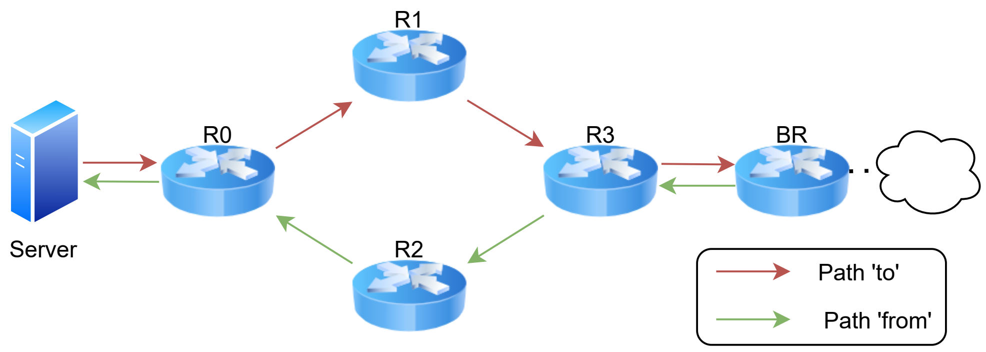


Figure 4.15: Simplified topology - The two path

The DxAgent rules has been chosen to simulate a possible requirement of a streaming service provider:

```
Owamp - not reachable : to_ow_del_med>50
Icmp - not reachable : avg_rtt>50
```

Let's imagine that the network operator of this streaming company knows that the most important thing for his network is that the packets sent from the server to the client are sent in a minimum of time but the client's acknowledgments may arrive with a greater delay. In this case, he could for example determine the first rule above. This one makes sure that the path **to** does not exceed a median of 50ms. However, for the path **from**, he could put no rule or put a rule with a larger maximum delay (example: 100 ms). This possibility is only available because OWAMP allows one-way metrics. Besides, we have also added a similar ICMP rule (except that it is the average RTT). In this case it is not possible to choose a delay requirement for a specific path and this is precisely what we want to demonstrate.

Results

This time, in addition to displaying the times when symptoms were discovered by the DxAgent, the graphic presneted in figure 4.16 displays the delay metrics. For OWAMP this is the median delay and for ICMP the average delay. Since the OWAMP implementation does not calculate the average delay, we will use the median delay, which is similar to the average delay in this case since delays only evolve in fixed stages.

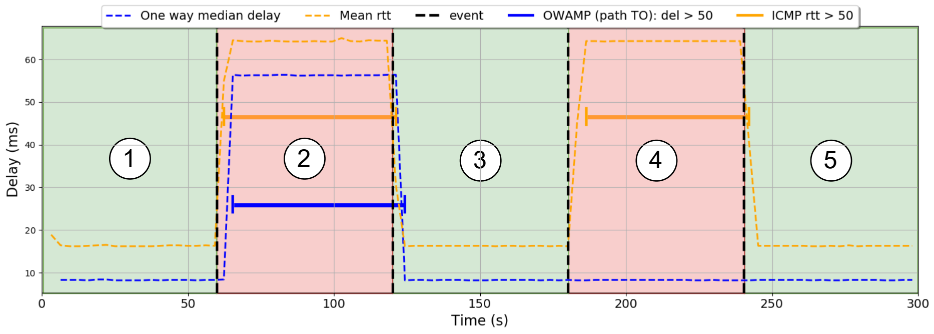


Figure 4.16: Delay and Symptoms - experiment 4 (OWAMP and ICMP)

The first thing we can compare is the difference between the dashed curves. Since the traditional ping is only able to calculate the RTT and not the one-way delay, its curve always shows a delay 8 ms (delay due to the **from** path) greater than the OWAMP curve for the initial states. But the most interesting thing to observe is the difference between these two curves at state 4, i.e. the moment when the delay on the **from** path has been increased by 48 ms. We notice that the OWAMP curve has not been influenced because it calculates the **to** path. This is exactly the result we wanted to obtain. This shows that the choice of one-way metrics for the DxAgent is very good if one needs different requirements depending on the direction of the traffic. With a probing tool that can only compute RTTs (like the traditional ping), a symptom is discovered by the DxAgent regardless of the direction of the path on which a network event occurs. This can be seen on the graphic (figure 4.16) where we notice that only the rule related to ICMP has been validated. Moreover, we can recall that delay is not the only metric to benefit from the "one-way" interest, there are also all the other OWAMP metrics: dupliactes, rordering, etc.

Conclusion

The objective of this master thesis was to extend the angle of view of DxAgent by adding active network measurements. We can say that this goal has been achieved.

First, an API for the OWAMP protocol had to be built in python and it is functional and publicly available to provide a parameterizable one-way metric tool. It was not an easy task as the implementation of this protocol is absurdly complex to use and poorly documented. Moreover, interfacing several programs through a scheduler managing a thread pool does not facilitate the implementation of this interface. This newly developed API has allowed the integration of new estimable metrics in the DxAgent. On the other hand, an interface to start ICMP pings with several machines with a frequent interval has also been developed.

This step of integrating the active network metrics into DxAgent also proved to be rather complicated since it was necessary to understand and dissect all the pre-existing code before imagining any improvement. Several challenges had to be overcome, such as how to trigger measurement tests and synchronize the results with the DxAgent scheduled update. In addition, a number of cases had to be handled to allow a high degree of flexibility in the combined use of OWAMP and DxAgent. The result is that it is now possible to retrieve metrics with a constant frequency from several machines in the network and to analyze potential anomalies.

Then, in order to show the efficiency, the functioning and the usefulness of these new measures, a use case was imagined. By imagining a small topology (of 6 machines anyway), we wanted to show the usefulness of the new additions. The implementation of these entities and the way to connect them proved to be a real challenge. We had to learn the concepts of Docker but also its inter-container communication system. But once the script allowing to automate scenarios was done, experiments could be done.

We could first show that the DxAgent is able to retrieve the delay of the studied path but also to display the right symptoms at the right time. It was also the occasion to see the impact of the different parameters of OWAMP and we noticed that they have a very minimal impact on the experiments. Then two other experiments allowed us to demonstrate that DxAgent is able to ensure some network properties of a path. The detection of duplicates and reordering proved to be fully functional and are therefore good additions to the detection agent. Furthermore, link failure detection was shown and we noticed how the use of two input sources (OWAMP, ICMP) can be very useful to detect a more precise symptom. Finally, the last experiment was aimed at showing the interest of one-way measurements using an asymmetric delay between the server and the target machine. This was an opportunity to show a concrete case where OWAMP can be valuable.

Before, this agent was very useful to detect symptoms specific to the studied machine. Now, it is armed with many other metrics related to network connections. Therefore, it will be even better adapted to a network based service. Of course this agent is, for now, only able to notify the network operator of symptoms of its service without automatic corrections. But it is not difficult to think that it could very well be integrated into a more autonomous system. Thanks to its new metrics, DxAgent is even closer to its initial goal of detecting service anomalies in an Intent-based Networking context.

Intent-based Networking and service assurance, in general, is in full evolution. Its operation makes it a very powerful and practical tool for a network operator. And I can't wait to see how these systems will develop and how they could revolutionize network architectures. I am glad that this thesis allowed me to learn about this subject and to have integrated a project highlighting it.

Bibliography

Sources

[1] *apscheduler.readthedocs.io - Advanced Python Scheduler Documentation.* https://apscheduler.readthedocs.io/en/stable/.

[2] *apscheduler.readthedocs.io - User Guide - Basic Concepts.* https://apscheduler.readthedocs.io/en/stable/userguide.html#basic-concepts.

[3] *cisco.com - CISCO Intent-Based Networking.* https://www.cisco.com/c/en/us/solutions/intent-based-networking.html.

[4] *docs.docker.com - docker network bridge.* https://docs.docker.com/network/bridge/.

[5] *docs.docker.com - Overview of Docker-Compose.* https://docs.docker.com/compose/.

[6] *docs.python.org - Subprocess Documentation.* https://docs.python.org/fr/3/library/subprocess.html.

[7] *github.com - Edeline Korian - dxagent.* https://github.com/ekorian/dxagent.

[8] *github.com - OWAMP.* https://github.com/perfsonar/owamp.

[9] *gminsights.com - Intent-based Networking Market.* https://www.gminsights.com/industry-analysis/intent-based-networking-ibn-market.

[10] *ietf.org - A One-way Active Measurement Protocol (OWAMP).* https://tools.ietf.org/html/rfc4656.

[11] *ietf.org - Intent-Based Networking - Concepts and Definitions.* https://tools.ietf.org/html/draft-irtf-nmrg-ibn-concepts-definitions-03.

[12] *ietf.org - RFC7575 - Autonomic Networking: Definitions and Design Goals.* https://tools.ietf.org/html/rfc7575.

[13] *ietf.org - Service Assurance for Intent-based Networking Architecture.* https://tools.ietf.org/html/draft-claise-opsawg-service-assurance-architecture-04.

[14] *lemagit.fr - definition IBN.* https://www.techzine.eu/blogs/infrastructure/43092/cisco-needs-time-to-make-intent-based-networking-successful/.

[15] *man7.org - close manual.* https://man7.org/linux/man-pages/man2/close.2.html.

[16] *man7.org - open manual.* https://man7.org/linux/man-pages/man2/openat.2.html.

[17] *man7.org - rename manual.* https://man7.org/linux/man-pages/man2/rename.2.html.

[18] *man7.org - tc qdisc man page.* https://man7.org/linux/man-pages/man8/tc.8.html.

[19] *marketsandmarkets.com - Cloud Monitoring Market by Component, Service Model, Organization Size, Industry, and Region - Global Forecast to 2022.* https://www.marketsandmarkets.com/Market-Reports/cloud-monitoring-market-252477280.html.

[20] *software.internet2.edu - One-Way Ping (OWAMP).* https://software.internet2.edu/owamp/index.html.

[21] *techzine.eu - CISCO needs time to make Intent-based Networking succesful.* https://www.techzine.eu/blogs/infrastructure/43092/cisco-needs-time-to-make-intent-based-networking-successful/.

[22] *wiki.linuxfoundation.org - netem.* https://wiki.linuxfoundation.org/networking/netem.

Suggested readings

[23] *datatracker.ietf.org - IP Performance Measurement (ippm).* https://datatracker.ietf.org/wg/ippm/about/.

[24] *docker.com - what is a container.* https://www.docker.com/resources/what-container.

[25] *ietf.org - A One-way Delay Metric for IPPM.* https://tools.ietf.org/html/rfc2679.

[26] *ietf.org - A One-way Packet Loss Metric for IPPM.* https://tools.ietf.org/html/rfc2680.

[27] *internet2.edu - Internet2 Community.* https://internet2.edu/.

[28] *jinja.palletsprojects.com - jinja.* https://jinja.palletsprojects.com/en/3.0.x/.

[29] *perfsonar.net - Perfsonar.* https://www.perfsonar.net/.

[30] *readthedocs.io - python-atomicwrites.* https://python-atomicwrites.readthedocs.io/en/latest/.

[31] *wikipedia.org - Ethtool.* https://en.wikipedia.org/wiki/Ethtool.

[32] *Wikipedia.org - Policy-based Management.* https://en.wikipedia.org/wiki/Policy-based_management.

[33] *Wikipedia.org - Software-defined Networking.* https://fr.wikipedia.org/wiki/Software-defined_networking.

[34] *wikipedia.org - YAML.* https://fr.wikipedia.org/wiki/YAML.