Alma Mater Studiorum · Università di Bologna

SCHOOL OF ENGINEERING AND ARCHITECTURE

Department of Electrical, Electronic, and Information Engineering "Guglielmo Marconi" - DEI

Second Cycle Degree in TELECOMMUNICATIONS ENGINEERING

Degree Thesis
in
LABORATORY OF NETWORKING M

DYNAMIC RESOURCE AND SERVICE DISCOVERY IN FOG COMPUTING

Supervisor:
Prof. Walter Cerroni

Candidate:
Mario Valieri

Co-supervisors:
Prof. Daniele Tarchi
Dott. Gianluca Davoli

Session III
Academic Year 2019/2020

To those who have always supported me,
in every possible way.

Contents

Abstract **3**

1 Introduction **4**

2 Service Orchestration in Fog Computing Scenarios **6**
2.1 Network nodes and service types 6
2.2 System Architecture 8
2.3 Services: definition and discovery 10
2.4 FORCH core: Zabbix 11

3 Dynamic Services and Node Discovery Protocols **13**
3.1 Service discovery protocols overview 13
3.1.1 Jini 14
3.1.2 XMPP 15
3.1.3 UPnP 15
3.1.4 WS-Discovery 16
3.1.5 DNS-SD 16
3.1.6 SSDP 16
3.1.7 SLP 17
3.2 SLP overview 18
3.2.1 Service definition 19
3.2.2 Operating mode 20
3.2.3 Message types 22

4 SLP Development **25**
4.1 General SLP APIs structure 25
4.2 Initial development strategy 27
4.2.1 PySLP library 27
4.2.2 PySLP testing 30
4.2.3 PySLP limitation and issues 57
4.3 Adopted solution 59
4.3.1 RFC 2614 compliant libraries 59
4.3.2 OpenSLP library 60

4.3.3 OpenSLP: Python 3 integration 62

5 SLP Integration in FORCH 65

5.1 Service-centric abstraction entities 66

5.1.1 Description of fo_service.py 67

5.2 OpenSLP wrapper 73

5.2.1 Description of fo_slp.py 73

5.3 DA cache abstraction 80

5.3.1 Description of fo_servicecache.py 81

6 Performance Evaluation 83

6.1 Service Search Tests 84

6.2 Refresh Metrics Test 88

7 Conclusion 91

7.1 Future improvements 92

Bibliography 93

Abstract

The cloud computing is already a well-known paradigm, known and used in both business and consumers areas. It provides a lot of advantages, but today the necessity of data security and low latency is rapidly increasing. Nowadays, in next generation networks (NGN), the fog computing paradigm is able to satisfy strict latency and security requirements using distributed computational power. In a dynamic fog orchestration scenario, the discovery of available resources and services is a fundamental aspect to achieve a good quality system.
This thesis is focused on the study and comparison of service and node discovery techniques and protocols, with the aim of finding an optimal solution to the problem related to a preexisting fog service orchestration system. Once the optimal solution is detected, a working implementation is proposed and evaluated with appropriate experimental measurements, particularly relevant for network administration.

Chapter 1

Introduction

The concept of an artificial machine that can help the human to compute a mathematical expression or solve mathematical problems exists since the ancient Greece. In the centuries this idea has evolved following the science evolution turning a mechanic mathematical tool like an abacus into an electric calculator known as computer. Of course, an abacus and a computer are two completely different things, but during the whole story of humanity before internet there was a clear common notion: bigger is the problem that you have to deal with, more powerful is the tool that you have to own.
This sentence is no more true at one hundred percent because every day we perform challenging mathematical computations using our smartphones, which are devices with limited computational capabilities.
When we ask to Google Maps the better path to reach a point, the optimization problem which must be solved to answer is huge and the smartphone is not able to solve it in a reasonable time. The solution is internet or better cloud computing. Sending our problems to a supercomputer, cloud computing allows us to handle problems which normally cannot be solved by our smartphone (i.e. the tool).
The evolution of the calculator went directly from local PCs to the cloud, skipping possible intermediate points. Now these intermediate solutions are more required because of the two main problems of cloud computing: latency and security.
Today's applications are supposed to be very reactive, more and more in real time; for instance, let us think to either the increasing number of First Person View (FPV) applications or the real time video transcoding services. Since this kind of services generates a big amount of data to be processed very fast, it would be better to reduce the transmission/reception latency as much as possible and this is not very simple if we use cloud servers which can be everywhere in the world. If a service which offers nearer servers would be available, it would be certainly better.
Another very important aspect is digital security. The cloud is very useful, but since internet is a very big and populated place, in general, higher is the number of network nodes in which our data travel, lower is the security. As before, nearer servers can provide

higher security since the number of used routers should be low. This kind of distributed computing paradigm already exists and is called fog computing.
As the name suggests, fog is something between local and cloud and is an intermediate step that allows to use nearer servers for the computational demand. Due to its nature, fog computing allows to reduce latency and increase data security, in fact data can potentially never go to internet. This is possible for instance, exploiting the computational power present in the base stations always connected with our smartphones [1]. While in the past years base stations were composed of dedicated hardware, today thank to the new Software Defined Radio model, they use programmable devices with computational power which can be potentially used also by someone who needs it. In this way, people could benefit of non-local services without sending data somewhere in the cloud. Of course, if it is necessary, in future base stations will contain dedicated PCs exclusively used for the provider fog network.
Fog computing is an evolving concept, and presents a number of management challenges. A recent solution for service orchestration was introduced with FORCH [2], a centralized orchestrator for dynamic service deployment over fog domains. Although a basic implementation of FORCH exists, many of its features need to be reworked, including the service and node discovery process, for which a dynamic and scalable implementation would dramatically improve the performance of the entire orchestrator.
This thesis aims to describe and implement a possible solution to introduce a dynamic discovery of the network nodes and services, proposing a final working solution to be integrated with FORCH project.
Due to its final integration, FORCH will be briefly described in chapter 2. In chapter 3 it will be discussed the most common service discovery protocols suitable to be implemented. Chapter 4 will report the whole development process of the chosen protocol, from the first unsuccessful attempts to the adopted solution. The implementation part will be covered by chapter 5, which will show the integration of the developed code in FORCH. Chapter 6 and 7 will conclude this thesis treating respectively the final results and the conclusion.

Chapter 2

Service Orchestration in Fog Computing Scenarios

In the introduction chapter, it has been mentioned a centralized dynamic service deployment system with a main orchestrating entity [2].
FORCH is a Python open source project which introduces a way to manage and utilize a dynamic fog network following the concept of "Everything-as-a-Service" ($XaaS$). The central node of the node of the network is called orchestrator and while from one side it provides a whole set of management functionalities like monitoring and scheduling of the resources, from the user side it implements a basic REST API which allows an easy access to every available function.

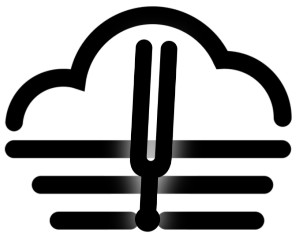


Figure 2.1: FORCH logo. It associates the tuning fork with a fog which becomes a cloud, in order to represent a sort of foggy-cloud orchestrator.

2.1 Network nodes and service types

The orchestrator is projected to discover and interface itself with three types of services:

- $SaaS$: Software-as-a-Service.

- *PaaS*: Platform-as-a-Service.

- *IaaS*: Infrastructure-as-a-Service.

These three types of services together are part of the Everything-as-a-Service model. The first type of service enables the user to utilize a non-local software which is remotely provided. At cloud level this is widely used in applications such as Office365, Gmail and other online mail providers.
The second one provides an environment where are available both software/development tools and operative systems. PaaS is usually used either as framework adopted by developers or analysis and business intelligence tools. In general, it is provided giving the control of one container running on a cloud server. An example of Paas is Microsoft Azure.
The last service type supplies not only a container but also a larger infrastructure which, for instance, can be used to store data, network security or additional computational power. The possibility to execute an heavy code on an external Amazon supercomputer can be considered as an example of IaaS. Three types of services are respectively three

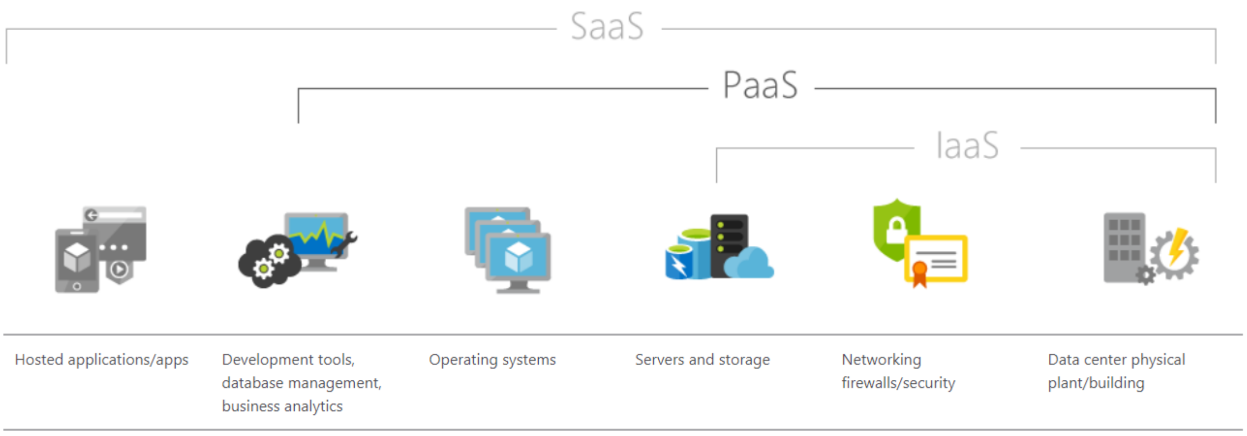

SaaS
PaaS
IaaS
Hosted applications/apps
Development tools, database management, business analytics
Operating systems
Servers and storage
Networking firewalls/security
Data center physical plant/building

Figure 2.2: Differences between service types. Source: [3]

types of entities which FORCH can allocate:

- **Applications (APPs)**: SaaS case. They are supposed either to take input values and return result after some kind of processing (like a sort of function or method), or to behave like a web-based application listening and serving clients.

- **Software Development Platforms (SDPs)**: PaaS case. They are supposed to execute code/libraries given by the user; in practice they provide a service of Remote Procedure Call (either compiled or interpreted).
- **Fog Virtualization Engines (FVEs)**: IaaS case. They provide a virtualized space like a container where whatever operation can be performed. Of course, this is the most flexible option.

Previously the word container has been used. This technology is widely used today and consists in a sort of virtual machine where the resources are not reserved a priori. When a virtual machine starts, the reserved RAM is immediately taken and, if it is not available, the virtual machine doesn't turn on. From the host side instead, if additional RAM is required, it cannot use the free available RAM of the running virtual machine because it is reserved until its ending. Containers solve this problem because, like a process, are able to use resources only when are actually needed and the isolation between host and other containers is still guaranteed. Today this type of light virtualization is something very diffused also thank to platforms such as Docker.
Docker is a free application that handles and simplifies the container management from the creation to the deletion of the container. It is very simple and user friendly compared to other containers software tools, and provides an online cloud where all docker users can share their images each others. This is extremely useful because at the same time you have the possibility to make free backups of your work and you can install pre-created images on your brand new containers. Docker offers official images which can be used to create containers with a specified clean operative system, without the need to perform a whole installation.
In this thesis Docker has been used during the implementation phase to test the written code.
Since using container technology it is possible to have different service types on a single node, the orchestrator is designed to manage the single services provided by nodes instead of adopting a node cataloguing.

2.2 System Architecture

Figure 2.3 represents how the orchestrator interacts with the other components. As already mentioned, FORCH is in the middle between users and nodes, providing respec-

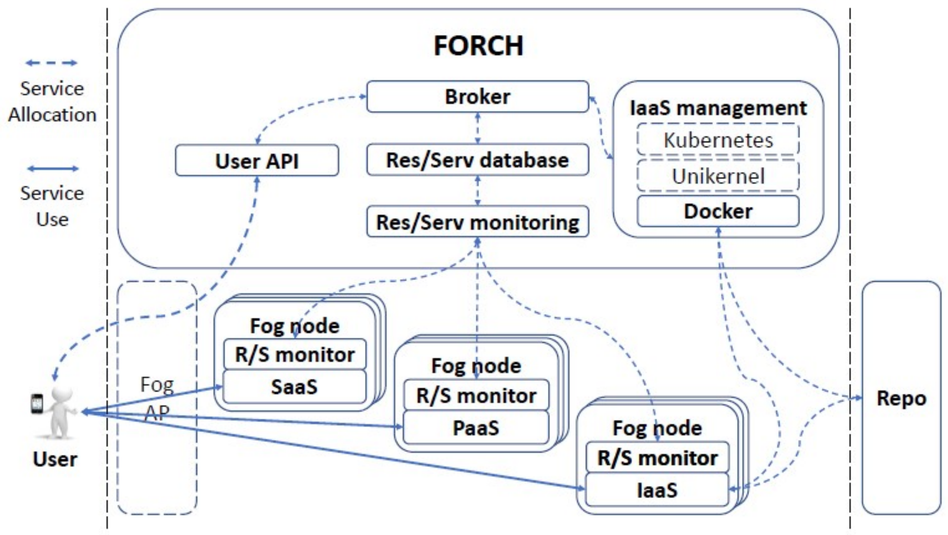


Figure 2.3: FORCH reference architecture.

tively the proper API and services coordination. The API is a standard REST API that allows an easy interfacing with the orchestrator. Each module inside FORCH uses API to communicate, but their REST endpoints are used only internally.
Once explained the API block, the figure shows three additional modules which have not been mentioned yet:

- Broker
- Resource and Service Database
- Resource and Service Monitoring
- IaaS Management

Broker is the FORCH's component which coordinates the other parts. It can be said that the broker is the part of the orchestrator which manages the communication between the different modules inside it.
Res/Serv database is the database where network resources/services are memorized and stored in a sort of cache. Each time that a service is discovered, it is added to this database.
The Resource and Service Monitoring block is committed to manage the monitoring part.
FORCH is a smart element that is able to understand the instantaneous working load

that a node is performing on a certain instant of time. In this way if the user requests a certain service, the orchestrator is able either to tell that the requested resource is currently unavailable or, if the same service is provided by more nodes, to choose which node is more adapt to provide it. This choice is based on the instantaneous computational load of the nodes and this information is provided by the monitor component. Node monitoring is perhaps the most fundamental part which makes FORCH so smart and so useful for the coordination of the fog network.
The last module is able to allocate and deploy services on Iaas nodes using virtualization technologies (e.g. containers). This module offers a whole set of possibilities, in fact in theory it is possible to have whatever service is needed. If for instance an unavailable service is needed, the IaaS management module can download and install a Docker image which provides that service and gives its access to the user. This is a fundamental feature which contributes to the powerfulness of fog networking.

2.3 Services: definition and discovery

Currently, in FORCH services are defined through JSON files which specify: where the service is located, its name and a human readable description of itself. There are three JSON files, one for each service type: *db_apps.json*, *db_sdps.json* and *db_fves.json*.

```json
{
  "apps": {
    "APP001": {
      "thumbnail": "http://137.204.57.73/restooltest/thumb_httpd.jpg",
      "name": "httpd",
      "descr": "Apache web server"
    },
    "APP002": {
      "thumbnail": "http://137.204.57.73/restooltest/thumb_stress.jpg",
      "name": "stress",
      "descr": "Stress host"
    },
    "APP003": {
      "thumbnail": "",
      "name": "ffmpeg",
      "descr": "Multimedia transcoder ffmpeg"
    }
    }
}
```

Code 2.1: Example of JSON service file containing SaaS services definition.

Service location is given using an HTTP URL that contains the static IP address of the node which hosts the service. FORCH is not able to dynamically discover network nodes, hence they must be manually configured by the network administrator. In this way the orchestrator knows a static address which allows it to contact the nodes.
This is the most critical issue of this project; the work described by this thesis tries to solve this problem giving to FORCH the capability to perform a dynamic node discovery. In today's world is not possible to imagine a network based project which is not able to manage scalability by itself, so the dynamic discovery capability is a fundamental feature, mostly for what concerns fog networking.

2.4 FORCH core: Zabbix

So far FORCH has been described in terms of logical architecture and provided features, neglecting its implementation aspects. FORCH is a quite complex project that would require an expert team of software developers if the starting point is a blank source code so, in order to start from something that already is available, it is based on a software platform called Zabbix.
Zabbix is an open source software used for networks, computers (i.e. servers, virtual machines, etc.) and cloud monitoring. In FORCH it is used like a sort of framework which provides the monitoring features required by the orchestrator through its Python API. Since FORCH is built on top of Zabbix, it can be said that it is the core of the project.
Zabbix is an already mature environment which integrates a GUI web page which allows the developer to monitor and configure the whole system. For instance, the static node configuration mentioned in section 2.3 is currently done in the relative configuration web page provided by Zabbix GUI.
Zabbix is so advanced that already provides a way to discover nodes and services, but it is quite limited. The reasons why an external service discovery protocol has been chosen are:

- *Modularity*: if FORCH is composed by as much independent modules as possible, it can be simply changed replacing one of them. Currently the orchestrator is based on Zabbix, but this does not mean that it is the best choice; in the future Zabbix could even be replaced. In such eventuality, an external service discovery protocol would simplify Zabbix substitution.

- *Limited supported services*: Zabbix supports the discovery of a limited list of standard services like FTP, SSH, WEB, POP3, IMAP, TCP, etc. FORCH is supposed to handle whatever service, standard or custom.

- *Security*: Zabbix can discover services only if the received Zabbix agents information are unencrypted. Currently FORCH does not use encryption, but it is already planned as future improvement.

- *Centralization*: even if FORCH is a centralized system, an external centralized protocol which is able to automatically reconfigure itself to work in a distributed way would be preferable in case of failures.

- *Data organization*: Zabbix is organized in a node-centric way. This means that the fundamental entity is a node and everything else is a related attribute or field. A service-centric data organization would be a better abstraction on which construct FORCH. In the latter case, the fundamental entities are the services and everything else is related to them.

Chapter 3

Dynamic Services and Node Discovery Protocols

As already mentioned, the discovery of resources and available services is a fundamental part of any system which aims to provide a dynamic and flexible services deployment.
In order to avoid static nodes configuration there are mainly two options: the development of a brand new dedicated protocol and the use of an already available protocol. Of course, the second option is the best one, because a mature protocol has already solved all the issues which carry out during the development of a discovery protocol.
In this chapter, it will be reported the analysis of available protocols dedicated to service discovery and it will be illustrated the details of the final adopted protocol.
Since a network service cannot be discovered if the host node is unknown, before to continue it is important to underline the fact that the majority of services discovery protocols is designed to perform services and node discovery at the same time.

3.1 Service discovery protocols overview

Looking for this kind of protocol it is immediately clear that they are used at different levels and in different environments during the development of a project. XML files, Java platforms and SOAP based protocols are only few examples of possible basis for protocol foundations.
The main relevant protocols considered during the research are:

- **Jini**: also called Apache River.
- **XMPP**: Extensible Messaging and Presence Protocol.

- **UPnP**: Universal Plug and Play.
- **WS-Discovery**: Web Services Dynamic Discovery.
- **DNS-SD**: DNS-based service discovery
- **SSDP**: Simple Service Discovery Protocol.
- **SLP**: Service Location Protocol.

Now each listed protocol will be briefly described as well as the compatibility with FORCH and project requirements.

3.1.1 Jini

Jini is a network architecture created by Sun Microsystems in November 1998. It was devised to be used for the development of Java distributed systems. Of course it is implemented in Java and is designed to discover and provide Java Remote Method Invocation (RMI) objects. Its architecture is centralized and the services information are stored using a Lookup Service. Network devices can discover services performing both active and passive Lookup Service discovery. It is devised for LAN that can be large like enterprises LAN and is quite secure, in fact it provides: authentication, authorization, confidentiality and integrity.
For what concerns FORCH integration there are two main aspects to be considered:

- It is based in Java and, even if there are methods to use Java code in Python, this should be avoided for simplicity and code readability matters.
- RMI is the acronym used by Java to provide the well-known Remote Procedure Call (RPC) feature. This technique allows a client program to remotely execute a piece of code (i.e. a method), provided by a dedicated server program. Each remote method which can be used by the client must be defined or discovered in the program's code, hence RMI belongs to a too low level of the software stack. FORCH services are supposed to be something more than a single Java method, hence Jini is not even a good candidate to implement the XaaS paradigm.

It is evident that Jini does not fit well with FORCH, hence it was discarded.

3.1.2 XMPP

Extensible Messaging and Presence Protocol is a set of open protocol for instant messaging based on XML. XMPP is not totally devoted to service location, but it provides a general network framework which includes great discovery features. In particular, XMPP is a great alternative because, differently from other protocols, it provides a solid base to discover services which can be both local or across the network. It is hence adapt to be used in those situations where are present obstacles like virtual machines, networks, and firewalls, which are normally present in the cloud environment. XMPP is designed to work on top of TCP packets but, in order to avoid problems with firewall and similar components, it can also adopt an HTTP based transport solution.
XMPP is surely a great option to be used inside this project, but FORCH is supposed to work in a local environment (at least for the moment), so the complexity introduced by XMPP and its XML files can be avoided. Moreover, it has been thought for messaging purposes, hence it would not be properly used in its context. XMPP is near to the application level, hence its implementation would include a lot of unnecessary features and code. In FORCH, a lower service discovery protocol would be sufficient and hence preferable. Finally, the use of such complex protocols can also mean lower performances than lighter and simple solutions.
It is important to put in evidence that local environment hypothesis is not so far from reality. If we imagine a possible fog network provided by a mobile network operator, it is reasonable to suppose that from its point of view its fog network is inside a local network or something similar.

3.1.3 UPnP

Universal Plug and Play is a network protocol designed to simplify the interconnection of devices inside home and enterprise LANs. Thank to this protocol today's computers are able to connect themselves in a totally automated way enabling features such as data sharing, communication and entertainment. UPnP is based on peer-to-peer technology, and utilizes SOAP encapsulated messages to select and use the required service; services are defined using XML files.
UPnP is a complete protocol which entirely manages the stack from the addressing to the presentation; it in fact supports zero-configuration networking (zeroconf). For this reason, the previous XMPP considerations about complexity, out-of-context utilization

and possible low performances are also valid for UPnP, hence it is not the best option to be used inside FORCH.

3.1.4 WS-Discovery

Web Services Dynamic Discovery is a multicast discovery protocol for service localization on a LAN. It uses TCP and UDP packets, multicast addressing and the nodes communicate each others using a web service standard called SOAP-over-UDP. It was born between 2008/2009 by a corporates group and can be used inside zeroconf project.
This protocol seems a good choice for FORCH, but its low diffusion could lead to a lack of documentation. The positive side is that it is available a Python 3 updated library which implements WS-Discovery, so it can be considered the best alternative so far.

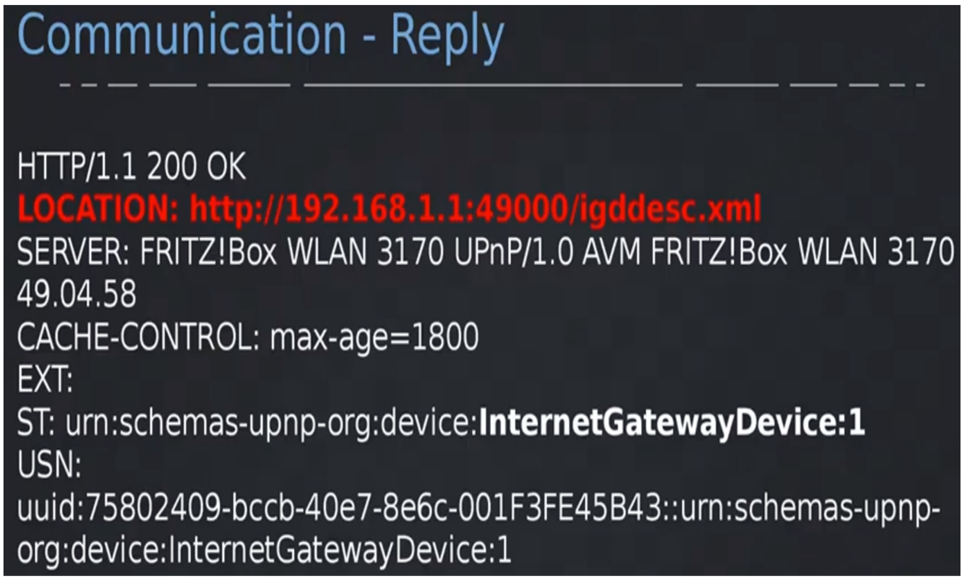
3.1.5 DNS-SD

Before starting to analyse existing protocols, the idea to use mDNS as discovery solution has been carried out. mDNS is a protocol defined inside zeroconf project and allows to have a domain name system without a DNS server. This is possible using multicast packet for the query/reply DNS messages.
Since the real FORCH issue was not the service discovery, but the node discovery, mDNS could be a valid solution which allows a node to scan the network and announce itself correctly. If we define a namespace containing all the possible domain names that nodes can use to announce themselves in the network, a node can understand which domain name to assume and the orchestrator can discover it both actively and passively. This idea is more or less used by the DNS-based service discovery systems.
DNS-SD allows clients to discover network services through DNS queries. DNS-SD can work both with standard unicast DNS server based system and mDNS distributed environments.
This is surely another valid option but, since multicast can potentially generate a lot of network traffic and FORCH tries to avoid to rely on entities like DNS servers, it is better to look for another solution.

3.1.6 SSDP

Simple Service Discovey Protocol is the basis of the UPnP discovery procedure. Considering only this protocol instead of the whole UPnP protocol, it is possible to reduce the

unnecessary complexity of UPnP which was the discard reason of UPnP.
SSDP is a text-based HTTPU (i.e. HTTP over UDP) protocol which uses multicast for services announcements and the services are defined inside an XML file hosted by the network nodes. In practice a node asks for a service to other nodes and they send their XML files containing their own characteristics and capabilities. Even if SSDP is simpler


Communication - Reply

```
HTTP/1.1 200 OK
LOCATION: http://192.168.1.1:49000/igddesc.xml
SERVER: FRITZ!Box WLAN 3170 UPnP/1.0 AVM FRITZ!Box WLAN 3170
49.04.58
CACHE-CONTROL: max-age=1800
EXT:
ST: urn:schemas-upnp-org:device:InternetGatewayDevice:1
USN:
uuid:75802409-bccb-40e7-8e6c-001F3FE45B43::urn:schemas-upnp-
org:device:InternetGatewayDevice:1
```

Figure 3.1: SSDP reply example.

than UPnP, it still presents some drawbacks:

- Since currently services are defined using JSON, XML files introduce a possible deep revision of the current code which, if possible, is better to avoid.
- SSDP has well known security vulnerabilities of DDoS type.
- SSDP is known to generate an high network traffic inside the LAN.

These issues could be considered acceptable but, of course, a better solution would be preferable.

3.1.7 SLP

Service Location Protocol is an open protocol which allows network devices to find services. It is designed to scale from LAN to large enterprise networks and works simultaneously in both centralized (unicast packets) and distributed mode (multicast packets).

It is a IETF standard defined in RFC 2165 [4] (version 1, no more used), RFC 2608 [5] (version 2) and RFC 3224 [6] (version 2 vendor extension) and services are described through a template defined by IANA in RFC 2609 [7]. [6] is the newest update of SLP but, since it is an extension of [5], the latter is still the most used RFC. There is also a standardized API proposal provided by RFC 2614 [8].
SLP has three fundamental entities:

- **Service Agents (SA)**: they are entities that announce (and usually provide) one or more services. They can announce their services with both unicast and multicast packets, depending on the presence of a local service cache called Directory Agent.
- **User Agents (UA)**: they are the nodes which request one or more services. They can send their requests either directly to SAs with multicast packets, or to DAs in unicast (it depends if there is at least one DA).
- **Directory Agents (DA)**: they are the central points of the centralized mode. DAs cache locally the services announced by the service agents and answer to the service requests of the user agents. Both service registration and service request messages are in unicast because both SAs and UAs learn DA address as first thing. Directory agent is an optional entity.

This protocol performs both active and passive search of the network nodes, while the services are always found in active mode. From the security point of view, SLP provides an optional authentication of DAs and SAs and allows to define limited logical scopes where query/reply can be done.
The flexibility of centralized or distributed mode given by this protocol, and the fact that it is an IETF standard, make this protocol a very good candidate to be integrated with FORCH project. Moreover, this standard is already used in a lot of commercial products such as network printers.
Thank also to the availability of PySLP [9, 10], which is a SLP library completely written in Python 3, this is the protocol which seems to fit best in FORCH. Indeed, this was the final choice.

3.2 SLP overview

Since the chosen protocol is the Service Location Protocol, in this section it will be described in detail, giving an overview of its principal and relevant aspects.

3.2.1 Service definition

Before to start to analyse how SLP works, it is important to understand how the services are made, i.e. their structure.
Services are defined in [7] which defines not only the services structure, but also templates which must be used to properly define and describe services. As far as FORCH is concerned, the formalism introduced by service templates is not needed because, at least for the moment, the project is still in a proof-of-concept phase. For this reason service templates will not be described here but, a SLP implementation compliant with the standard, gives the possibility to use them in future.
Services are defined and deployed through URLs, whose structure is composed of:

- **Service Type**: it can be standard or custom and gives an idea of the provided service, constituting a set of services of that typology. Standard services structure is

  service:protocol

  but the general service type syntax is

  service:[descr1_1:descr_2:descr_n]:protocol

  where services descriptors between square brackets are optional and, if present, they create a custom service type.

- **Service URL**: it is a standard URL, hence it contains the address of the node that hosts the service (either literal or its IP) and the location of the service inside the node.

- **Service Attributes**: they are additional fields that characterize the services, for instance the TCP port through which it is possible to reach a service. They are separated by ; character and are written using the notation $key = value$.

Services have also an additional parameter called $lifetime$, which defines how much time a service remains in DAs cache; it goes from 0 to 0xffff (about 18 hours). Lifetime is extremely useful because it avoids that DAs continue to announce no longer available services.

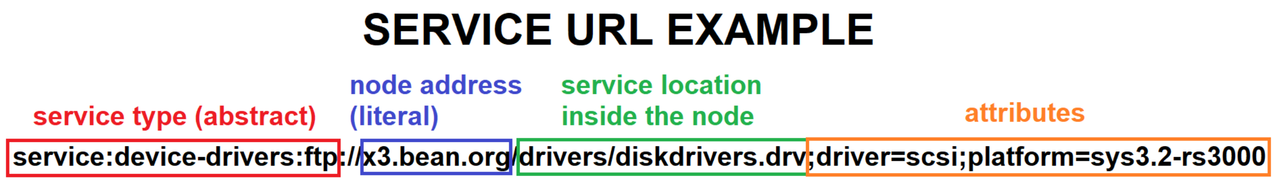

SERVICE URL EXAMPLE
service type (abstract)
node address (literal)
service location inside the node
attributes
service:device-drivers:ftp://x3.bean.org/drivers/diskdrivers.drv;driver=scsi;platform=sys3.2-rs3000

Figure 3.2: SLP service URL example.

3.2.2 Operating mode

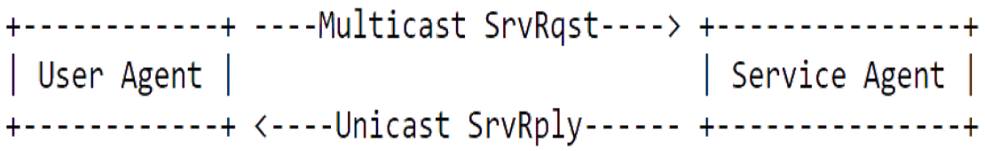
As described in 3.1.7, three main entities defined by SLP are: User Agent (**UA**), Service Agent (**SA**) and directory Agent (**DA**).
These three agents can simultaneously communicate between each other in a distributed and centralized way, employing respectively multicast and unicast packet. As previously anticipated, these two modalities can coexist at the same time exploiting the scope feature defined by SLP.
Scopes are sets of services which create a sort of separated virtual SLP domains. Each SLP agent has its own scope which is "DEFAULT" if it is not explicitly indicated. Each agent must take into account only services, devices and messages belonging to its scope. Since each agent must reply inside its scope, scopes allow to limit network traffic and to administrate and scale SLP networks, which can become bigger in an enterprise context. The three SLP modalities are below described in details.

Distributed mode

In order to utilize SLP in distributed mode, the Directory Agents are not required; they in fact, are utilized as service cache points, which are meaningless in a distributed environment. The active entities are only the User Agents and the Service Agents.
If for instance a UA needs a service, it sends a multicast service request message (*SrvRqst*) to the network and waits for a unicast service reply (*SrvRply*) given by the SAs which know that service.

Centralized mode

SLP centralized operating mode is obtained using Directory Agents as intermediate nodes which store a local service cache.
DAs are discovered either statically or dynamically, in particular [5] says: «UAs can discover DAs using static configuration, DHCP options 78 and 79, or by multicasting (or

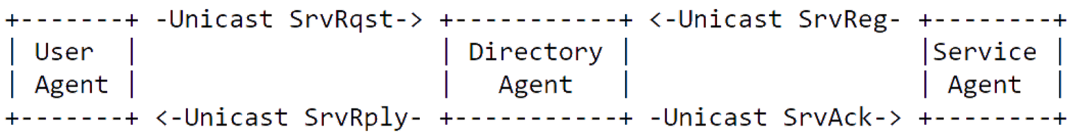


```
+------------+ ----Multicast SrvRqst----> +---------------+
| User Agent |                            | Service Agent |
+------------+ <----Unicast SrvRply------ +---------------+
```

Figure 3.3: Example of direct service discovery between an User Agent and a Service Agent in distributed mode.

broadcasting) Service Requests». In case of dynamic discovery, DAs can be found both in active and passive way.

- Active means that UAs and SAs send a multicast service request with a reserved service type: "service:directory-agent". Once that the DAs receive the request, they respond with a DAAdvert (Directory Agent Advertisement) message, which is received back, allowing to discover the Directory Agents. This procedure is valid also for the discovery of SAs, using the reserved service type: "service:service-agent". This is useful when more than one scope is used.
- Passive means that UAs and SAs do not send a discovery message, but DAs announce themselves sending a DAAdvert. This happens regularly after a given amount of time defined in [5].



```
+-------+ -Unicast SrvRqst-> +-----------+ <-Unicast SrvReg- +--------+
| User  |                    | Directory |                   |Service |
| Agent |                    |   Agent   |                   | Agent  |
+-------+ <-Unicast SrvRply- +-----------+ -Unicast SrvAck-> +--------+
```

Figure 3.4: Example of service discovery between an User Agent and a Service Agent intermediated by a Directory agent because centralized operating mode is utilized.

Simultaneous distributed and centralized modes using scopes

As already anticipated, an SLP UA can discover services through SAs and DAs registered in different scopes at the same time.
In figure 3.5 it can be seen that scope X works as described in distributed paragraph while scope Y works in a centralized way. To achieve this result it is sufficient to register the user agent on both X, Y scopes.

```
+---------+   Multicast  +-----------+   Unicast   +-----------+
| Service | <--SrvRqst-- |   User    | --SrvRqst-> | Directory |
|  Agent  |              |   Agent   |             |   Agent   |
| Scope=X |   Unicast    | Scope=X,Y |   Unicast   |  Scope=Y  |
+---------+ --SrvRply--> +-----------+ <-SrvRply-- +-----------+
```

Figure 3.5: Example of a simultaneous distributed and centralized service discovery exploiting SLP scope feature.

Since FORCH project has a centralized behaviour where the central point is the orchestrator, it makes sense to use SLP in centralized mode to reduce the generated network traffic. From SLP point of view, the orchestrator is simultaneously an user and a directory agent. In case of FORCH project, the orchestrator is the only DA present in the network.

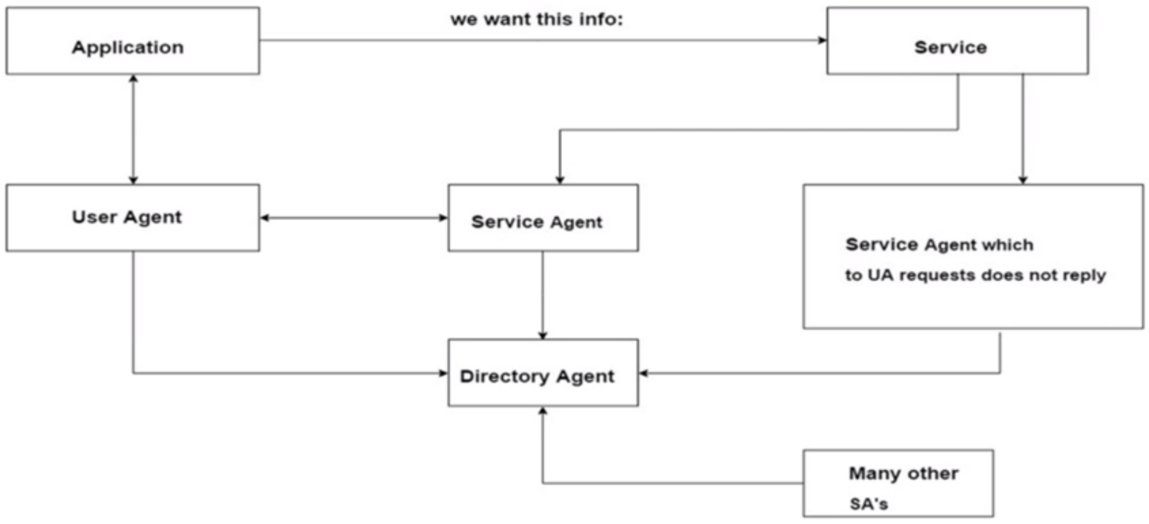


Figure 3.6: General case where which each SLP operating mode is used.

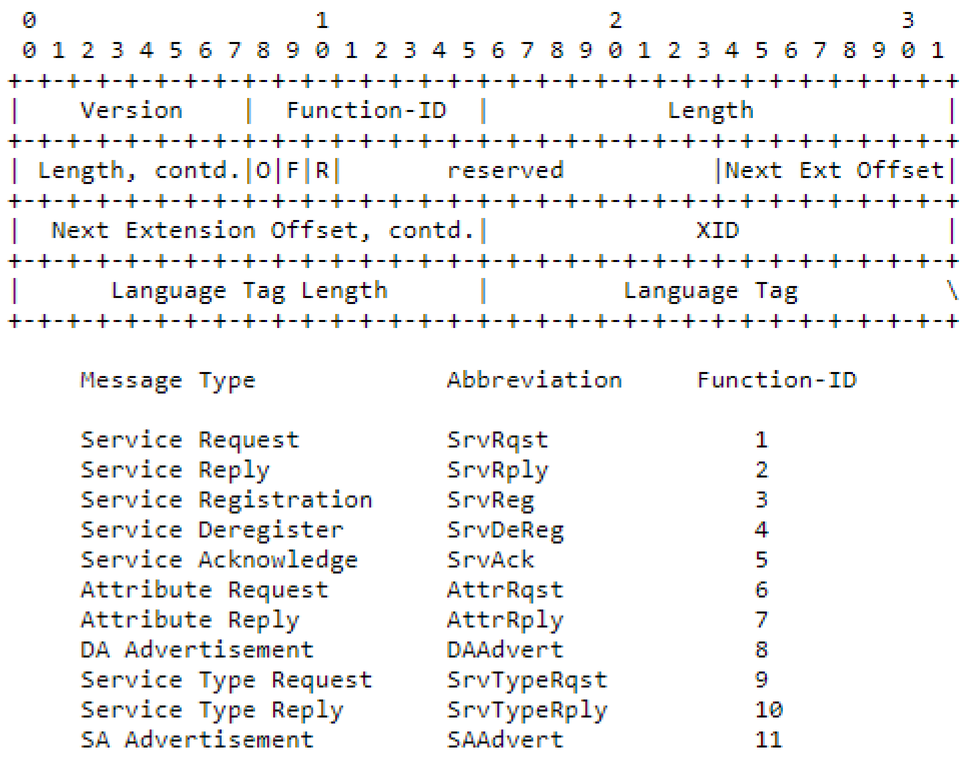
3.2.3 Message types

Currently only few types of SLP messages have been mentioned. Messages are divided in two sets: required and optional. Of course, the second ones are not mandatory messages which are useful to provide additional user features, and an efficient management of service with dynamic attributes.

Afterwards, each message defined in [5] will be reported and briefly explained.

- **Service Request (*SrvReq*, required)**: message used by User Agents to find a service inside the network. SrvReq can be sent either in unicast or in multicast.
- **Service Reply (*SrvRply*, required)**: reply message after a service request. Being an answer to a node, it is always transmitted in unicast.
- **Service Registration (*SrvReg*, required)**: unicast message used in centralized mode by Service Agents to announce their services to Directory Agents.
- **Service Deregistration (*SrvDeReg*, optional)**: message used in centralized mode by SAs to remove a DAs cache service entry.
- **Service Acknowledgment (*SrvAck*, required)**: reply message sent by DA to SA after a SrvReg message.
- **Attribute Request (*AttrRqst*, optional)**: message used by UAs to obtain one or more attributes associated to a service which has been already discovered. According to the chosen operating modality, packets can be either unicast or multicast.
- **Attribute Reply (*AttrRply*, optional)**: unicast reply message of an AttrRqst which contains the required attributes.
- **Directory Agent Advertisement (*DAAdvert*, required)**: message sent by DAs which announces their presence. It can be either unicast if DAAdvert follows a service request whose aim is to discover DAs (active DA discovery), or it can be multicast if DAAdvert is sent without any solicitations (passive DA discovery).
- **Service Type Request (*SrvTypeRqst*, optional)**: message used to discover the service types present in the network. In this way, not only the services can be discovered, but also the type of services provided by the network. Despite this is an optional message, in FORCH it is extremely useful because it allows to implement an extremely flexible fog network which can allocate services that learn at run-time. According to SLP operating mode either unicast or multicast packets can be used.
- **Service Type Reply (*SrvTypeRply*, optional)**: unicast reply message following a SrvTypeRqst.

- **SA Advertisement** (*SAAdvert*, **required**): multicast message used by UAs to discover SAs and the scopes which they support. This kind of message can be sent by UAs only when they can select their own scopes in order to work simultaneously with both DAs and SAs (figure 3.5).



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Version    |  Function-ID  |            Length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Length, contd.|O|F|R|       reserved          |Next Ext Offset|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Next Extension Offset, contd.|              XID              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Language Tag Length      |         Language Tag          \
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Message Type	Abbreviation	Function-ID
Service Request	SrvRqst	1
Service Reply	SrvRply	2
Service Registration	SrvReg	3
Service Deregister	SrvDeReg	4
Service Acknowledge	SrvAck	5
Attribute Request	AttrRqst	6
Attribute Reply	AttrRply	7
DA Advertisement	DAAdvert	8
Service Type Request	SrvTypeRqst	9
Service Type Reply	SrvTypeRply	10
SA Advertisement	SAAdvert	11

Figure 3.7: SLP messages common header and summary of all possible SLP message types.

Chapter 4

SLP Development

When a practical development process starts, it is fundamental to follow the right way of implementation from the beginning. Despite this, many trials are often needed in order to understand which is the better way to proceed. Clearly, the integration of SLP in FORCH is not an exception.
SLP was initially implemented using the already mentioned Python library PySLP but, after an initial testing and wrapping part of this library (which took a quite huge amount of time), it was necessary to add a consistent part of code in order to implement some missing SLP features. Of course this could be done but, since a library development means also to spend a lot of time in debugging and testing, it is better to look for an alternative, if available.
This chapter will illustrate the whole development process performed in order to integrate SLP inside FORCH, including both the successful and the useless trials done.

4.1 General SLP APIs structure

Although different implementations of SLP with several programming languages exists, each of them usually follows (more or less) the indication provided in [8].
Architectural components of SLP (DA, UA, SA, scopes, etc.) are partially implemented in *slpd* and partially into SLP client library.

slpd means Service Location Protocol Daemon and is a background process which acts either as SA or DA server. slpd has a configuration file through which SLP parameters can be set and its main functions are:

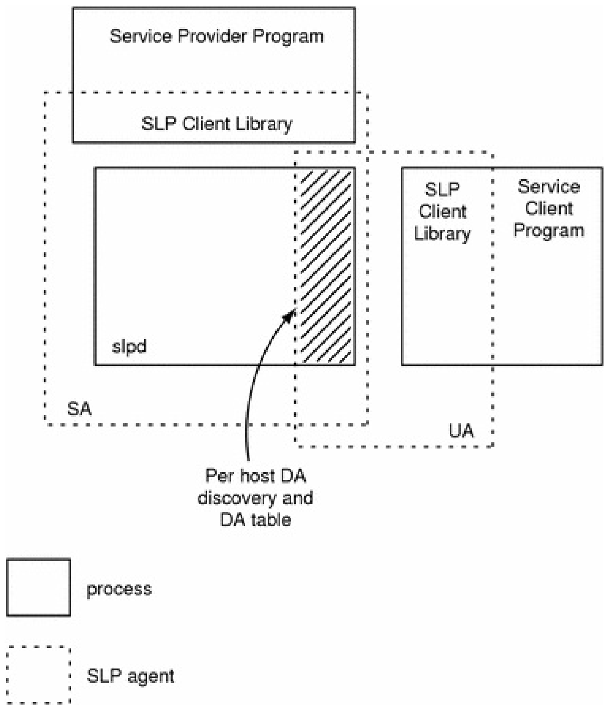


Figure 4.1: SLP API architecture overview.

- Active and passive DA discovery for the UAs and SAs on the local host.
- Management of a table of DAs for the UAs and SAs on the local host.
- DA implementation if slpd is properly configured.

slpd is the entity which accepts service registration request from SAs and keeps them internally. In practice, it implements an internal local cache which memorizes services provided by the local host but, if configured as DA, it announces itself to the network, accepts external service registrations and replies to network service requests. SAs implementation can be though as a process which registers its services to a local, unexposed DA. Hence, slpd can be considered the active agent of SLP implementation.
This API architecture allows to reduce SLP overhead in fact, in case of UA implementation which performs only service request and either does not use DA and scopes or has them statically configured, slpd is not required.
In other words, slpd is only required if a process needs to perform local or external Service Registrations/Deregistrations.

SLP client library is the module that actually implements SLP messages and communicates with and through slpd. It provides all the methods needed to generate, send and receive messages and, as already mentioned, it can be used in a standalone way by UAs which perform only service requests.

The last important aspect of [8] to underline is related to service's lifetime. Lifetime is the already mentioned SLP time period after which an announced service expires. The RFC that describes the protocol [5] doesn't say what an agent should do after this period, because this is an implementation specific aspect. It would be natural to imagine that a SLP library should have a way to perform an automatic services refresh, but this is not the [8] suggested behaviour. The SLP API RFC in fact performs an automatic service refresh only for the services declared with the maximum allowed lifetime period. Every service declared with a lifetime less than the maximum one, hence needs to be manually re-registered.

4.2 Initial development strategy

The most intuitive way to follow when a project has to be implemented, is to search available libraries developed in the same program language chosen for the project. Of course, this was done also at the beginning of this thesis, hence this section will treat the initial attempts in order to integrate SLP inside FORCH.

4.2.1 PySLP library

PySLP (`https://pypi.org/project/pyslp/`, `https://github.com/maksim-v-seliverstov/pyslp`), already introduced in 3.1.7, is an open source library that implements SLP in Python 3 by Seliverstov Maksim. It is partially compliant with the API proposed by [8] and it is the only available Python SLP library, hence it was the easiest starting point. PySLP is quite independent, in fact it uses only internal Python library, relying mainly on three modules:

- **socket**: this is a well known library present in every programming language, which provides the tools to utilize the network functionalities. It gives the possibility for a program to access to the network setting basically the IP, the used protocol and port. Since SLP is a network protocol, its presence was quite obvious.
- **struct**: this library allows to convert C struct data in Python data and vice-versa.

It is useful to manage the byte read from a file or, in this case, incoming from the network.

- **asyncio**: this is maybe the most utilized module inside PySLP. It is a high level library which allows to write concurrent code, hugely used for a simultaneous listening and sending of SLP packets over the network.

PySLP is composed of various Python source files, one for each basic module needed by the library. Afterwards a brief overview of each of them will be given.

- **creator.py**: this file contains the methods which create the network packets of the messages, according to SLP requirements. The implemented methods are:
  - *create_header*: generation of SLP packet header.
  - *create_acknowledge*: generation of SLP ACK packets.
  - *create_url_entry*: generation of SLP service URLs.
  - *create_reply*: generation of Service Reply messages.
  - *create_registration*: generation of Service Registration messages.
  - *create_request*: generation of Service Request messages.
  - *create_attr_request*: generation of Attribute Request messages.
  - *create_attr_reply*: generation of Attribute Reply messages.
  - *create_deregistration*: generation of Service Deregistration messages.
- **multicast.py**: this file contains only two methods which implement the transmission of multicast packets. The contained methods are:
  - *create_listener*.
  - *create_sender*.
- **parse.py**: this file contains methods and utilities which parse the incoming SLP packets. The relevant methods contained are:
  - *convert_to_int*: conversion of the incoming packet data from bytes to integers.
  - *_parse*: core function which actually implements the parsing. The majority of the other parsing methods are a sort of wrap of this one.

  - *parse_header*: parsing of SLP packet header.
  - *parse_url_entry*: parsing of SLP service URLs.
  - *parse_registration*: parsing of Service Registration messages.
  - *parse_request*: parsing of Service Request messages.
  - *parse_reply*: parsing of Service Reply messages.
  - *parse_acknowledge*: parsing of SLP ACK messages.
  - *parse_attr_request*: parsing of Attribute Request messages.
  - *parse_attr_reply*: parsing of Attribute Reply messages.
  - *parse_deregistration*: parsing of Service Deregistration messages.

- **slpd.py**: this file implements the SLP daemon described in section 4.1, hence DAs can avoid to start it. This is the first source code that can run either in a standalone way using an independent shell (defining its own main), or inside the library using the method

  - *create_slpd.*

  The other methods implemented by slpd.py are used in the internal mechanisms of the library, hence they will not be reported because of their low relevance.

- **slptool.py**: this is the file which actually allows to use PySLP. It contains a class called SLPClient which actually implements all the methods to discover SLP services and their attributes.
  Slptool.py contains also code which is not inside SLPClient class but, since this code is used internally by the library, only the methods of SLPClient which are relevant for a library user will be listed. The untreated code (mostly contained in a class called Receiver), is the one which handles internal SLP errors and incoming packets, understanding which type of SLP message has been received.

  - *send*: this is the highest level method which sends the data packets prepared by the subsequent methods.
  - *register*: this method is responsible to register services in the network. It takes in input *servicetype*, *serviceurl*, *attributelist* and $lifetime$ parameters and generates a multicast packet which announces the service. In practice this

is the method used to send Service Registration messages. Looking [5], *register* must be called only if a DA is present, because it is supposed to register the SA services into the DA's cache. Actually this method can be utilized also to let SA's slpd know which services it has to announce. However this is a practical use contained in [8], and it is supposed to generate only loopback traffic. Multicast packets should not be generated in this case.

  - *deregister*: it generates Service Deregistration messages.
  - *findsrvs*: method used to find the service inside a network. In practice it implements the Service Request messages.
  - *findattrs*: method used to find the attributes of a given service present inside a network. In practice, it implements the Attribute Request messages.

- **utils.py**: this file contains the utilities methods used inside the library. Actually it contains only the method *get_lst* which returns a list containing the item passed in input; if the input parameter is already a list it is directly returned.

Of course, in the previous methods lists, the Python language defined methods (constructor, destructor, etc.) have been neglected.

4.2.2 PySLP testing

In order to see if PySLP is a good choice to integrate SLP in FORCH, some steps have been performed:

1. Implementation of PySLP UA and SA running both on local host.
2. Implementation of PySLP UA and SA running on two different containers.
3. Wrapping of PySLP with class which could be used for the integration.
4. Use of the new classes for a simulation with four different containers: one coordinator and three services nodes.

First step was only a simple test to see if PySLP really worked. Thanks to the creator of the library, it required a very small coding effort. He in fact included in slpd.py and slptool.py some testing code which allows the two files to be directly executed in different independent shells. Afterwards, the main code inside the two mentioned Python files is presented below:

```python
if __name__ == '__main__':
    # init library
    loop = asyncio.get_event_loop()
    ip_addrs = '127.0.0.1'
    if ip_addrs is None:
        raise Exception('You should set ip address')
    slp_client = SLPClient(ip_addrs=ip_addrs)

    # prepare example services for their registration
    service_type = 'service:test'
    cnt = -1
    for url in ['service:test://test.com', 'service:test://test_1.com']:
        cnt = cnt + 1
        loop.run_until_complete(
            slp_client.register(
                service_type=service_type,
                lifetime=15,
                url=url,
                attr_list=chr(ord('A')+cnt) + '=' + str(1+cnt)
            )
        )
        print('{} - service is registered successfully'.format(url))

    # find and print registered services given the service types
    url_entries = loop.run_until_complete(
        slp_client.findsrvs(service_type=service_type)
    )
    print(url_entries)
    print('findsrvs for {} - {}'.format(service_type, url_entries))

    # find and print the attributes of the found services
    for _url_entries in url_entries[0]:
        for url in _url_entries:
            attr_list = loop.run_until_complete(
                slp_client.findattrs(url=url)
            )
            print('findattrs for {} - {}'.format(url, attr_list))

            # deregister service
            loop.run_until_complete(
                slp_client.deregister(url=url)
            )
            print('{} - service is deregistered successfully'.format(url))
```

Code 4.1: slptool.py slightly modified main code

```python
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    ip_addrs = ['127.0.0.1']
    if ip_addrs is None:
        raise Exception('You should set ip address')
    loop.run_until_complete(create_slpd(ip_addrs))
    loop.run_forever()
```

Code 4.2: slpd.py main code

In order to try if PySLP works, the first thing to do is to execute slpd (which will work forever), while the second one is to launch slptool.py:

```
user@host:~$ python3 slptool.py &
user@host:~$ python3 slpd.py
```

The expected output generated by slpd.py execution should be nothing except its process ID, while slptool.py should display the service creation/discovery procedure.

```
user@host:~$ python3 slpd.py &
[1] 5044

user@host:~$ python3 slptool.py
Traceback (most recent call last):
  File "slptool.py", line 119, in _wait
    result = done.pop().result()  # raise exception
  File "slptool.py", line 102, in _wait
    result = f.result()
  File "slptool.py", line 79, in _send
    raise SLPClientError('Internal error')
__main__.SLPClientError: Internal error

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "slptool.py", line 213, in <module>
    attr_list=''
  File "/usr/lib/python3.6/asyncio/base_events.py", line 484, in
     run_until_complete
    return future.result()
  File "slptool.py", line 146, in register
    yield from self.send(data)
  File "slptool.py", line 135, in send
    return (yield from self._wait(fs))
  File "slptool.py", line 123, in _wait
    raise SLPClientError('SLP error code: {}'.format(error_code))
UnboundLocalError: local variable 'error_code' referenced before
    assignment
```

Code 4.3: First PySLP attempt without root privileges.

It can be seen that the output reports an apparently strange error which, at the end, can be simply solved executing slpd.py with super user permissions. This is quite intuitively but, since PySLP is open source, a proper explicative error message is the first improvement which could be performed.
Afterwards, the above commands are run again using *sudo*:

```
user@host:~$ sudo python3 slpd.py &
[1] 2846

user@host:~$ python3 slptool.py
service:test://test.com - service is registered successfully
service:test://test_1.com - service is registered successfully
([['service:test://test.com', 'service:test://test_1.com']],
    ['127.0.0.1'])
findsrvs for service:test - ([['service:test://test.com', 'service:test
    ://test_1.com']], ['127.0.0.1'])
findattrs for service:test://test.com - A=1
service:test://test.com - service is deregistered successfully
findattrs for service:test://test_1.com - B=2
service:test://test_1.com - service is deregistered successfully
```

Code 4.4: First PySLP attempt with root privileges.

As it can be seen, the library seems to work well, hence step two can be processed.

Second step basically repeats the previous point exiting from the local host environment. In principle, this step could be performed on two different computers but, since FORCH is deeply based on container technology, it was performed on docker containers. Docker gives the possibility to automatize the installation and the configuration of a container using a script called *Dockerfile*. The Dockerfile contains various types of commands that are prevalently used to:

- Specify the source Docker image using the command **FROM**.

- Execute automatic commands on the source image previously defined using **RUN**. The commands are not executed by the built container, but they are executed at build time. In the former case the command that should be used is **CMD**.
- Copy files into the stock Docker image used, exploiting the command **COPY**.

Of course, Dockerfile does not have only three commands, but those previously reported are the most important. The Dockerfile used for step two purposes use also the **WORKDIR** command to set the default start-up directory of the container. WORKDIR is actually something more, in Dockerfile environment, it is equivalent to the bash **cd** command which sets the execution directory of the following command-line instructions. Afterwards, it will be reported the Dockerfile needed to perform step two. Of course, the files slpd.py and slptool.py previously defined must be included in the Dockerfile directory.

```
# use Python official clean image
FROM python:3.6.9

# install nano editor and PySLP library
RUN apt-get update
RUN apt-get install nano
RUN pip3 install --upgrade pip
RUN pip3 install pyslp

# copy everything present in the Dockerfile directory into the container
COPY . /home
WORKDIR /home
```

Code 4.5: PySLP testing: step two Dockerfile.

Now that Dockerfile is ready, it remains only to:

- Create and start the containers.
- Check the IP address of the network interface of the containers.
- Modify slpd.py and slptool.py mains in order to: set the IP address found on the previous point, and separate services registration and requests between the two containers, creating a SA and an UA.

- Start the two modified .py files on the two hosts and check if everything works.

Afterwards, it will be shown the shell process carried out to create one of the two images. The procedure performed for the second one is the same.

```
user@host:~/dockerfile_directory$ sudo docker build -t service_agent .
Sending build context to Docker daemon  17.41kB
Step 1/7 : FROM python:3.6.9
3.6.9: Pulling from library/python
...
Status: Downloaded newer image for python:3.6.9
 ---> 5bf410ee7bb2
Step 2/7 : RUN apt-get update
 ---> Running in 49449632c19e
...
Removing intermediate container 49449632c19e
 ---> 61f22591880f
Step 3/7 : RUN apt-get install nano
 ---> Running in 525ef3bd4d17
...
Removing intermediate container 525ef3bd4d17
 ---> 2a6ef7700e33
Step 4/7 : RUN pip3 install --upgrade pip
 ---> Running in 9c34e57dc886
Collecting pip
  Downloading https://files.pythonhosted.org/packages/cb/28/91
      f26bd088ce8e22169032100d4260614fc3da435025ff389ef1d396a433/pip
      -20.2.4-py2.py3-none-any.whl (1.5MB)
Installing collected packages: pip
  Found existing installation: pip 19.3.1
    Uninstalling pip-19.3.1:
      Successfully uninstalled pip-19.3.1
Successfully installed pip-20.2.4
Removing intermediate container 9c34e57dc886
 ---> 8687bb0be682
Step 5/7 : RUN pip3 install pyslp
 ---> Running in 66317332161d
Collecting pyslp
  Downloading pyslp-0.1.7-py3-none-any.whl (10 kB)
Installing collected packages: pyslp
Successfully installed pyslp-0.1.7
Removing intermediate container 66317332161d
 ---> 50d0e47b2bdb
Step 6/7 : COPY . /home
 ---> 6adee9f79e95
Step 7/7 : WORKDIR /home
 ---> Running in 1a4e3426f306
Removing intermediate container 1a4e3426f306
 ---> 7768fcbecac3
```

```
Successfully built 7768fcbecac3
Successfully tagged service_agent:latest
```

Code 4.6: Creation of a SLP SA Docker image from a Dockerfile properly configured.

Now that the SA container is ready, the following points can be performed. Before to proceed it is interesting to highlight that in order to execute RUN commands, Docker creates temporary containers, one for each RUN.

```
user@host:~/dockerfile_directory$ sudo docker run -it service_agent
    bash
root@25c547235139:/home# ls
Dockerfile  slpd.py  slptool.py
root@25c547235139:/home# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    group default qlen 1000
     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
     inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
16: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
     state UP group default
     link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
     inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
root@25c547235139:/home# nano slptool.py
root@25c547235139:/home# nano slpd.py
root@25c547235139:/home# python3 slpd.py &
root@25c547235139:/home# python3 slptool.py
service:test://test.com - service is registered successfully
service:test://test_1.com - service is registered successfully
```

Code 4.7: Start of the previous SLP SA Docker image followed by a test services registration.

These were the commands to register services on SA SLP daemon. Using the command **ls**, it can be seen that the files included in the Dockerfile directory can be correctly found in /home, which is also the default directory at container start-up (as expected because of WORKDIR command).
For what concerns the modification of Python files, only the IP address in the slpd.py ___main___ has been changed, according to the one found with **ip addr show** (172.17.0.2). The ___main___ of slptool.py instead required more changes in order to set the proper IP address and to avoid service request messages.

```python
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    ip_addrs = '172.17.0.2' #'127.0.0.1'
    if ip_addrs is None:
        raise Exception('You should set ip address')
    slp_client = SLPClient(ip_addrs=ip_addrs)
    service_type = 'service:test'

    cnt = 0
    for url in ['service:test://test.com', 'service:test://test_1.com']:
        loop.run_until_complete(
            slp_client.register(
                service_type=service_type,
                lifetime=15,
                url=url,
                attr_list=chr(ord('A')+cnt) + '=' + str(1+cnt)
            )
        )
        print('{} - service is registered successfully'.format(url))
        cnt = cnt+1
```

Code 4.8: slptool.py modified ___main___ code in order to simulate only a SA.

Afterwards, as done for the SA container, it will be reported the output of the shell used to configure and run the UA container, and the modified ___main___ of slptool.py.

```
user@host:~/dockerfile_directory$ sudo docker build -t user_agent .
...
user@host:~/dockerfile_directory$ sudo docker run -it user_agent bash
root@e30a1c80255d:/home# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    group default qlen 1000
     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
     inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
18: eth0@if19: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
     state UP group default
     link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
     inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
root@e30a1c80255d:/home# nano slptool.py
root@e30a1c80255d:/home# python3 slptool.py
([['service:test://test_1.com', 'service:test://test.com']],
    ['172.17.0.3'])
```

```
findsrvs for service:test - ([['service:test://test_1.com', 'service:
    test://test.com']], ['172.17.0.3'])
findattrs for service:test://test_1.com - B=2
service:test://test_1.com - service is deregistered successfully
findattrs for service:test://test.com - A=1
service:test://test.com - service is deregistered successfully
root@e30a1c80255d:/home# python3 slptool.py
([[]], ['172.17.0.3'])
findsrvs for service:test - ([[]], ['172.17.0.3'])
root@e30a1c80255d:/home# python3 slptool.py
([[]], ['172.17.0.3'])
findsrvs for service:test - ([[]], ['172.17.0.3'])
```

Code 4.9: Creation and start of a SLP UA Docker image followed by a test services request.

```python
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    ip_addrs = '172.17.0.3' #'127.0.0.1'
    if ip_addrs is None:
        raise Exception('You should set ip address')
    slp_client = SLPClient(ip_addrs=ip_addrs)
    service_type = 'service:test'

    url_entries = loop.run_until_complete(
        slp_client.findsrvs(service_type=service_type)
    )
    print(url_entries)
    print('findsrvs for {} - {}'.format(service_type, url_entries))
    for _url_entries in url_entries[0]:
        for url in _url_entries:
            attr_list = loop.run_until_complete(
                slp_client.findattrs(url=url)
            )
            print('findattrs for {} - {}'.format(url, attr_list))
            loop.run_until_complete(
                slp_client.deregister(url=url)
            )
            print('{} - service is deregistered successfully'.format(url))
```

Code 4.10: slptool.py modified main code in order to simulate only a UA.

Paying attention to code 4.9, it can be seen that everything works as expected; every service announced by the SA is correctly found by the UA. The second **python3 slptool.py** command (performed some seconds after the first one) does not return anything because, as it can be seen in code 4.8, services have a lifetime of only 15 seconds. This confirms that also the lifetime mechanism is correctly implemented by PySLP.
Finally, it is important to put in evidence the fact that slpd.py was not used at all. Its code was not modified, as well as the daemon was not started. This because the UA performs only simple services requests and, according to section 4.1, the SLP daemon is not necessary.

Third step is the real first development part. The idea is to wrap PySLP inside a custom code, in order to provide a simpler integration of PySLP inside FORCH.
According to Open-Closed Principle contained in **SOLID** principles of object-oriented programming, the original PySLP code has not been touched.
Actually a slight modification of slptool.py would have been necessary to apply the previously mentioned library improvement, which is about the necessity to have a proper error message in presence of *slpd* launched without *sudo*. The PySLP error returned in code 4.4 is generated by an **except** instruction inside slptool.py. The error is raised if the SLP daemon process cannot be found because either it effectively misses, or it is started without the root privileges. The improvement required to add some code inside the internal **__wait** method present in slptool.py:

```python
...
except:
    if not('error_code' in locals()) and not('error_code' in globals()):
        error_code = 'Probably SLP daemon miss. Check also slpd privileges.'
    raise SLPClientError('SLP error code: {}'.format(error_code))
...
```

Code 4.11: slptool.py code to display an appropriate message in case slpd.py does not start correctly.

In order to respect the OCP, instead to modify slptool.py, the method _wait has been overridden with the modified code in the library wrapper, which contains a child class of the one which owns _wait method.

Before to continue it is important to specify that, since so far PySLP has been used without any DAs, the current idea at this point is to use SLP in distributed mode. Service nodes are SAs, while the orchestrator is a UA. Furthermore, as it will be revealed at the end of the section, PySLP has not been chosen as final solution for the implementation of SLP inside FORCH. Hence, the name of the classes, items, and in general the whole code, must be still considered at the development stage.

Given the previous clarifications, the main improvements that PySLP needs from the point of view of FORCH, will be listed below:

- **1**. Possibility to get the IP address of the network interfaces, allowing also to enable automatically SLP on every available IP address. The implementation of this point required an external Python library called: **netifaces**.
- **2**. Definition of a service class with methods for parsing FORCH services JSON files. In this way service objects are officially introduced and used in substitution of the services URLs strings.
  Service class is also useful to introduce real lists of attributes; PySLP **attr_list** in fact was only a string and not the Python list which was supposed to be.
  For a correct SLP integration, FORCH JSON files are supposed to change slightly. For example, a field (e.g. *attributes_list*) should be added and the attributes contained should be separated by a common defined symbol (% was used during the test).
- **3**. SLP daemon transparency from the FORCH point of view.
- **4**. Reorganization of SLPClient class defined in slptool.py in order rise the abstraction level and to provide methods that give access to the improvements done.
- **5**. Automatic services lifetime refresh at its expiration. The implementation of this point required an external Python library called: **timeloop**.
- **6**. Define a single class that aggregates everything.

Afterwards, it will be reported the code relative to the implementation of each listed improvement. The whole code is contained in **PySLPWrapper.py**, which depends on this modules:

```python
#system libraries
import netifaces
import json
import os
import time
import threading
import sys
import warnings
import asyncio
from concurrent.futures import FIRST_COMPLETED, ALL_COMPLETED

#external libraries
from timeloop import Timeloop
from datetime import timedelta

from pyslp.utils import get_lst
from pyslp.slptool import SLPClient, SLPClientError
from pyslp.slpd import create_slpd
```

Code 4.12: Dependencies of PySLPWrapper.py.

In order to implement the features relative to network interfaces and IP address and other eventual future utility methods, **SLPUtils** class has been created. Currently, it only contains the method **getIPByIfacesList**, which returns each IP address associated either to all present interfaces, or to a passed interfaces list.

```python
class SLPUtils():
    @staticmethod
    def getIPByIfacesList(iface = None):
        ip_addrs = []

        if iface == None:
            #get IP addresses from every iterface
            for ni in netifaces.interfaces():
                ip_addrs +=
                 ↪  [netifaces.ifaddresses(ni)[netifaces.AF_INET][0]['addr']]
        else:
            iface = get_lst(iface)
            for ni in iface:
                if isinstance(ni, str):
                    ip_addrs +=
                    ↪  [netifaces.ifaddresses(ni)[netifaces.AF_INET][0]['addr']]
                else:
                    warnings.warn('"{}": {}. Not processed because it is not
                    ↪  a string.'.format(ni, type(ni)))

        return ip_addrs
```

Code 4.13: PySLPWrapper.py: implementation of • **1**.

For what concerns the service objects definition, afterwards it is reported the **SLPService** class implementation:

```
class SLPService():

    attrListSeparator = '%'

    def __init__(self, service_name, url_ip, protocol = '', port = '',
    ↪  lifetime = 0xffff, attr_list = ''):
        if protocol != '':
            protocol = ':' + protocol
        self.service_type = 'service:' + service_name + protocol

        if port != '':
            port = ':' + port
        url_ip = get_lst(url_ip)
        self.urlList = [self.service_type + '://' + u + port for u in url_ip]

        self.lifetime = lifetime
        self.attr_list = attr_list

    def __eq__(self, obj):
        return isinstance(obj, SLPService) and obj.service_type ==
        ↪  self.service_type

    def __ne__(self, obj):
        return not self == obj

    @staticmethod
    def parseServicesListFromJson(ip, jsonServicesFile):
        ipList = get_lst(ip)
        jsonServicesFileList = get_lst(jsonServicesFile)

        servicesList = []

        for arg in jsonServicesFileList:
            if not(isinstance(arg, str)):
                warnings.warn('"{}": {}. Not processed because it is not a
                ↪   string.'.format(arg, type(arg)))
                continue

            if not(os.path.isfile(arg)):
                warnings.warn('"{}" is not a file or it doesn\'t
                ↪   exist.'.format(arg))
                continue

            with open(arg, 'r') as f:
                jsonDict = json.load(f)

            for asAServiceType in jsonDict:
                for service in jsonDict[asAServiceType]:
                    attr_list =
                    ↪   jsonDict[asAServiceType][service]['attr_list'].
                    ↪   split(SLPService.attrListSeparator)
                    servicesList += [SLPService(
                    ↪   service_name=jsonDict[asAServiceType][service]['name'],
                        url_ip=ipList,
                        protocol='http',
                        lifetime =
                        ↪   int(jsonDict[asAServiceType][service]['lifetime']),
                        attr_list=attr_list)]

        return servicesList
```

Code 4.14: PySLPWrapper.py: implementation of • **2**.

In order to make slpd transparent, **SLPDaemon** class has been created. It contains every method useful to SLP daemon administration.

```python
class SLPDaemon():

    daemonLoopStarted = False

    def __init__(self, ip_addrs, loop = None):
        self._ip_addrs = get_lst(ip_addrs)
        self._loop = loop or asyncio.new_event_loop()

    def __del__(self):
        self._loop.stop()

    def start(self):
        self._loop.run_until_complete(create_slpd(self._ip_addrs))
        SLPDaemon.daemonLoopStarted = True
        self._loop.run_forever()

    def stop(self):
        self._loop.stop()

    def getIPs(self):
        return self._ip_addrs

    def setIPByIfacesList(self, iface = None):
        self._ip_addrs = SLPUtils.getIPByIfacesList(iface)

        return self._ip_addrs
```

Code 4.15: PySLPWrapper.py: implementation of • **3**.

Afterwards, it is the moment of the heavier class, which cover • **3** and • **4**. Point three of the list is solved by class **SLPUser**, which is a child of SLPClient, while point four is implemented by an inner method which exploits timeloop library. Timeloop gives the possibility to periodically execute a piece of code.
In order to refresh expired services, the previous mentioned method uses a list of counters, one for each service which is locally owned and announced to the network. The counters are initialized with an integer equal to the service lifetime and every second they are decreased by one. Every time a counter reaches 0, the relative service is automatically refreshed.

```python
class SLPUser(SLPClient):

    def __init__(self, ip_addrs = None, mcast_group='239.255.255.253',
     ↪ mcast_port=427, loop=None, scope='DEFAULT'):
        super().__init__(ip_addrs, mcast_group, mcast_port, loop, scope)

        self._servicesList = []
        self._tl = None

    def __del__(self):
        if self._tl != None:
            self._tl.stop()

    #ovveride of father method to raise a precise and clear error if the slpd
     ↪ doesn't start with sudo
    @asyncio.coroutine
    def _wait(self, fs):
        flag_completed = False

        timeout = 5
        while not flag_completed:
            done, pending = yield from asyncio.wait(fs, timeout=timeout,
             ↪ return_when=ALL_COMPLETED)

            if not done:
                for f in pending:
                    f.cancel()
                raise SLPClientError('Internal error')

            for f in done:
                try:
                    result = f.result()
                    error_code = result['error_code']
                    if error_code == 0:
                        flag_completed = True
                        break
                except:
                    pass

            timeout /= 2

            if flag_completed:
                break

            if pending:
                fs = list(pending)
            else:
                try:
                    result = done.pop().result()  # raise exception
                    if not pending:
                        raise SLPClientError('SLP error code:
                         ↪  {}'.format(error_code))
                except:
                    if not('error_code' in locals()) and not('error_code' in
                     ↪  globals()):
                        error_code = 'Probably either slp daemon didn\'t
                         ↪  start or it didn\'t start with "sudo"'
                    raise SLPClientError('SLP error code:
                     ↪  {}'.format(error_code))

        return result

    def getIPs(self):
        return self.ip_addrs

    def setIPByIfacesList(self, iface = None):
        self.ip_addrs = SLPUtils.getIPByIfacesList(iface)

        return self.ip_addrs

    def getServices(self):
        return self._servicesList

    def setServices(self, servicesList):
        self._servicesList = servicesList

        return servicesList

    #adds or replace services
    def updateServices(self, servicesList):
        servicesList = get_lst(servicesList)

        #check if service is already present and delete it
        self.delServices(servicesList)
```

```python

        self._servicesList += servicesList

    def delServices(self, servicesList):
        i = 0

        while i<len(self._servicesList):
            i2 = 0

            while i2<len(servicesList):
                if self._servicesList[i] == servicesList[i2]:
                    self._servicesList.pop(i)
                    i = i-1
                    i2 = len(servicesList)
                else:
                    i2 += 1
            i += 1

    def registerServices(self):
        self._tl = Timeloop()

        self._jobServicesList = None

        @self._tl.job(interval=timedelta(seconds=1))
        def registerAndRefreshServices():                        # pylint:
        ↪   disable=unused-variable
            if self._jobServicesList == None:
                self._jobServicesList = self._servicesList
                self._lifetimeList = [s.lifetime for s in
                ↪   self._jobServicesList]
            else:
                self._jobServicesList = []

                for i in range(len(self._lifetimeList)):
                    if self._lifetimeList[i] == 0:
                        self._lifetimeList[i] =
                        ↪   self._servicesList[i].lifetime
                        self._jobServicesList += [self._servicesList[i]]

            if self._jobServicesList != []:
                urls = self._registerServices(self._jobServicesList)
                print('{} - service is registered/refreshed
                ↪   successfully'.format(urls)) #for debugging

            self._lifetimeList = [e-1 for e in self._lifetimeList]

        self._tl.start()

    def _registerServices(self, serviceList):
        urls = []

        for s in serviceList:
            #workaround because library doesn't work with list of attributes
            attrString = ''

            if s.attr_list != '':
                for attr in get_lst(s.attr_list):
                    attrString += attr + SLPService.attrListSeparator

                attrString = attrString[:-1] #remove unwanted final
                 ↪  "SLPService.attrListSeparator"

            for url in s.urlList:
                self.loop.run_until_complete(
                #self._loop.run_until_complete(
                    self.register(
                        service_type=s.service_type,
                        lifetime=s.lifetime,
                        url=url,
                        attr_list=attrString
                    )
                )
                urls += [url]

        return urls

    def deregisterServices(self):
        urls = []

        if self._tl != None:
            self._tl.stop()
            self._tl = None

            for s in self._servicesList:
                for url in s.urlList:
```

```python
                    self.loop.run_until_complete(
                        self.deregister(url=url)
                    )
                    urls += [url]

        return urls

    def findService(self, service_type):
        url_entries = self.loop.run_until_complete(
        #url_entries = self._loop.run_until_complete(
            self.findsrvs(service_type=service_type)
        )

        return url_entries

    def findServiceAttributes(self, url):
        attr_list = self.loop.run_until_complete(
            self.findattrs(url=url)
        )

        #workaround because library doesn't work with list of attributes
        if attr_list != []:
            attr_list = attr_list.split(SLPService.attrListSeparator)

        return attr_list
```

Code 4.16: PySLPWrapper.py: implementation of • **4** and • **5**.

Finally, the last point aims to hid the complexity of the library and the previous treated code in a single class called **SLP**, which extends SLPUser class. SLP class joins all the previous codes, giving to the user a single place from where to manage every SLP aspect. SLP class provides the access to SLPUtils method and, being a child of SLPUser, it provides every method of the father. At the same time, in its constructor, it starts a thread running slpd, hiding the latter to the user.

```python
class SLP(SLPUser):
    def __init__(self, ip_addrs = None, mcast_group='239.255.255.253',
    ↪ mcast_port=427, loop=None, scope='DEFAULT', startDaemon=True):
        if startDaemon == True:
            class myDaemonThread(threading.Thread):
                def __init__(self):
                    threading.Thread.__init__(self, daemon=True)
                    self._sldp = None

                def __del__(self):
                    if self._sldp != None:
                        self._sldp.stop()

                def run(self):
                    self._slpd = SLPDaemon(ip_addrs)
                    # print('Start daemon') #for debugging
                    self._slpd.start()

            t = myDaemonThread()
            t.start()

            while SLPDaemon.daemonLoopStarted == False:
                pass

        super().__init__(ip_addrs, mcast_group, mcast_port, loop, scope)

    def __del__(self):
        super().__del__()

    @staticmethod
    def getIPByIfacesList(iface = None):
        return SLPUtils.getIPByIfacesList(iface)
```

Code 4.17: PySLPWrapper.py: implementation of • **6**.

Fourth step aims to test the code of the third one in a more complex environment with respect to the one of step two.
This time it will be deployed one UA (i.e. the orchestrator) and three SAs (i.e. three hypothetical service nodes). Each SA provides some services belonging to one of the three typologies mentioned in section 2.1 (i.e. Saas, PaaS, IaaS). The Dockerfile is the same of the previous step, but inside its directory there are also PySLPWrapper.py, a Python script to execute and the three FORCH service JSON files.

The orchestrator's ___main___ script is called **slpClient_orch.py** and takes in input the network interface to be used for SLP, and a JSON file containing the known service types. If the first parameter misses, SLP is supposed to work on all available network interfaces; if the second parameter misses, service types will be taken from every available JSON file present in the directory. The orchestrator performs only a simple service discovery, hence the SLP daemon is not necessary. Afterwards, it will be reported slpClient_orch.py code:

```python
import time
import sys

from PySLPWrapper import SLP, SLPService

if len(sys.argv)<=1:
    slp_client = SLP(ip_addrs=SLP.getIPByIfacesList(), startDaemon=False)
    servicesList = SLPService.parseServicesListFromJson(slp_client.getIPs(),
    ↪  ["./db_apps.json","./db_sdps.json","./db_fves.json"])
elif len(sys.argv)==2:
    if sys.argv[1] == 'all':
        slp_client = SLP(ip_addrs=SLP.getIPByIfacesList(), startDaemon=False)
    else:
        slp_client = SLP(ip_addrs=SLP.getIPByIfacesList(sys.argv[1]))
    servicesList = SLPService.parseServicesListFromJson(slp_client.getIPs(),
    ↪  ["./db_apps.json","./db_sdps.json","./db_fves.json"])
else:
    if sys.argv[1] == 'all':
        slp_client = SLP(ip_addrs=SLP.getIPByIfacesList(), startDaemon=False)
    else:
        slp_client = SLP(ip_addrs=SLP.getIPByIfacesList(sys.argv[1]),
        ↪  startDaemon=False)
    servicesList = SLPService.parseServicesListFromJson(slp_client.getIPs(),
    ↪  sys.argv[1:])

print(slp_client.getIPs())
print()

url_entries = []

for s in servicesList:
    url_entry = slp_client.findService(s.service_type)
    url_entries.append(url_entry)
    print('findsrvs for {} - {}\n'.format(s.service_type, url_entry))

if isinstance(url_entries[1][0], str):
    for _url_entries in url_entries[0]:
        for url in _url_entries:
            attr_list = slp_client.findServiceAttributes(url)
            print('findattrs for {} - {}'.format(url, attr_list))
    print()
else:
    for uE in url_entries:
        for _url_entries in uE[0]:
            for url in _url_entries:
                attr_list = slp_client.findServiceAttributes(url)
                print('findattrs for {} - {}'.format(url, attr_list))
        print()
```

Code 4.18: slpClient_orch.py code. This is an example of proof of concept to test PySLPWrapper.py from the User Agent point of view.

The nodes's ___main___ script is called **slpClient_node.py** and takes in input the same parameters of slpClient_orch.py. Differently from the previous script, the JSON files are integrally used to know the services supported and announced by the node. Since nodes have the role of Service Agents, SLP daemon is necessary and the services are registered instead of being discovered. Due to parameters introduced (e.g. services lifetime), JSON files have been integrated with the missing services information. For instance, *APP001* shown in code 2.1 becomes:

```json
{
  "apps": {
    "APP001": {
      "thumbnail": "http://137.204.57.73/restooltest/thumb_httpd.jpg",
      "name": "httpd",
      "descr": "Apache web server",
      "lifetime": 2,
      "attr_list": "a=A%b=B"
    },
    ...
}
```

Code 4.19: Example of JSON service file with additional parameters required by SLP.

Such as APP001, also the other services have been set with a very short lifetime period. Of course, the actual lifetime of real services is much longer, but in this case slpClient_node.py aims to test PySLPWrapper.py, including its automatic services refresh feature.
Afterwards, slpClient_node.py code will be reported:

```python
import time
import sys
import asyncio

from PySLPWrapper import SLP, SLPService

if len(sys.argv)<=1:
    slp_client = SLP(ip_addrs=SLP.getIPByIfacesList())
    servicesList = slp_client.setServices(
    ↪  SLPService.parseServicesListFromJson(slp_client.getIPs(),
    ↪  ["./db_apps.json","./db_sdps.json","./db_fves.json"]))
elif len(sys.argv)==2:
    if sys.argv[1] == 'all':
        slp_client = SLP(ip_addrs=SLP.getIPByIfacesList())
    else:
        slp_client = SLP(ip_addrs=SLP.getIPByIfacesList(sys.argv[1]))
    servicesList = slp_client.setServices(
    ↪  SLPService.parseServicesListFromJson(slp_client.getIPs(),
    ↪  ["./db_apps.json","./db_sdps.json","./db_fves.json"]))
else:
    if sys.argv[1] == 'all':
        slp_client = SLP(ip_addrs=SLP.getIPByIfacesList())
    else:
        slp_client = SLP(ip_addrs=SLP.getIPByIfacesList(sys.argv[1]))
    servicesList = slp_client.setServices(
    ↪  SLPService.parseServicesListFromJson(slp_client.getIPs(),
    ↪  sys.argv[2:]))

print(slp_client.getIPs())
print()

```

```python
for s in servicesList:
    print(s.getServiceDictEntry())
print()

#enable this to try the method which updates service parameters
#slp_client.updateServices(SLPService("a","127.0.0.1", lifetime=3,
↪  attr_list=['a', 'b']))

slp_client.registerServices()

#keep the process active forever in order to keep alive slpd and refresh
↪  services lifetime
asyncio.new_event_loop().run_forever()
```

Code 4.20: slpClient_node.py code. This is an example of proof of concept to test PySLPWrapper.py from the Service Agent point of view.

Now that the code is ready, it remains only to build, run the containers and test the system. The manual execution of slpd is no more required because it is automatically managed by the new library wrapper.
Since all the commands to be build and run the containers have already been shown, it will be reported only the output generated by the execution of the new Python scripts.

```
root@5fd073d2f0b8:/home# python3 slpClient_orch.py eth0
['172.17.0.2']

findsrvs for service:httpd:http - ([[]], ['172.17.0.2'])
findsrvs for service:stress:http - ([[]], ['172.17.0.2'])
findsrvs for service:python:http - ([[]], ['172.17.0.2'])
findsrvs for service:alpine:http - ([[]], ['172.17.0.2'])
findsrvs for service:sleepython:http - ([[]], ['172.17.0.2'])
findsrvs for service:docker:http - ([[]], ['172.17.0.2'])
findsrvs for service:fog05:http - ([[]], ['172.17.0.2'])
```

Code 4.21: Orchestrator output generated when no services nodes are deployed on the network.

```
root@b2d0bddb01fc:/home# python3 slpClient_node.py eth0 ./db_apps.json
['172.17.0.3']

{'service:httpd:http': {'url': ['service:httpd:http://172.17.0.3'], '
   lifetime': 2, 'attr_list': ['a=A', 'b=B']}}
```

```
{'service:stress:http': {'url': ['service:stress:http://172.17.0.3'], '
    lifetime': 4, 'attr_list': ['']}}

[2020-11-12 07:22:43,375] [timeloop] [INFO] Starting Timeloop..
[2020-11-12 07:22:43,375] [timeloop] [INFO] Registered job <function
    SLPUser.registerServices.<locals>.registerAndRefreshServices at 0
    x7f3ba222f730>
[2020-11-12 07:22:43,375] [timeloop] [INFO] Timeloop now started. Jobs
    will run based on the interval set
['service:httpd:http://172.17.0.3', 'service:stress:http://172.17.0.3']
     - service is registered/refreshed successfully
['service:httpd:http://172.17.0.3'] - service is registered/refreshed
    successfully
^C
```

Code 4.22: Node1 output generated during slpClient_node.py execution.

```
root@5fd073d2f0b8:/home# python3 slpClient_orch.py eth0
['172.17.0.2']

findsrvs for service:httpd:http - ([['service:httpd:http
    ://172.17.0.3']], ['172.17.0.2'])
findsrvs for service:stress:http - ([['service:stress:http
    ://172.17.0.3']], ['172.17.0.2'])

findsrvs for service:python:http - ([[]], ['172.17.0.2'])
findsrvs for service:alpine:http - ([[]], ['172.17.0.2'])
findsrvs for service:sleepython:http - ([[]], ['172.17.0.2'])
findsrvs for service:docker:http - ([[]], ['172.17.0.2'])
findsrvs for service:fog05:http - ([[]], ['172.17.0.2'])
findattrs for service:httpd:http://172.17.0.3 - ['a=A', 'b=B']
findattrs for service:stress:http://172.17.0.3 - []
```

Code 4.23: Orchestrator output generated when Node1 has been started.

```
root@5fd073d2f0b8:/home# python3 slpClient_orch.py eth0
['172.17.0.2']

findsrvs for service:httpd:http - ([['service:httpd:http
    ://172.17.0.3']], ['172.17.0.2'])
findsrvs for service:stress:http - ([['service:stress:http
    ://172.17.0.3']], ['172.17.0.2'])
findsrvs for service:python:http - ([['service:python:http
    ://172.17.0.4']], ['172.17.0.2'])
findsrvs for service:alpine:http - ([['service:alpine:http
    ://172.17.0.4']], ['172.17.0.2'])
findsrvs for service:sleepython:http - ([['service:sleepython:http
    ://172.17.0.4']], ['172.17.0.2'])
findsrvs for service:docker:http - ([['service:docker:http
    ://172.17.0.5']], ['172.17.0.2'])
findsrvs for service:fog05:http - ([['service:fog05:http
    ://172.17.0.5']], ['172.17.0.2'])

findattrs for service:httpd:http://172.17.0.3 - ['a=A', 'b=B']
findattrs for service:stress:http://172.17.0.3 - []
findattrs for service:python:http://172.17.0.4 - ['a=A', 'b=B']
findattrs for service:alpine:http://172.17.0.4 - []
findattrs for service:sleepython:http://172.17.0.4 - ['a=A', 'b=B', 'c=
    C']
findattrs for service:docker:http://172.17.0.5 - []
findattrs for service:fog05:http://172.17.0.5 - []
```

Code 4.24: Orchestrator output generated when all the three nodes have been started.

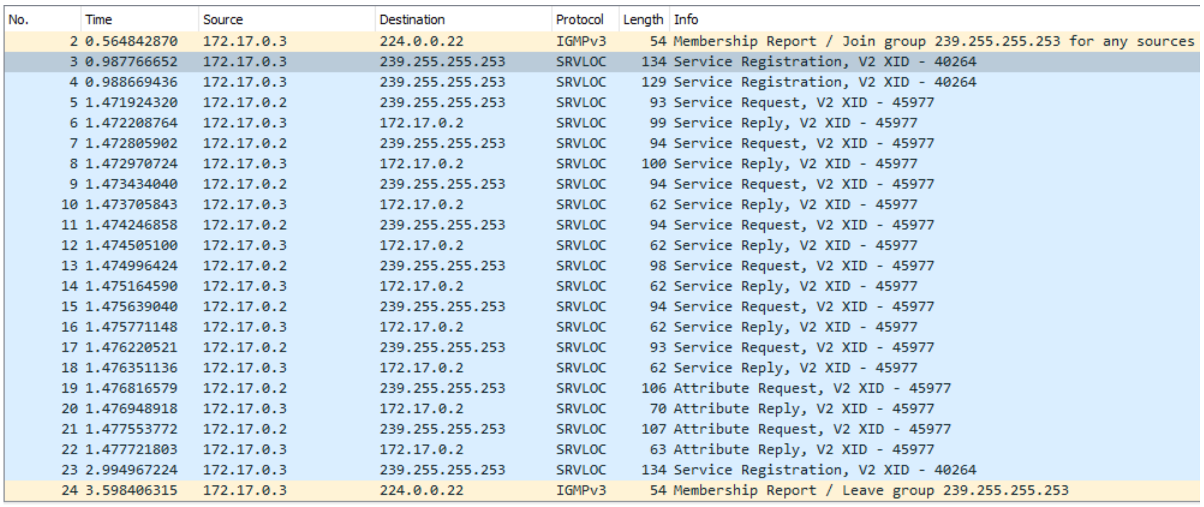
In code 4.22 it can be seen the services that Node1 announces and their automatic re-registration after their expiration time.
Since the other two nodes have been started with:

```
root@f3a8ab983e49:/home# python3 slpClient_node.py eth0 ./db_sdps.json
root@32a245595d8c:/home# python3 slpClient_node.py eth0 ./db_fves.json
```

they generate outputs similar to the one of node1, but with different services. For this reason their terminal output will not be reported.
Code 4.23 and 4.24 show that everything works as expected, in fact, every service is found as well as their attributes.
For completeness, afterwards it will be shown the SLP network traffic generated by node1's service registrations and by orchestrator's services requests. Of course, the network traffic generated by the remaining nodes is similar to the one generated by node1.

Figure 4.2 shows the packet generated by the previous reported scripts. The different kind of visible packets and their flow, will be now commented.

- Packets 2 and 24 are respectively the join/leave messages to the multicast group.
- Packets 3 and 4 are the service registrations of the services parsed from db_apps.json.
- Packets from 5 to 18 are the service request/reply messages from and to the orchestrator. There is one request for each service inside the three mentioned JSON files. Node1 knows only the first two services, in fact it replies with a non empty packet only to the first two requests. The latter sentence can be easily verified using the figure; empty service reply messages have a length of 62.



No.	Time	Source	Destination	Protocol	Length	Info
2	0.564842870	172.17.0.3	224.0.0.22	IGMPv3	54	Membership Report / Join group 239.255.255.253 for any sources
3	0.987766652	172.17.0.3	239.255.255.253	SRVLOC	134	Service Registration, V2 XID - 40264
4	0.988669436	172.17.0.3	239.255.255.253	SRVLOC	129	Service Registration, V2 XID - 40264
5	1.471924320	172.17.0.2	239.255.255.253	SRVLOC	93	Service Request, V2 XID - 45977
6	1.472208764	172.17.0.3	172.17.0.2	SRVLOC	99	Service Reply, V2 XID - 45977
7	1.472805902	172.17.0.2	239.255.255.253	SRVLOC	94	Service Request, V2 XID - 45977
8	1.472970724	172.17.0.3	172.17.0.2	SRVLOC	100	Service Reply, V2 XID - 45977
9	1.473434040	172.17.0.2	239.255.255.253	SRVLOC	94	Service Request, V2 XID - 45977
10	1.473705843	172.17.0.3	172.17.0.2	SRVLOC	62	Service Reply, V2 XID - 45977
11	1.474246858	172.17.0.2	239.255.255.253	SRVLOC	94	Service Request, V2 XID - 45977
12	1.474505100	172.17.0.3	172.17.0.2	SRVLOC	62	Service Reply, V2 XID - 45977
13	1.474996424	172.17.0.2	239.255.255.253	SRVLOC	98	Service Request, V2 XID - 45977
14	1.475164590	172.17.0.3	172.17.0.2	SRVLOC	62	Service Reply, V2 XID - 45977
15	1.475639040	172.17.0.2	239.255.255.253	SRVLOC	94	Service Request, V2 XID - 45977
16	1.475771148	172.17.0.3	172.17.0.2	SRVLOC	62	Service Reply, V2 XID - 45977
17	1.476220521	172.17.0.2	239.255.255.253	SRVLOC	93	Service Request, V2 XID - 45977
18	1.476351136	172.17.0.3	172.17.0.2	SRVLOC	62	Service Reply, V2 XID - 45977
19	1.476816579	172.17.0.2	239.255.255.253	SRVLOC	106	Attribute Request, V2 XID - 45977
20	1.476948918	172.17.0.3	172.17.0.2	SRVLOC	70	Attribute Reply, V2 XID - 45977
21	1.477553772	172.17.0.2	239.255.255.253	SRVLOC	107	Attribute Request, V2 XID - 45977
22	1.477721803	172.17.0.3	172.17.0.2	SRVLOC	63	Attribute Reply, V2 XID - 45977
23	2.994967224	172.17.0.3	239.255.255.253	SRVLOC	134	Service Registration, V2 XID - 40264
24	3.598406315	172.17.0.3	224.0.0.22	IGMPv3	54	Membership Report / Leave group 239.255.255.253

Figure 4.2: Packets generated by the execution of slpClient_node.py on node1 and slpClient_node.py on the orchestrator.

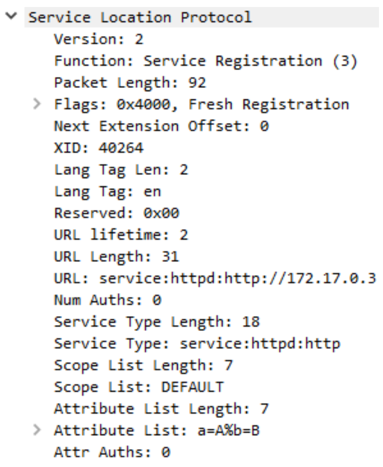
- Packets from 19 to 22 are the attribute request/reply messages from and to the orchestrator. Differently from the previous point, there are only two requests because they are done only for the discovered services, which are the two ones of node1.

- Packet 23 is a new registration of the first service. This happened because, as can be seen in figure 4.3(a), the first announced service has a lifetime value of only two seconds. The short lifetime was intentionally set to test the auto-refreshing service feature before mentioned.

4.2.3 PySLP limitation and issues

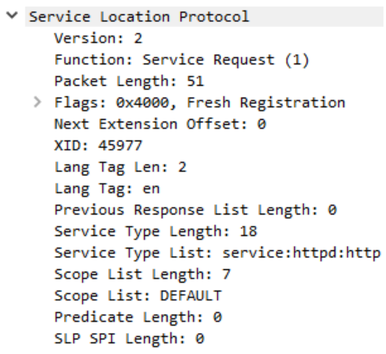
The subsection shows that PySLP works, but comparing [5] with PySLP it can be noted some behavioural mismatches about service registrations.

- SrvReg messages are supposed to be only used in the centralized mode of SLP. However, PySLP uses them also in the distributed case to register the local services to slpd. This behaviour, which can be seen as not SLP compliant, is actually contained in [8], hence it is the right implementation way. The problem is that the service registrations must be sent in unicast to the loopback instead of multicast. The reuse of SrvReg described by SLP API RFC is surely smart, it in fact avoids



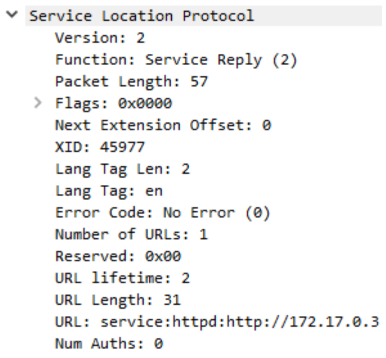
```
Service Location Protocol
    Version: 2
    Function: Service Registration (3)
    Packet Length: 92
    Flags: 0x4000, Fresh Registration
    Next Extension Offset: 0
    XID: 40264
    Lang Tag Len: 2
    Lang Tag: en
    Reserved: 0x00
    URL lifetime: 2
    URL Length: 31
    URL: service:httpd:http://172.17.0.3
    Num Auths: 0
    Service Type Length: 18
    Service Type: service:httpd:http
    Scope List Length: 7
    Scope List: DEFAULT
    Attribute List Length: 7
    Attribute List: a=A%b=B
    Attr Auths: 0
```

(a) Service Registration example.



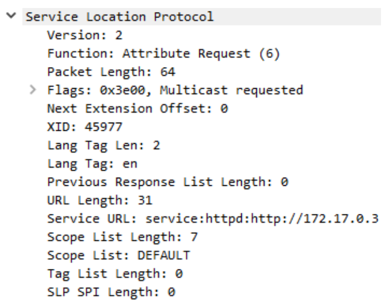
```
Service Location Protocol
    Version: 2
    Function: Service Request (1)
    Packet Length: 51
    Flags: 0x4000, Fresh Registration
    Next Extension Offset: 0
    XID: 45977
    Lang Tag Len: 2
    Lang Tag: en
    Previous Response List Length: 0
    Service Type Length: 18
    Service Type List: service:httpd:http
    Scope List Length: 7
    Scope List: DEFAULT
    Predicate Length: 0
    SLP SPI Length: 0
```

(b) Service Request example.



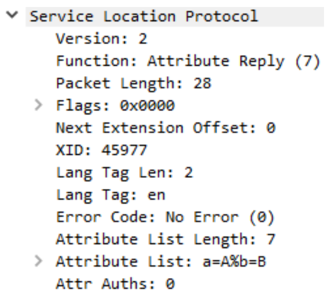
```
Service Location Protocol
    Version: 2
    Function: Service Reply (2)
    Packet Length: 57
    Flags: 0x0000
    Next Extension Offset: 0
    XID: 45977
    Lang Tag Len: 2
    Lang Tag: en
    Error Code: No Error (0)
    Number of URLs: 1
    Reserved: 0x00
    URL lifetime: 2
    URL Length: 31
    URL: service:httpd:http://172.17.0.3
    Num Auths: 0
```

(c) Service Reply example.



```
Service Location Protocol
    Version: 2
    Function: Attribute Request (6)
    Packet Length: 64
    Flags: 0x3e00, Multicast requested
    Next Extension Offset: 0
    XID: 45977
    Lang Tag Len: 2
    Lang Tag: en
    Previous Response List Length: 0
    URL Length: 31
    Service URL: service:httpd:http://172.17.0.3
    Scope List Length: 7
    Scope List: DEFAULT
    Tag List Length: 0
    SLP SPI Length: 0
```

(d) Attribute Request example.



```
Service Location Protocol
    Version: 2
    Function: Attribute Reply (7)
    Packet Length: 28
    Flags: 0x0000
    Next Extension Offset: 0
    XID: 45977
    Lang Tag Len: 2
    Lang Tag: en
    Error Code: No Error (0)
    Attribute List Length: 7
    Attribute List: a=A%b=B
    Attr Auths: 0
```

(e) Attribute Reply example.

Figure 4.3: SLP messages generated during code testing: one figure for each type of exchanged message.

to create unnecessary dedicated mechanisms to achieve the scope, but it messes the protocol if used or interpreted in a wrong way. SLP doesn't suppose to use multicast for SrvRegs, hence this characteristic makes PySLP uncompliant with the standard. This uncorrect use of SrvRegs allows services to be discovered, but brings an avoidable high network overhead. If a slpd receives a service registration it records the service in its local cache and, since each slpd listens to multicast address, every SA will reply to an hypothetical service request message. About this point SLP is very clear: if there are no DAs, only the SA that owns the services must reply to a service request.

- As said in previous sections, the best way to use SLP inside FORCH is in centralized mode. Reading [8] can be understood that the difference between DAs and SAs is only in the parameters used to start slpd. It is not clear if and how PySLP slpd's implementation provides this feature. This is a big issue from the point of view of FORCH integration.

- Some messages (e.g. service type request/reply) miss. Looking PySLP source code, it is clear that, if required, they must be implemented from zero.

PySLP is hence definitively not compliant with neither [5] nor [8], and these issues put in evidence that probably the library hasn't ever been really finished. In the end, the code isn't enough mature to be used in a structured project like FORCH. Each listed point must be solved in order use PySLP inside FORCH, requiring a non-trivial huge coding effort. Of course this remains an option, but before to start with these massive library improvements, it is better to look for another option already sufficiently developed.

4.3 Adopted solution

In this section it will be illustrated the found PySLP alternatives, and it will be reported the final code implemented to integrate SLP in FORCH.

4.3.1 RFC 2614 compliant libraries

PySLP was the only available SLP Python library, hence to avoid to continue its implementation, the only possibility was to find a library in another programming language and use it through some Python hacks. The use of non-Python code in Python, could seem an hard task but, since there are a lot of tutorials and modules, this option is apparently simpler than concluding PySLP implementation. For instance, for Java code there is a bridge module called Py4J while, since Python is written in C [11], C code integration is directly supported from Python without any additional module [12]. The last sentence makes C the most preferable external programming language to be used with Python. Considering now that [8] doesn't define only a possible theoretical API implementation, but provides also an interface-level implementation API in C and Java language, probably the best SLP libraries have been written with these two languages.

jSLP and its successor

The best SLP Java implementation is, or at least was called jSLP and is now integrated in the Eclipse ECF Project [13]. Since this library is still actively developed [14, 15] it would be the best choice, but it is not in C language.

OpenSLP

The only available C option is called OpenSLP [16]. Differently from jSLP which in some way depends from Eclipse, OpenSLP is totally independent and wants to be as much as possible compliant with [8]. It provides an acceptable online documentation and, even if it is not actively updated, the developer seems to be still available for pull requests [17]. Since the official releases are stopped since 2013, to apply the latest commits the code must be autonomously compiled, but fortunately the documentation is quite clear. OpenSLP is not a simple SLP C library, but it is a project that aimed to spread SLP as much as possible, regardless the programming language. The major work was and is done on the C implementation, but the official GitHub repo contains also a no more maintained Java library and a Python 2 wrapper of the C implementation. When OpenSLP was more active, it was also officially included in Debian GNU/Linux. It means that it could be simply installed using *apt-get* command.

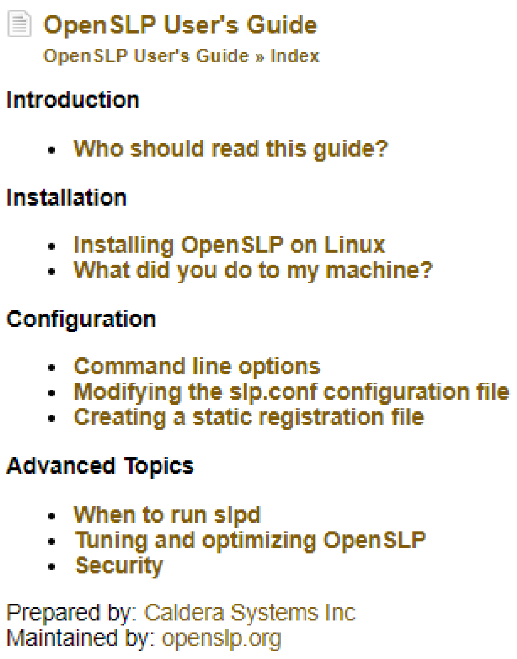
Due to maintenance reasons, probably the best solution would probably be the jSLP successor, but since FORCH is written in Python and OpenSLP offers a good starting point for Python 3 integration, in this case the best choice is OpenSLP.

4.3.2 OpenSLP library

OpenSLP project and its documentation can be easily found in the dedicated web site [16]. The documentation is divided in two.

- OpenSLP User's Guide: which contains all the indications to use SLP from the shell. Although this part of documentation should not be fundamental for developers, actually it is indispensable, because there are no APIs to control the daemon. Since there cannot exist more than one slpd for SLP host, this could be comprehensible, but in my opinion the fact that the daemon must be launched using shell is a strong drawback. In this way is not possible to completely embed SLP inside a program without performing a shell in vocation (always not a good-practice). This issue actually came from [8], hence it is not an OpenSLP fault.

- OpenSLP Programmer's Guide: which contains the API functions available for the integration of SLP inside a program by developers. Of course for what concerns this project, this is the most relevant and used part of the documentation.



OpenSLP User's Guide
OpenSLP User's Guide » Index

Introduction

- Who should read this guide?

Installation

- Installing OpenSLP on Linux
- What did you do to my machine?

Configuration

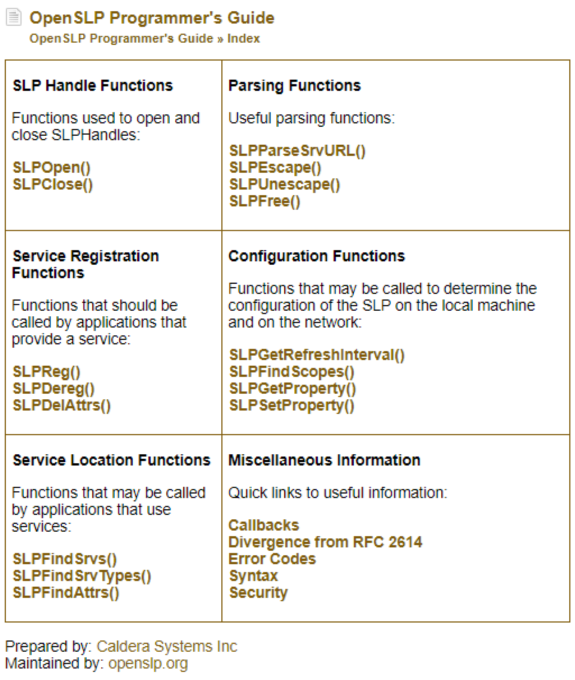
- Command line options
- Modifying the slp.conf configuration file
- Creating a static registration file

Advanced Topics

- When to run slpd
- Tuning and optimizing OpenSLP
- Security

Prepared by: Caldera Systems Inc
Maintained by: openslp.org

(a) OpenSLP User's Guide index.



OpenSLP Programmer's Guide
OpenSLP Programmer's Guide » Index

SLP Handle Functions
Functions used to open and close SLPHandles:
SLPOpen()
SLPClose()
	Parsing Functions
	Useful parsing functions:
	SLPParseSrvURL()
	SLPEscape()
	SLPUnescape()
	SLPFree()

Service Registration Functions
Functions that should be called by applications that provide a service:
SLPReg()
SLPDereg()
SLPDelAttrs()
	Configuration Functions
	Functions that may be called to determine the configuration of the SLP on the local machine and on the network:
	SLPGetRefreshInterval()
	SLPFindScopes()
	SLPGetProperty()
	SLPSetProperty()

Service Location Functions
Functions that may be called by applications that use services:
SLPFindSrvs()
SLPFindSrvTypes()
SLPFindAttrs()
	Miscellaneous Information
	Quick links to useful information:
	Callbacks
	Divergence from RFC 2614
	Error Codes
	Syntax
	Security

Prepared by: Caldera Systems Inc
Maintained by: openslp.org

(b) OpenSLP Programmer's Guide index.

Figure 4.4: SLP documentation indexes.

Despite afterwards OpenSLP will be mainly used in "programmer mode", the "user mode" is useful also during the development parts. It in fact allows users to use SLP through shell, providing a very good way to debug code which uses the programmer mode. Each feature contained in [5] is performed by the command **slptool**, while the SLP daemon defined in [8] can be controlled using the command **slpd**.
Unfortunately the project is no longer actively maintained, and this is a problem especially for what concerns the documentation; the current code is definitively not well described by docs.
During the development part of this thesis, almost every function contained in figure

4.4(b) has been used, and this unearthed some critical issues; for instance, SLPSetProperty() is defined by [8] but the documentation says that it is not implementable because of a critical issue in [18]. Althought at the end SLPSetProperty() has not been used in the final source code, at the beginning it seemed the only option to configure slpd (actually it set only the client's configuration parameters) and this brought to a deeper search in OpenSLP source code, in order to try to find a workaround. It came out that actually the function is implemented but works until SLPGetProperty() is called [19]. This puts in evidence the truthiness of the previous sentence.
An incomplete summary of the differences between OpenSLP and [8] can be found in the dedicated section of the programmes' guide [20].

4.3.3 OpenSLP: Python 3 integration

Now that OpenSLP has been chosen, it only remains to find a way to use it inside Python 3 code. A good starting point could be the Python 2 wrapper contained in OpenSLP GitHub repository, but actually that wrapper was discovered successively. During the search of a SLP Python library, it came out another unmentioned option different from PySLP: python-libslp by tsmetana [21]. This option has not been mentioned before because without knowing OpenSLP, it was not clear what it was actually; a Python library without any .py source file was quite strange. At the end, python-libslp is a well done Python wrapper of OpenSLP. It was written in Python 2, hence it needed to be adapted, and actually I did it in my python-libslp fork [22]. Despite tsmetana's repo was no more updated since 2013, after an issue opened by me he ported the code to Python 3 maintaining Python 2 backward compatibility. Since my repo doesn't provide backward compatibility and tsmetana is surely much skilled that me in Python, my repo can be considered already obsolete. There is only an issue with its repo which depends from the OS configuration [23].
Since python-libslp depends on OpenSLP, before to compile it, OpenSLP must be installed. It will now be reported the whole process necessary to have a working Python 3 importable module called **slp**.

OpenSLP installation

Since OpenSLP is no more actively maintained, it is no more possible to install it using apt-get command. In the user's guide there is a section dedicated to the installation of the library starting from the releases [24], but since the last release does not include the

latest commits, it has been chosen to compile and install the library from the source code.
The steps to be executed are:

```
user@host:~$ sudo apt-get update
user@host:~$ sudo apt -y install pkg-config bison automake flex libtool
user@host:~$ git clone https://github.com/openslp-org/openslp.git
user@host:~$ cd openslp/openslp/
user@host:~$ sudo ./autogen.sh
user@host:~/openslp/openslp$ sudo ./configure --prefix /usr
user@host:~/openslp/openslp$ sudo make
user@host:~/openslp/openslp$ sudo make install
```

Code 4.25: OpenSLP: compilation and installation steps.

At the end of the procedure, libslp should be installed in */usr*. It is important to underlay that there can be more dependencies than the ones present in the first line of code 4.25.

Python 3 slp module

Now it remains only to compile python-libslp and install the **slp** module:

```
user@host:~$ git clone https://github.com/tsmetana/python-libslp.git
user@host:~$ cd python-libslp/
user@host:~/python-libslp$ # if python --version is lower than 3.0, use
     a text editor to change PKG_CHECK_MODULES line in PKG_CHECK_MODULES
    (Python, python3 >= 3.0,,) inside configure.ac
user@host:~/python-libslp$ autoreconf -fvi
user@host:~/python-libslp$ autoconf
user@host:~/python-libslp$ sudo ./configure
user@host:~/python-libslp$ sudo make
user@host:~/python-libslp$ sudo make install
user@host:~/python-libslp$ cd src/
```

Now inside the current directory create a file *setup.py* containing:

```python
from distutils.core import setup
setup (name = 'slp',
       version = '0.1',
       author = "tsmetana",
       description = """SLP Library""",
       py_modules = ["slp"],
       packages=[''],
       package_data={'': ['slp.so']},
        )
```

It remains only to install the library:

```
user@host:~/python-libslp/src$ pip3 install . --user
```

Code 4.26: Python 3 **slp** module: compilation and installation steps.

To test if everything is correctly installed, try to execute ***import slp*** inside a python3 shell. There should not be any output, hence any error.

Chapter 5

SLP Integration in FORCH

Now that SLP is ready to be used as FORCH service discovery component, this chapter will cover the final implementation of the necessary code. Due to the size of the developed code, only the main abstraction entities (i.e. classes) will be described here. The complete source code is available on GitHub [25].

As mentioned at the end of section 2.4, FORCH is better represented by a service-centric data organization, and should be composed by independent modules as much as possible. For this reason the development process has been focused on two main objectives:

- (Re)definition of the high level abstractions and their hierarchy.
- Hide SLP as much as possible from the orchestrator's point of view.

The first point brought to rewrite the source code of *Resources/Services Database* FORCH module from the scratch, while the second point has been satisfied with a wrapper of OpenSLP.

The implemented code has been divided in four source files:

- **fo_service.py**, which provides the new service-centric based abstractions.
- **fo_servicecache.py**, which maps and provides the methods to access to the DA cache.
- **fo_slp.py**, which wraps OpenSLP and gives the access to it from the external code.

- **fo_zabbix.py**, which provides an abstraction of the Zabbix node entities.

At the end of the development process, the whole reported code has been included in a single module, hence it can be simply used in others .py files using: ***import forch***.

5.1 Service-centric abstraction entities

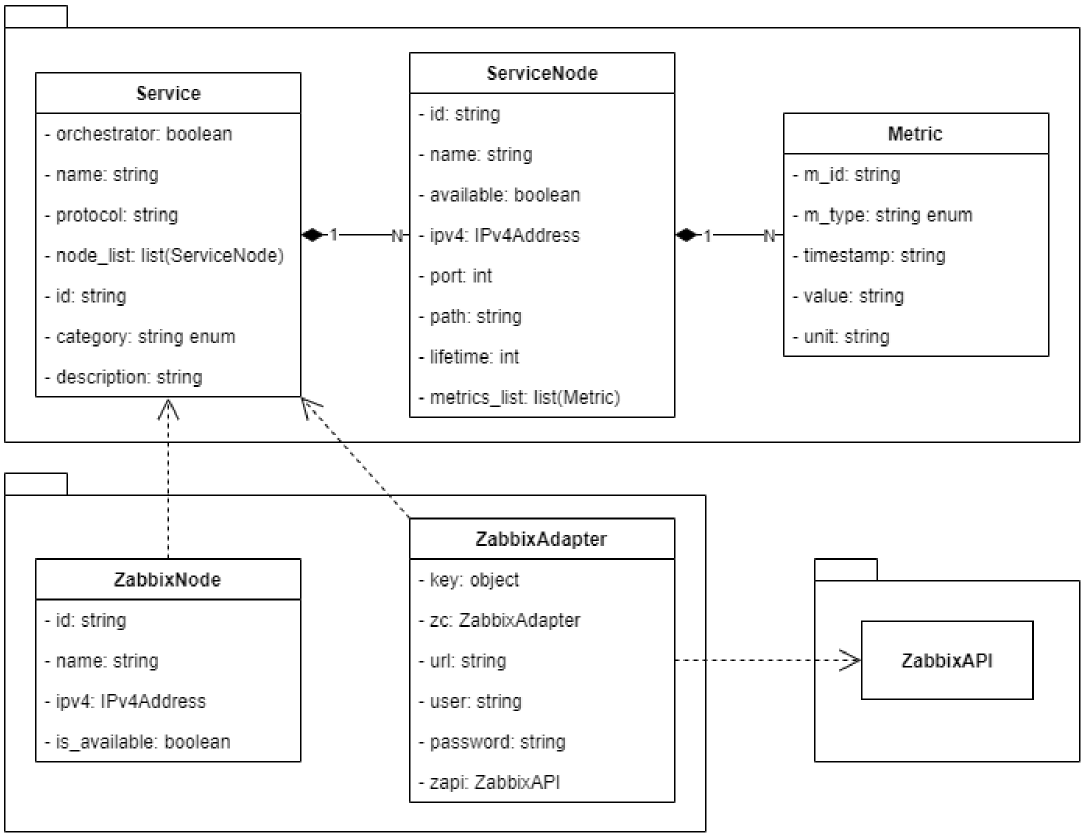


Figure 5.1: Class diagram of the main entities used in the service-centric structure.

Res/Serv Database module used Python dictionaries and JSON files as object, in fact it was not structured using classes and relative objects. Despite that, it can be said that the system was node-centric. It means that the orchestrator first looked for the nodes and then for their relative services. This structure was too low-level oriented and was based on Zabbix logic, hence it has been changed in order to fit well in a service based scenario like the fog networking. That choice allows to perform a logical decoupling between Zabbix and FORCH's code, that is always a good-practice from a software engineering

point of view.
Since Python is an object oriented program language, the module has been rewritten using proper classes to abstract the three main entities:

- Service: representing the services provided by the fog nodes.
- ServiceNode: representing the fog nodes.
- Metric: representing the fog node's metrics.

As can be seen in figure 5.1, metrics exist only given an owner node, while service nodes exits only given an owner service. The upper part of the figure is contained in fo_service.py, while the lower part is contained in fo_zabbix.py. ZabbixNode and ZabbixAdapter are two classes used to interface Zabbix with the new service-based logic. These two classes, and more generally fo_zabbix.py, were not required in node-centric logic, because it is the one used by Zabbix. Since fo_zabbix.py has not been written by me, for what concerns this thesis it is out of scope, hence it will not be reported.

5.1.1 Description of fo_service.py

Afterward it will be reported each relevant member of fo_service.py, but before to begin to describe each class and its methods, in order to further clarify the new service-based structure, it is better to provide the source code structure generated following the class diagram in figure 5.1.

```
class Service:
  def __init__(self, *, name:str="", protocol:str="",
  ↪  node_list:List[Service.__ServiceNode]|None=None, id:str="",
  ↪  category:ServiceCategory|None=None, description:str=""):
    ...
  ...

  class __ServiceNode:
    # 0xffff = slp.SLP_LIFETIME_MAXIMUM
    def __init__(self, *, id:str, ipv4:IPv4Address, name:str="",
    ↪  available:bool=False, port:int=0, path:str="", lifetime:int=0xffff,
    ↪  metrics_list:List[Service.__ServiceNode.__Metric]|None=None):
      ...
    ...

    class __Metric:
      def __init__(self, m_id:str="", m_type:MetricType=MetricType.CPU,
       ↪  timestamp:str="", value:str="", unit:str=""):
        ...
      ...
```

Code 5.1: Python 3 classes generated by the class diagram in figure 5.1. Since ZabbixNode and ZabbixAdapter are only dependencies, they miss because used in an untracked manner in Service methods.

In code 5.1 it can be seen that Metric is an inner class of ServiceNode, which is an inner class of Service. Despite this solution can seem too low flexible because the inner classes are private, it is the best way to represent the compositions in figure 5.1. Composition in fact supposes that Metric objects's life cycle is regulated by ServiceNode, and that ServiceNode objects's life cycle is regulated by Service.
It is important to say that the ***__eq__*** method has been overridden in each class, in order to consider two objects equal only if their ids are the same. Hence, two Services, ServiceNodes or Metrics are the same only if their ids are equal.

In the constructors of Service and Metric there are two non-basic data types: **ServiceCategory** and **MetricType**. These two types are simple enum classes used to limit *category* and *m_type* fields to a defined set of possible values. These classes are also extremely useful for code maintenance, because they allow an easily and controlled expansion of the values allowed for the related fields.

```
class ServiceCategory(Enum):
  IAAS = "FVE"     # Fog Virtualization Engine
  PAAS = "SDP"     # Software Development Platform
  SAAS = "APP"     # APPlication
  FAAS = "LAF"     # Lightweight Atomic Function
  NONE = "None"
```

Code 5.2: ServiceCategory enum class and its possible values.

```
class MetricType(Enum):
  CPU = "CPU utilization"
```

```
RAM = "Memory utilization"
```

Code 5.3: MetricType enum class and its possible values.

Now that the general class structure has been described, it will be reported the methods of the three described classes.

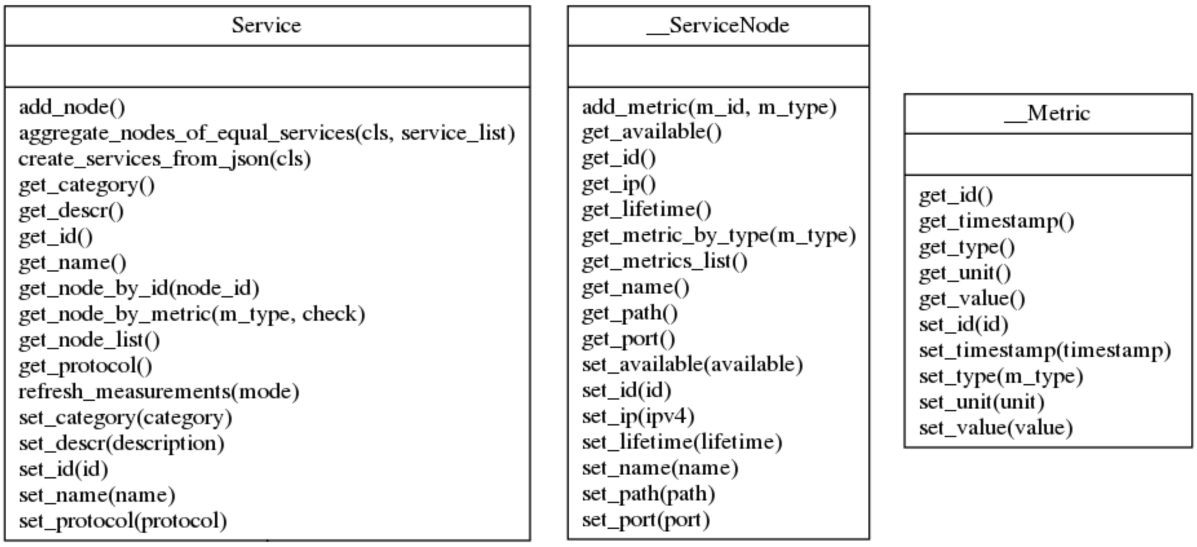


Figure 5.2: Methods of the three main service abstraction classes.

In order to respect encapsulation principle, each class has getters and setters methods for each of its private field. The only exceptions are:

- *node_list's setter* because since each data of a node must be cross-verified from Zabbix, a node must be added individually using *add_node()* method.
- *metric_list's setter* because of the same reason of the previous point. In this case the method to be used is *add_metric()*.

Service class

The non-trivial methods of Service are:

```python
def add_node(self, *, ipv4:IPv4Address, port:int=0, path:str="",
↪  lifetime:int=0xffff) -> str|None
```

This method adds a node in the node list of a service object. If Service's orchestrator static field is true, the node is added only if it is known by Zabbix. The method returns either the node id or *None* if it fails. The fail happens only if a node has not been successfully validated from Zabbix (of course, only if the validation is required).

```python
@classmethod
def aggregate_nodes_of_equal_services(cls, service_list:List[Service])
↪  -> List[Service]
```

This method finds and joins the nodes related to the same service. In practice, the method scans the input services list and, if it finds the same service in two different entries of the list, it aggregates them in only one. It returns the list with the aggregated services. The output list contains only one entry for each service.

```python
@classmethod
def create_services_from_json(cls, *, json_file_name:str|Path,
↪  ipv4:IPv4Address) -> List[Service]
```

This method takes in input a JSON file, an IPv4 and return a list with the services parsed from the input file. Code 4.19 shows the JSON services file structure to be parsed using PySLP, for what concerns OpenSLP the JSON file has been redefined. In particular, each SLP URL field has a dedicated JSON attribute, and the services are dividend in the three XaaS categories introduced in the second chapter.

```json
{
  "SAAS": {
    "APP001": {
      "thumbnail": "",
      "protocol": "http",
      "path": "",
      "port": "",
      "name": "httpd",
      "descr": "Apache web server",
      "lifetime": 65535
    },
    ...
  },
  ...
}
```

Code 5.4: Final JSON services file structure.

-
  ```
  def get_node_by_id(self, node_id:str) -> Service.__ServiceNode|None
  ```

This method returns the ServiceNode entry of node_list Service field with the id equal to the parameter node_id. If there is not a suitable entry in the list, the method returns *None*.

-
  ```
  def get_node_by_metric(self, m_type:MetricType=MetricType.CPU,
  ↪  check:str="min") -> Service.__ServiceNode|None
  ```

Given a MetricType, the method returns the node_list entry with either the maximum or the minimum value of that metric. If either there is not a suitable entry in the list, or *check* parameter is neither "min" nor "max", the method returns *None*.

-
  ```
  def refresh_measurements(self,
  ↪  mode:MeasurementRetrievalMode=MeasurementRetrievalMode.SERVICE) ->
  ↪  None
  ```

This method refreshes each metric of each ServiceNode present in node_list field of Service. It is not definitive, because it defines three different working modalities which bring to the same result in different ways. In practice the *mode* field sets the granularity of the Zabbix query necessary to refresh the metrics.

MeasurementRetrievalMode is the enum class that contains the allowed working modalities:

```python
class MeasurementRetrievalMode(IntEnum):
  SERVICE = 1
  NODE = 2
  METRIC = 3
```

- **SERVICE** mode performs only a single query to Zabbix in order to obtain every needed value in one shot.
- **NODE** mode performs a Zabbix query for each ServiceNode.
- **METRIC** mode performs a dedicated query for each Metric to refresh.

This method will be used in the results chapter to carry out some overhead measurements.

ServiceNode class

The non-trivial methods of ServiceNode are:

-
  ```python
  def add_metric(self, m_id:str, m_type:MetricType) -> None
  ```

  This method adds a metric into the *metrics_list* field of ServiceNode objects. It simply calls the Metric class constructor, and it is necessary because Metric class is private and cannot be instantiated outside ServiceNode class.

-
  ```python
  def get_metric_by_type(self, m_type:MetricType) ->
  ↪  Service.__ServiceNode.__Metric|None
  ```

  This method return the *metrics_list* entry given a specific *m_type*. If no entry is found, it returns *None*. Since given a node, Zabbix does not provide metrics of the same type, the method supposes that, for each MetricType, there is only a single MetricType entry in *metrics_list*. In case of more than one entry with the same MetricType, the method returns the first one found.

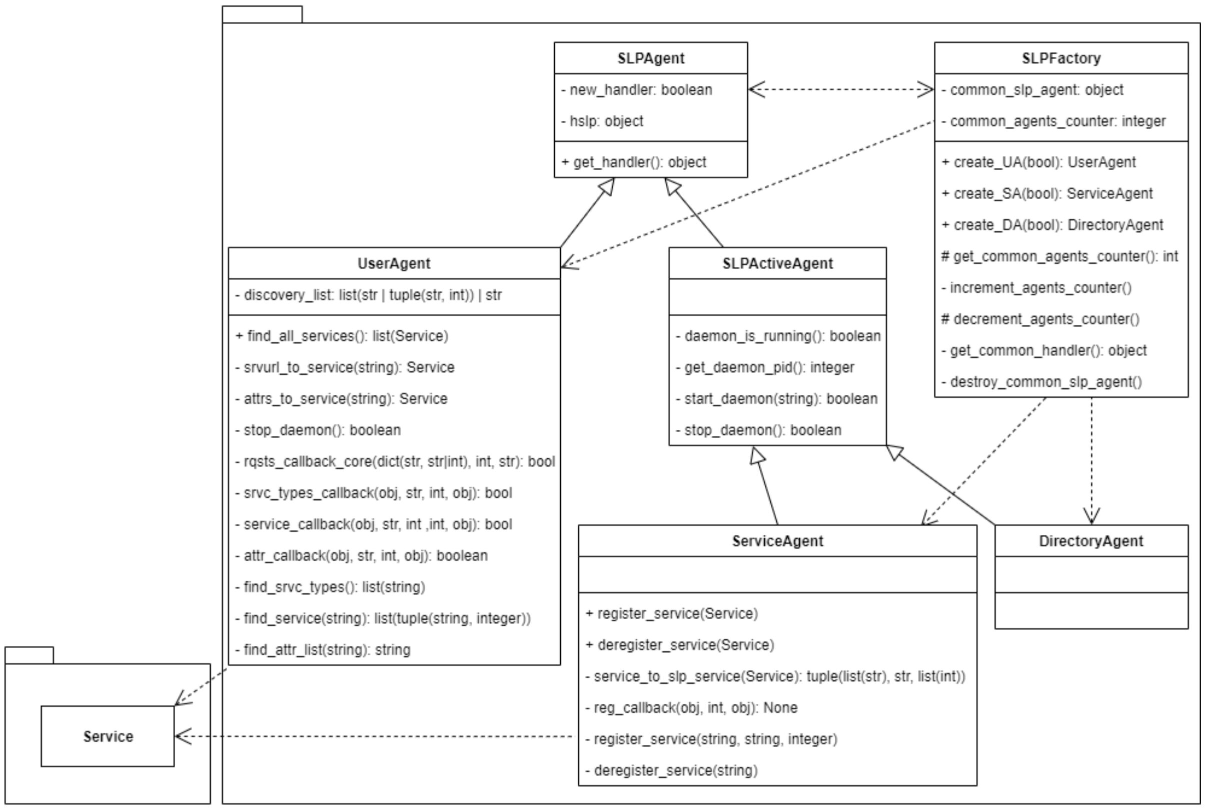
Metric class

As can be seen in figure 5.2, Metric class has only getters and setters methods, hence they will not be described.

5.2 OpenSLP wrapper

As previously mentioned, one of the main objectives of this implementation is to hide as much as possible SLP, providing an abstraction class. Service objects are supposed to be the fundamental entities which must be used and shared in the communication between FORCH modules. This means that the wrapper must perform a sort of translation between Services and SLP. File fo_slp.py actually is an adapter committed to perform the just mentioned translation.

5.2.1 Description of fo_slp.py



SLPAgent
- new_handler: boolean
- hslp: object
+ get_handler(): object
SLPFactory
- common_slp_agent: object
- common_agents_counter: integer
+ create_UA(bool): UserAgent
+ create_SA(bool): ServiceAgent
+ create_DA(bool): DirectoryAgent
# get_common_agents_counter(): int
- increment_agents_counter()
# decrement_agents_counter()
- get_common_handler(): object
- destroy_common_slp_agent()
UserAgent
- discovery_list: list(str | tuple(str, int)) | str
+ find_all_services(): list(Service)
- srvurl_to_service(string): Service
- attrs_to_service(string): Service
- stop_daemon(): boolean
- rqsts_callback_core(dict(str, str|int), int, str): bool
- srvc_types_callback(obj, str, int, obj): bool
- service_callback(obj, str, int ,int, obj): bool
- attr_callback(obj, str, int, obj): boolean
- find_srvc_types(): list(string)
- find_service(string): list(tuple(string, integer))
- find_attr_list(string): string
SLPActiveAgent
- daemon_is_running(): boolean
- get_daemon_pid(): integer
- start_daemon(string): boolean
- stop_daemon(): boolean
ServiceAgent
+ register_service(Service)
+ deregister_service(Service)
- service_to_slp_service(Service): tuple(list(str), str, list(int))
- reg_callback(obj, int, obj): None
- register_service(string, string, integer)
- deregister_service(string)
DirectoryAgent
Service

Figure 5.3: Class diagram of fo_slp.py.

Figure 5.3 shows SLP wrapper from the logical level point of view. Since OpenSLP use common handler over each type of agent, to hide OpenSLP complexity, the agent objects cannot be instantiated directly, but only using **SLPFactory** class. SLPFactory is a class committed to create and if needed manage the agent objects. All its public methods are static.

SLP agents are implemented using the inheritance hierarchy shown in figure 5.3, which allows the reuse of code portions all over the agent classes.

- **SLPAgent** is the first class of the hierarchy and provides each object of a SLP handler object (*hslp* field). The field *new_handler* tracks if the handler has been created by the current agent. The main objective of this class is to provide the base constructor and destructor which will be inherited by every agent object.
- **UserAgent** is the class that provides the [8] client functions related to User Agents entities, i.e. the features that do not require to run the SLP daemon.
- **SLPActiveAgent** is the class that provides the common slpd management functions needed by SA and DA. Due to the lack of slpd programmer API already commented in section 4.1, its methods make a large use of shell calls.
- **ServiceAgent** is the class that provides the client functions that need a running SLP daemon to be executed.
- **DirectoryAgent** is the class that abstracts DA entities. It does not include specialized class members, but overrides the constructor in order to start slpd in the proper way.

This classes have been structured following figure 5.3, in terms of code this means:

```
class SLPFactory:
  ...

  class __SLPAgent:
    ...

  class __SLPActiveAgent(__SLPAgent):
    def __init__(self, slp_handler:object|None=None):
      ...
    ...

  class __UserAgent(__SLPAgent):
    ...

  class __ServiceAgent(__SLPActiveAgent):
    ...

  class __DirectoryAgent(__SLPActiveAgent):
    ...
```

Code 5.5: Python 3 classes generated by the class diagram in figure 5.3.

Now that the logical structure has been described, afterwards it will be reported the main relevant methods of each class.

SLPFactory class

SLPFactory non trivial methods are:

```
@classmethod
def create_UA(cls, new_handler:bool=False)

@classmethod
def create_SA(cls, new_handler:bool=False)

@classmethod
def create_DA(cls, new_handler:bool=False)
```

This methods are the factory methods that create respectively UA, SA and DA objects. The *new_handler* parameter specifies if a new dedicated SLP handler must be created. Different SLP handles allows to execute OpenSLP functions in concurrent mode. SLP agents with a common handler can execute their instructions only sequentially. In FORCH, the multiple-handler-feature is not used.

```python
@classmethod
def __get_common_handler(cls) -> object
```

This method returns, or creates and returns a SLP handler used to create every agent that does not need a dedicated one. The common handler is saved in *common_slp_agent* field.

```python
@classmethod
def __increment_agents_counter(cls)

@classmethod
def _decrement_agents_counter(cls)
```

These methods respectively increment and decrement the number of agents created with the common handler. The counter variable is the *common_agents_counter* field, and when it reaches zero, the common handler is destroyed in order to free resources.

```python
@classmethod
def __destroy_common_slp_agent(cls)
```

This method destroy the common SLP handler.

SLPAgent class

SLPAgent class does not have custom methods, hence it will not be described, because the relevant part is the implementation of the constructor and the destructor ([25]).

UserAgent class

UserAgent methods are:

```python
def find_all_services(self) -> List[Service]
```

This is the only public method and returns a list with all the found services in the network.

```python
@staticmethod
def __srvurl_to_service(srvurl:str) -> Service
```

This method is a private utility method that converts a SLP service url string (srvurl) to a Service object.

```python
@staticmethod
def __attrs_to_service(attrs_str:str) -> Service|None
```

This method is a private utility method that convert a SLP attributes list string (attrs) to a Service object.

```python
def __find_srvc_types(self) -> List[str]

def __find_service(self, service_type:str) -> List[Tuple[str, int]]

def __find_attr_list(self, srvurl:str) -> str
```

These three methods wrap the OpenSLP functions. They wrap respectively: *SLPFindSrvTypes()*, *SLPFindSrvs()* and *SLPFindAttrs()*.

```python
@staticmethod
def __srvc_types_callback(h:object, srvc_type:str, errcode:int,
↪   cookie_data:object) -> bool

@staticmethod
def __service_callback(h:object, srvurl:str, lifetime:int, errcode:int,
↪   cookie_data:object) -> bool

@staticmethod
def __attr_callback(h:object, attrs:str, errcode:int,
↪   cookie_data:object) -> bool
```

These methods are the respective callbacks of the previous three methods reported. They populate the variable returned by the above methods.

```
@staticmethod
def __rqsts_callback_core(param_dict:Dict[str,str|int], errcode:int,
↪  rqst_type:str="") -> bool
```

This method is the core of the callback request methods. This means that it contains the code common to the previous three reported callback methods. Each callback method performs a proper invocation to this one.

SLPActiveAgent class

SLPActiveAgent methods are:

```
@classmethod
def __daemon_is_running(cls) -> bool
```

This method returns True if there is one slpd instance in execution.

```
@staticmethod
def __get_daemon_pid() -> int|None
```

This method return slpd PID if it is running, otherwise None. If more than one slpd instances are found, the method rises an exception.

```
@classmethod
def __start_daemon(cls, optns:str="") -> bool
```

This method starts slpd using the terminal options specified in *optns* input parameter. If it finds another running slpd, it fails and returns False, otherwise True.

```
@classmethod
def __stop_daemon(cls) -> bool
```

This method finds, stops slpd instance and returns True. If no instance is found, it returns False.

ServiceAgent class

ServiceAgent methods are:

```
def register_service(self, service:Service)

def deregister_service(self, service:Service)
```

These methods are the only public methods and respectively register and deregister a given Service object passed in *service* parameter.

```
def __register_service(self, srvurl:str, attrs:str="",
↪  lifetime:int=slp.SLP_LIFETIME_DEFAULT)

def __deregister_service(self, srvurl:str):
```

These two methods wrap the OpenSLP functions. They wrap respectively: *SLPReg()* and *SLPDereg()*.

```
@staticmethod
def __reg_callback(h:object, errcode:int, data:object) -> None
```

This is the callback method used during service registration. It returns always None and is used to print eventual errors.

```
@staticmethod
def __service_to_slp_service(service:Service) -> Tuple[List[str], str,
↪ List[int]]
```

This is an utility method that takes in input a service and returns:

- a service url list with one srvurl for each node in the service's *node_list* field
- an attribute list string
- a integer list containing the lifetimes associated to each srvurl list. Each lifetime list entry index is associated with the same srvurl list entry index.

DirectoryAgent class

As already mentioned, DirectoryAgent class simply overrides its father constructor in order to start slpd as DA. The new constructor specifies that slpd must be started with slp_DA.conf. Inside the same directory of fo_slp.py there is also slp_SA.conf, which is dedicated to start the SAs slpd. The two files differ only for the *net.slp.isDA* setting, which in slp_DA.conf is set to true.
DirectoryAgent code is very short, hence afterward it will be reported the whole class:

```
class __DirectoryAgent(__SLPActiveAgent):
  def __init__(self, slp_handler:object=None):
    slp.SLPSetProperty("net.slp.isDA", "true") # Maybe useless, but it sets
    ↪ correctly the global environment
    super().__start_daemon("-c
    ↪ {}".format(str(Path(__file__).parent.joinpath("slp_DA.conf"))))
    super().__init__(slp_handler)
```

Code 5.6: DirectoryAgent class code.

5.3 DA cache abstraction

At this point, all the tools needed by FORCH are ready. It remains only to provide a sort of database where the orchestrator can find the network services. Since SLP is used in centralized mode, and the DA is the orchestrator itself, this database is already inside it, but it is not directly accessible because is hidden inside OpenSLP. In order to read it, it is necessary to instantiate an UA which, using unicast loopback SLP request messages, obtains the needed information by the local DA.

5.3.1 Description of fo_servicecache.py

What said before has been done inside *fo_servicecache.py* file, which defines a **ServiceCache** class containing a list of Services and a UserAgent object. Since the file is very short and elementary, a complete analysis like the ones done for the other source files would be superfluous, hence afterwards it will be reported only the whole code.

ServiceCache class

```python
from typing import List
from forch.fo_service import Service
import copy
from .fo_slp import SLPFactory

class ServiceCache:
  def __init__(self, *, refresh=False):
    self.__slp_ctrl: SLPFactory.__UserAgent = SLPFactory.create_UA()
    self.__service_list: List[Service] = []
    if refresh:
      self.refresh()

  def get_list(self, *, deepcopy:bool=False) -> List[Service]:
    if deepcopy:
      return copy.deepcopy(self.__service_list)
    return self.__service_list

  def clear(self):
    self.__service_list = []

  def refresh(self):
    self.clear()
    self.__service_list = self.__slp_ctrl.find_all_services()
```

Code 5.7: Code contained in fo_servicecache.py.

The orchestrator is supposed to instantiate a single ServiceCache object to be used to know and keep track of the available nodes and their offered services.

Chapter 6

Performance Evaluation

The code is now fully developed and ready to be evaluated in terms of generated overhead. In order to test the implementation performances three kinds of macro-measurements have been carried out:

- Time, number of packets and packets size generated by services discovery, versus the number of services announced by a single node. This measurement has been performed in both centralized and distributed SLP cases.
- Time, number of packets and packets size generated by service discovery, versus the number of nodes on the network. Each node is supposed to announce only one service. This measurement has been performed in both centralized and distributed SLP cases.
- Time, number of packets and packets size generated by metrics refresh, versus the number of metrics to refresh giving a single node. This measurement has been performed in each of the three *MeasurementRetrievalMode* modalities (section 5.1.1).

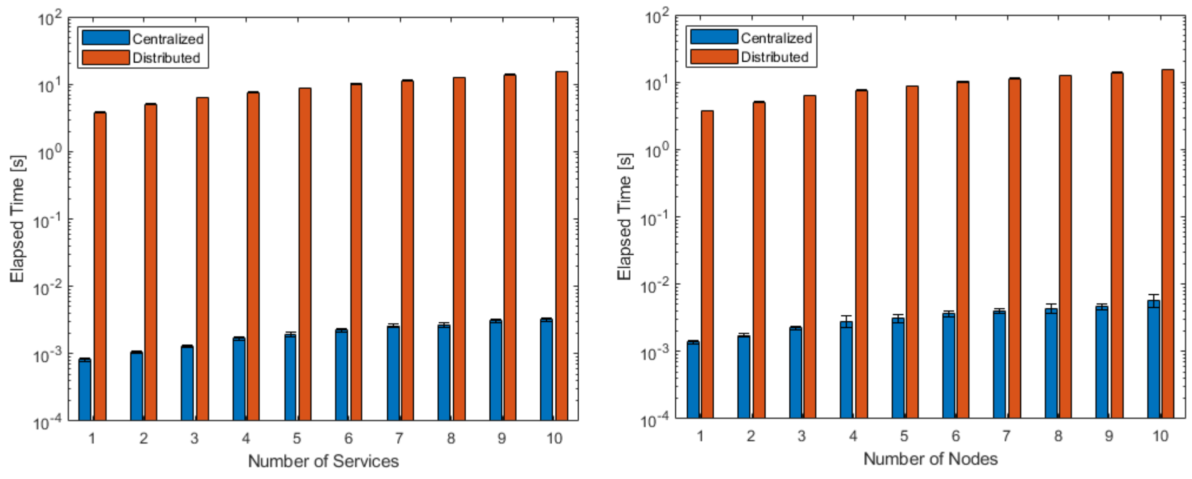
In order to have reliable statistic data, each measurement has been performed one hundred times and averaged. The resulting plots show the found mean values and the related confidence intervals.
The used test scripts can be found under tests folder inside [25] repository. They will not be reported because they simply use the code described in this thesis. The used network packet analyzer was not the well-known WireShark, but is Scapy. Scapy is a huge Python network utility library that allows to sniff packets using Python code. This is a great feature, but is less advanced than WireShark, hence attention must be payed.

For example, if it sniffs on loopback interface, the packets are captured twice: when they leave/arrive from/to the interface.

6.1 Service Search Tests

In this section, the first two macro-measurements will be compared. The tests have been performed in both SLP working modalities, this means that results with a huge difference of generated data will be compared. Since the centralized mode produces less data than the distributed one, its confidence intervals are higher, hence less precise. In the distributed mode, the amount of generated data is so huge that the confidence intervals disappear because too short.

Elapsed search time



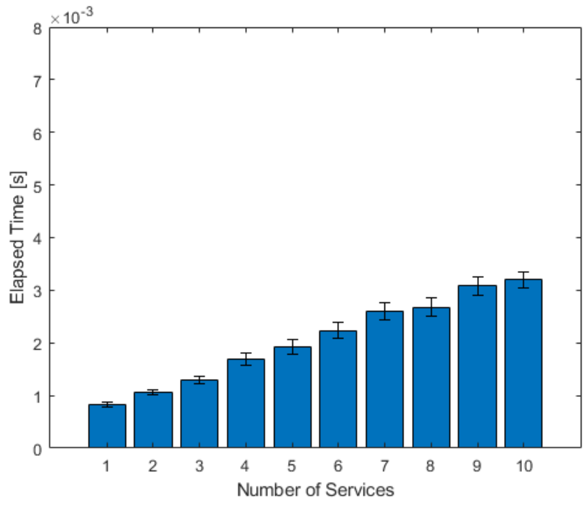
(a) Services discovery time versus the number of services.

(b) Services discovery time versus the number of nodes.

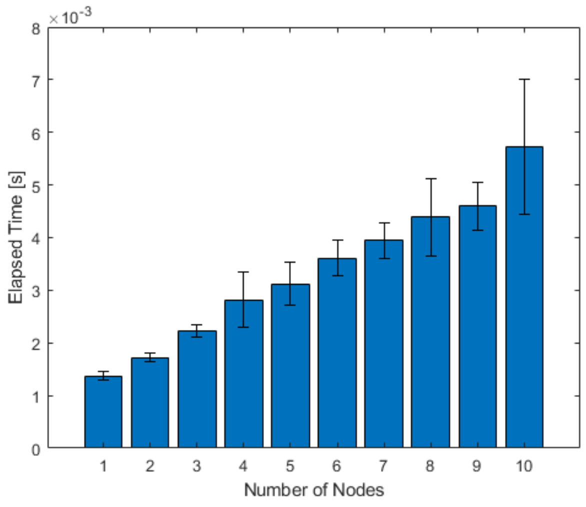
Figure 6.1: Time needed to find network services as a function of either the number of services or nodes. The figure includes both centralized and distributed working modalities.

Figure 6.1 shows the time needed to discover the available network services as a function of the number of either services (6.1(a)) or nodes (6.1(b)). In both the subfigures it is evident that the centralized mode is much faster than the distributed one. The difference is so high, that in order to see the blue bars, it has been necessary to use the logarithmic scale y axis. This is due to the fact that, if a DA is present, every SA announces its services in unicast, and the orchestrator's UA reads the DA cache on the loopback interface. In this way multicast is no more necessary, and the discovery process is less time consuming. In centralized mode, the minimum and maximum elapsed times of 6.1(a) are 83 μs and 320 μs, while in 6.1(b) they are 137 μs and 573 μs. In distributed mode, the minimum and maximum elapsed times of 6.1(a) are 3.788 s and 15.162 s, while in 6.1(b) they are 3.795 s and 15.142 s. This means that in terms of time the best operating mode is the centralized one, and it is better to have services deployed by a low number of nodes.

For sake of completeness, in figures 6.2 and 6.3 the four plots contained in figure 6.1 are singly reported in linear scale.

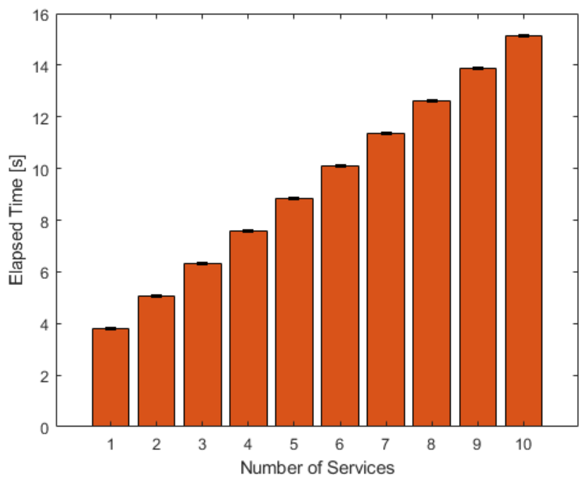


(a) **Centralized mode**: services discovery time versus the number of services.

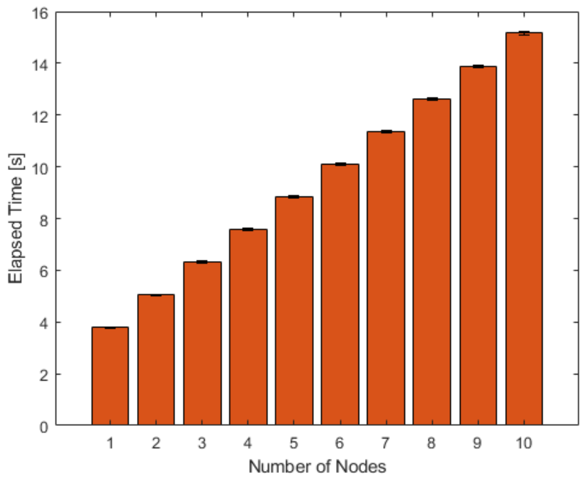


(b) **Centralized mode**: services discovery time versus the number of nodes.

Figure 6.2: Time needed to find network services as a function of either the number of services or nodes. The figure is in linear scale and refers only to the centralized mode.



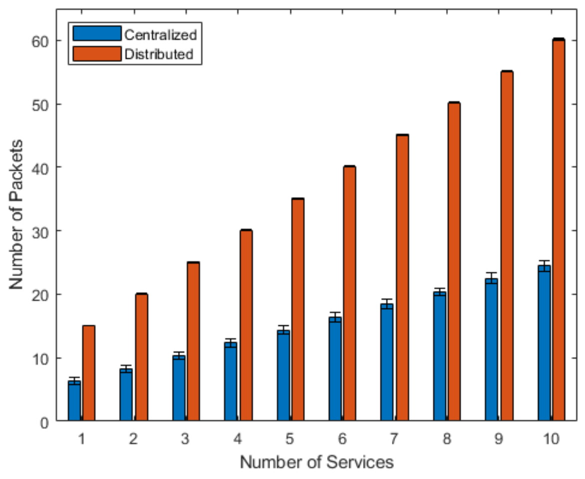
(a) **Distributed mode**: services discovery time versus the number of services.



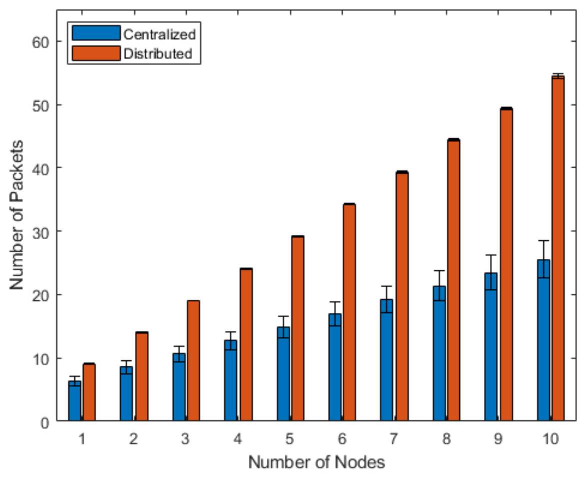
(b) **Distributed mode**: services discovery time versus the number of nodes.

Figure 6.3: Time needed to find network services as a function of either the number of services or nodes. The figure is in linear scale and refers only to the distributed mode.

Amount of generated packets during discovery



(a) Number of generated packets versus the number of services.



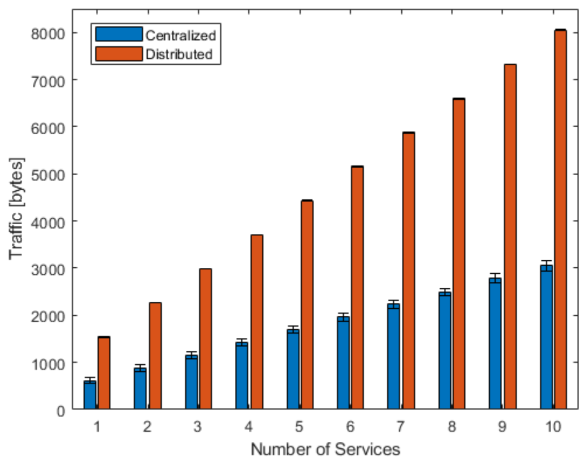
(b) Number of generated packets versus the number of nodes.

Figure 6.4: Amount of generated packets to find network services as a function of either the number of services or nodes. The figure includes both centralized and distributed working modalities.

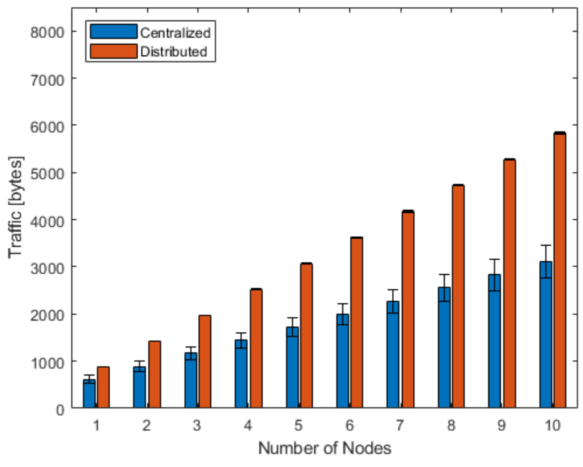
Figure 6.4 shows the number of packets generated during the discover of the available network services. As before the independent variable is the number of either services (6.4(a)) or nodes (6.4(b)). In both the subfigures it is evident that the centralized mode produces less packets than the distributed one. The reason is the same of the previous case. In centralized mode, the minimum and maximum number of generated packets in 6.4(a) are 6.31 and 24.48, while in 6.4(b) they are 6.41 and 25.58. In distributed mode, the minimum and maximum elapsed times of 6.4(a) are 15.06 and 60.18, while in 6.4(b) they are 9.06 and 54.55. It is evident that it is better to use the centralized mode, but in the distributed one it is better to have few services deployed by more nodes. It has been already put in evidence that the centralized mode produces less packets, but it is important to underline that these packets are not real traffic, because they are captured on the loopback interface. The only real network packets are generated at the first iteration of the test, due to DA announcement and the data exchange with SAs needed to learn the available services. The number of the non-loopback packets goes from 17 for the case with one service, up to 36 when ten services are announced. Varying the number of nodes, the quantity of the non-loopback packets goes from 29 with one node, up to 138 when ten nodes are present. Even if the real centralized network overhead is lower varying services instead of nodes, in both cases it is very small.

Total packets size during discovery

Figure 6.5 shows the total byte size of the packets generated during the discover of the available network services. The x axis variable is still the number of either services (6.5(a)) or nodes (6.5(b)). Also in this case, the centralized mode is the one that produces less overhead, and mainly it is fake because is on the loopback interface. In centralized mode, the minimum and maximum number of generated packets in 6.5(a) are 609.87 *bytes* and 3052.26 *bytes*, while in 6.5(b) they are 615.18 *bytes* and 3110.53. In distributed mode, the minimum and maximum elapsed times of 6.5(a) are 1539.74 *bytes* and 8049.22 *bytes*, while in 6.5(b) they are 876.74 *bytes* and 5838.89 *bytes*. Comparing this figure with the previous one, it can be seen if there is a trend of the packets to increase their size. Differently from figure 6.5, in the distributed case the values contained in the two plots of figure 6.4 are quite similar. In this case, it can be deduced that if a node announces a certain number of services (6.5(a)), the average size of the generated packets is higher than the case with one service deployed by the same number of nodes (6.5(b)). Indeed, the average packet sizes in distributed mode are 144.27 bytes per packet varying service number, and 109.08 bytes per packet varying node.



(a) Size of generated packets versus the number of services.



(b) Size of generated packets versus the number of nodes.

Figure 6.5: Size of generated packets to find network services as a function of either the number of services or nodes. The figure includes both centralized and distributed working modalities.

As far as this case is concerned, centralized mode is still the most performing one, and like the previous measurement, in distributed mode it is still better to avoid to concentrate an high number of services in a low number of nodes.

6.2 Refresh Metrics Test

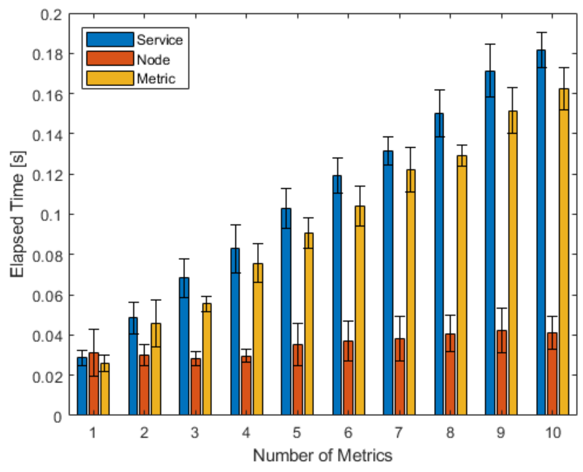
In this section it will be reported the last performed macro-measurement, which aims to find the best way to query Zabbix in order to lower the overhead during the refresh of the metrics. The test involved one orchestrator node, one service node and has been performed varying the number of metrics to refresh: from 1 to 10. The service node was supposed to announce only one service.
Since the orchestrator implements the Zabbix server with which it communicates using REST API, the traffic has been captured on the port 80 of the loopback interface.
As already mentioned there are three mainly modalities to query Zabbix:

- **SERVICE**: which performs a single Zabbix filtered query which asks for the metric values of each node associated to a given service.
- **NODE**: for each node associated to a given service, this mode performs a single Zabbix non-filtered query which asks for the required node metric values. This

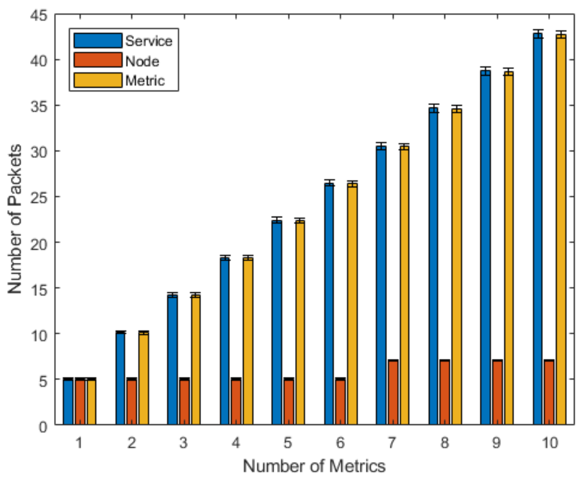
mode differs from SERVICE because iterates over nodes, hence performs a number of queries equal to the number of service nodes.

- **METRICS**: which iterating over each metric of each node of a given service, performs a non filtered query which asks for a single metric at a time.

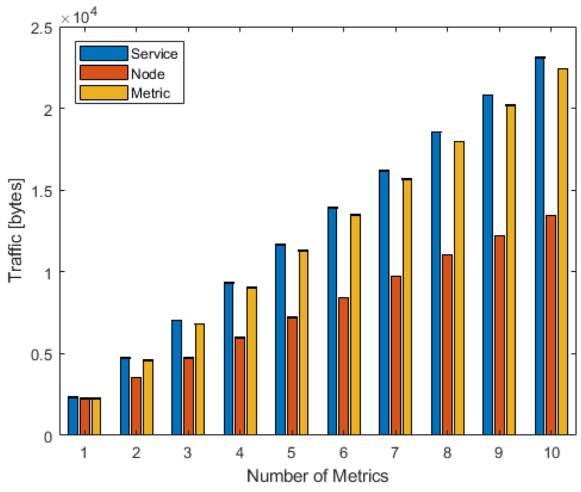
Figure 6.6 immediately shows that the most efficient mode is the second one (i.e. the node base mode). In every subfigure it dominates over the service and the metric modes. This probably is due to internal Zabbix data organization which is probably node based. Against every prevision, the worst performances are obtained using the first mode. It can be supposed that service based mode requires more time because the computational power required to filter the high amount of data. Since the number of packets is almost equal to the metric based mode, probably the filtering is done by the client. The last sentence can be confirmed also by the figure 6.6(c), which shows that the total size of the data received in service mode is the highest one (i.e. the sent data are not filtered at server side). The metric case works as expected; the amount of sent packets increases with the amount of the metrics required.
As already mentioned, the incredibly high efficiency of the node mode can be explained by the internal Zabbix structure. Thinking at lower level, probably Zabbix memorizes the nodes in consecutive blocks of RAM, allowing the fastest possible way to access to the memory. This means also that is sufficient to read the memory allocated to each node and send it with consecutive TCP packets. This hypothesis is also supported by the fact that in 6.6(b) the node traffic is constant and at a certain point it has a step-increase, while in 6.6(c) there is a linear increase of the exchanged data size. The time needed to refresh the metrics in node mode goes from 31 *ms* to 41.3 *ms*, while the minimum and the maximum number of generated packets are 5 and 8. Regarding the amount of generated data, the minimum is 2262 *bytes*, while the maximum is 13448 *bytes*.



(a) Elapsed refresh time versus the number of metrics.



(b) Number of generated packets versus the number of metrics.



(c) Size of generated packets versus the number of metrics.

Figure 6.6: Metrics refresh performances as a function of the number of metrics to refresh. The measurements have been performed in every *MeasurementRetrievalMode*.

Chapter 7

Conclusion

In general, the work behind this thesis has analytically inspected and tried to find a concrete solution to the problem of service discovery in local networks. Although the solution has been applied to a specific project, *the fog orchestrator FORCH*, both the analysis and the implementation can be adopted in other contexts where service discovery is needed. Before this work, FORCH only supported static service configuration, whereas now it features a completely autonomous service discovery module based on SLP protocol. The code produced for this thesis was successfully integrated within FORCH, renovating its most prominent module, which oversees performing service discovery, among other functions. In this way now FORCH is based on a service-centric logic, as opposed to the previous node-centric approach, which helps to make service deployment more scalable, reliable and efficient, and, in more practical words, greatly facilitates code maintenance, a key feature for a project that aims to further evolve in the future. I am not a software engineer, but I tried to implement the code following the software engineering principles as much as possible, from SOLID principles to the well-known design patterns. Due to the low complexity of the code and the Python language features, there are no interfaces that satisfy the program-to-an-interface principle. I focused more to the correct application of the single responsibility principle, the open/close principle and the Liskov substitution principle. Some applied design patterns, like singleton, factory method and adapter, can be found in the code as well. This was my first time trying to write good-quality code, and the fact that subsequently, modifying the code and operating bug fixing came rather easily, suggests that the code is reasonably well written. Overall, the integration of the new code with the codebase of FORCH improved the functionalities of the orchestrator, which gives credit to the quality of the code.

7.1 Future improvements

Unfortunately, there are still some unsolved issues, therefore I would like to suggest some future improvements that could further enhance my work.

- **Issue**: As already mentioned, slpd is managed using shell invocations through Python libraries, but Python does not allow such kind of invocations when it is shutting down. This means that the code cannot stop the SLP daemon during the on-exit script objects destruction. In practice, the developed module will probably be terminated only on system shutdown, hence this is not a critical issue, but it is better to remind that slpd must be terminated manually using ***sudo pkill -15 slpd***. A possible solution is to do not detach the daemon from the parent shell and rely on the automatic child process termination when the parent dies. I tried this possibility without success, maybe more expert Python programmers can find the solution.

- **Feature**: Since currently FORCH uses only numeric IP addresses, the ServiceNode class lacks a literal URL string which could be used to announce services that can be reached using a DNS query. This feature is supported by SLP, but since it is not used, it has been omitted. The FORCH architecture does not constraint the orchestrator location, hence in future this feature could be really useful.

- **Feature**: SLP has a lot of not used and not implemented features such as scopes and security authentication. Right now it is not known if these features will be useful, but for sake of generality the Python wrapper should support them.

As far as my work is concerned, these are the main suggested current improvements, but as often happens in these situations, additional features will be discovered only when they will be effectively necessary.

Bibliography

[1] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*, pages 103–130. Springer, 2018.

[2] G. Davoli, D. Borsatti, D. Tarchi, and W. Cerroni. Forch: An orchestrator for fog computing service deployment. In *2020 IFIP Networking Conference (Networking)*, pages 677–678, 2020.

[3] Microsoft Azure. What is paas? platform as a service. `https://azure.microsoft.com/en-us/overview/what-is-paas/`. (Accessed on 19/01/2021).

[4] Charles E. Perkins John Veizades, Erik Guttman and Scott Kaplan. Service Location Protocol. RFC 2165, RFC Editor, June 1997.

[5] John Veizades Erik Guttman, Charles Perkins and Michael Day. Service Location Protocol, Version 2. RFC 2608, RFC Editor, June 1999.

[6] Erik Guttman. Vendor Extensions for Service Location Protocol, Version 2. RFC 3224, RFC Editor, January 2002.

[7] James Kempf Erik Guttman, Charles E. Perkins. Service Templates and Service: Schemes. RFC 2609, RFC Editor, June 1999.

[8] James Kempf and Erik Guttman. An API for Service Location. RFC 2614, RFC Editor, June 1999.

[9] Seliverstov Maksim | PyPI. pyslp. `https://pypi.org/project/pyslp/`. (Accessed on 08/12/2020).

[10] Seliverstov Maksim. pyslp — github. `https://github.com/maksim-v-seliverstov/pyslp/tree/master/pyslp`. (Accessed on 08/12/2020).

[11] Wikipedia. CPython. `http://en.wikipedia.org/w/index.php?title=CPython&oldid=991877702`, 2020. [Online; accessed 08-December-2020].

[12] Python 3.8.7rc1 documentation. Extending python with c or c++. `https://docs.python.org/3.8/extending/extending.html`. (Accessed on 08/12/2020).

[13] Markus Alexander Kuppe Jan S. Rellermeyer. jslp - java slp (service location protocol) implementation | maven. `http://jslp.sourceforge.net/`. (Accessed on 08/12/2020).

[14] Eclipse Foundation: Eclipse Communication Framework. Commits history of protocols/bundles/ch.ethz.iks.slp | github. `https://github.com/eclipse/ecf/commits/master/protocols/bundles/ch.ethz.iks.slp`. (Accessed on 08/12/2020).

[15] Eclipse Foundation: Eclipse Communication Framework. Repository: org.eclipse.ecf » ch.ethz.iks.slp | maven. `https://mvnrepository.com/artifact/org.eclipse.ecf/ch.ethz.iks.slp`. (Accessed on 08/12/2020).

[16] OpenSLP.org. About openslp. `http://www.openslp.org/`. (Accessed on 08/12/2020).

[17] OpenSLP.org. Commits history of openslp-org/openslp. `https://github.com/openslp-org/openslp/commits/master`. (Accessed on 08/12/2020).

[18] OpenSLP.org. Openslp programmers' guide: Slpsetproperty. `http://www.openslp.org/doc/html/ProgrammersGuide/SLPSetProperty.html`. (Accessed on 09/12/2020).

[19] OpenSLP.org. Master branch: openslp/libslp_property.c | github. `https://github.com/openslp-org/openslp/blob/master/openslp/libslp/libslp_property.c`. (Accessed on 09/12/2020).

[20] OpenSLP.org. Openslp programmers guide - divergence from rfc 2614. `http://www.openslp.org/doc/html/ProgrammersGuide/Divergence.html`. (Accessed on 09/12/2020).

[21] Tomas Smetana: tsmetana. tsmetana/python-libslp: Openslp api python binidngs | github. `https://github.com/tsmetana/python-libslp`. (Accessed on 09/12/2020).

[22] Mario Valieri: mrv96. mrv96/python-libslp: Openslp api python bindings. `https://github.com/mrv96/python-libslp`. (Accessed on 09/12/2020).

[23] Mario Valieri: mrv96. Porting to python3 - issue #3 - tsmetana/python-libslp | github. `https://github.com/tsmetana/python-libslp/issues/3#issuecomment-691456912`. (Accessed on 09/12/2020).

[24] OpenSLP.org. Openslp users guide - installing openslp on linux. `http://www.openslp.org/doc/html/UsersGuide/Installation.html`. (Accessed on 09/12/2020).

[25] Gianluca Davoli: giditre. Mrv96-thesis tag: giditre/unibo_gaucho | github. `https://github.com/giditre/unibo_gaucho/tree/mrv96-thesis/pyforch`. (Accessed on 17/01/2021).