Action full title:

Universal, mobile-centric and opportunistic communications architecture

Action acronym:

UMOBILE

Deliverable:

D4.4 “Set of QoS interfaces and algorithms”

Project Information:

Project Full Title	Universal, mobile-centric and opportunistic communications architecture
Project Acronym	UMOBILE
Grant agreement number	645124
Call identifier	H2020-ICT-2014-1
Topic	ICT-05-2014 Smart Networks and novel Internet Architectures
Programme	EU Framework Programme for Research and Innovation HORIZON 2020
Project Coordinator	Prof. Vassilis Tsaoussidis, Athena Research Center

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

Deliverable Information:

Deliverable Number-Title	D4.4 Set of QoS interfaces and algorithms
WP Number	WP4
WP Leader	Rute Sofia (SENCEPTION)
Task Leader (s)	Carlos Molina-Jimenez (UCAM)
Authors	**UCAM**: Adisorn Lertsinsrubtavee, Carlos Molina-Jimenez **ATHENA**: Sotiris Diamantopoulos, Christos-Alexandros Sarros **UCL**: Sergi Rene, Ioannis Psaras
Contact	Carlos.Molina@cl.cam.ac.uk
Due date	M30: 31/07/2017
Actual date of submission	31/07/2017

Dissemination Level:

PU	Public	
CO	Confidential, only for members of the consortium (including the Commission Services)	
CI	Classified, as referred to in Commission Decision 2001/844/EC	

Document History:

Version	**Date**	**Description**
Version 0.1	15/06/2017	First draft to the consortium
Version 0.2	9/ 07 /2017	Second draft to the consortium after receiving contributions from partners
Version 0.3	14/ 07/2017	Third draft to the involved partners for alignment with D3.2 and D3.4
Version 0.4	20/07/2017	Final version

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

Table of Contents

Table of Contents 3
List of figures 4
List of tables 4

EXECUTIVE SUMMARY 5

1. INTRODUCTION 6

2. UMOBILE ARCHITECTURE 9

3. QOS MECHANISMS 10
3.1 Application level QoS mechanisms 10
3.2 Network level QoS mechanisms 10
3.3 Layered QoS mechanisms 11
3.4 UMOBILE QoS mechanisms 11

4. SERVICE MIGRATION PLATFORM 12
4.1 Reactive and proactive (pre-fetching) service migration 14
4.1.1 Reactive service migration 15
4.1.2 Proactive service migration (pre-fetching) 15
4.1.3 Introductory example 15
4.1.4 Decision engine algorithms and interfaces 16

5 DTN FRAMEWORK 18
5.1 Architecture and functionality of the DTN framework 18
5.2 DTN integration interfaces 19

6 FLOWLET CONGESTION CONTROL 19
6.1 Architecture and functionality of the INRPP congestion control 19
6.2 INRPP integration interfaces 22

7 EVALUATION OF QOS-RELATED PARAMETERS 23
7.1 Experiment settings 23
7.2 Impact of the number of concurrent requests on QoS 24
7.3 Impact local container replication 26
7.4 Local vs Remote Container Replication 29
7.5 Container replication cost 31

8 FULFILMENT OF QOS REQUIREMENTS 32
8.1 Handling numbers of concurrent requests with local replication 36
8.2 Meeting of availability requirements with DTN support 41
8.2.1 QoS requirements and scenario 42
8.2.2 Service migration-DTN integration architecture 43
8.2.3 Practical demonstration 44
8.2.4 Discussion of results 46
8.3 Meeting of availability requirements with INRPP support 47
8.3.1 Service migration-INRPP integration architecture 47
8.3.2 QoS requirements and scenario 48
8.3.3 Service migration-INRPP evaluation results 49

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

9 CONCLUDING REMARKS 51

10 REFERENCES 53

List of figures

Figure 1: Business model. 8
Figure 2: UMOBILE architecture. 9
Figure 3: Layered QoS mechanisms 11
Figure 4: Service migration view of the UMOBILE architecture. 12
Figure 5: Left: e2e Flow Control: Bandwidth is split according to the slowest link on the path. Right: INRPP: Bandwidth is split equally up to the bottleneck link (global fairness). Detour applies to guarantee local stability. 20
Figure 6: Average success rate of four containers. 25
Figure 7: Experiment settings. 26
Figure 8 : CPU utilization of busybox on the RPi. 27
Figure 9: CPU load of busybox on the RPi. 27
Figure 10: CPU utilisation of nginx on the RPi. 28
Figure 11: CPU load of nginx on the RPi. 28
Figure 12: CPU utilization of tomcat on the RPi. 28
Figure 13: CPU load of tomcat on the RPi. 28
Figure 14: CPU utilization of busybox containers. 29
Figure 15: CPU utilization of nginx containers. 29
Figure 16: CPU utilization of nginx containers. 29
Figure 17: Response time of the busybox web server. 30
Figure 18: Response time of the nginx web server. 30
Figure 19: **Response time of the tomcat web server** 30
Figure 20: CPU utilization of a RPi under stressing loads. 31
Figure 21: CPU load of a RPi under stressing loads. 31
Figure 22: Container replication cost. 32
Figure 23: The Service Manager and its QoS mechanisms. 33
Figure 24: Integration of Service Migration and DTN to meet availability requirements. 42
Figure 25: UMOBILE components involved in the integration of service migration and DTN. 43
Figure 26: Laboratory settings for proving integration of service migration and DTN. 44
Figure 27: UMOBILE components involved in the integration of service migration and INRPP. 47
Figure 28: Integration Service Migration and INRPP to meet the QoS requirements 49
Figure 29: Average flow completion time for two docker images using INRPP with full priority (FP) and no-priority (NP) compared with random flows using plain NDN. 49
Figure 30: Average throughput for two docker images using INRPP with full priority (FP) and no-priority (NP) compared with random flows using plain NDN. 50

List of tables

Table 1: Examples, of services with their service description. 14
Table 2: Sizes of Dockerized images of four popular web services. 24
Table 3: QoS mechanisms used for supporting services of different classes with specific QoS parameters. 33

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

Executive summary

Background: This document is the *D4.4 deliverable: Set of QoS interfaces and algorithms.* D4.4 is due on month 30 (Jul 2017) and is one of the five deliverables included in *WP4: Service enablement.*

The main objective of WP4 is to enhance UMOBILE architecture by means of the development of mechanisms to provide QoS. These mechanisms will enable the deployment of services with different levels of QoS demands ranging from less-than-best-effort to guaranteed QoS. In this delivery, we contemplate three classes of services: **less-than-best-effort, best-effort** and **premium.**

In pursuit of this aim, these mechanisms are expected to take advantage of the ICN features of the UMOBILE architecture.

The specific objectives of the WP4 are the following:

O4.1: To enable services which fully exploit the inherent opportunistic nature of communication.

O4.2: To enable the "Internet" experience as many people know it, with applications such as web, email, and the like. The challenge here lies in dealing with the inherent disconnectivity of challenged environments by catering to the network challenges and/or adjusting the expected user experience.

O4.3: To develop mechanisms for processing of sensor data through context understanding.

O4.4: To provide different levels of QoS depending on the needs of each user/network ranging from less-than-best-effort to guaranteed services.

The outcome of WP4 will be the enablement of services that support the key characteristics of the developed platform: delay-tolerance and content-centricity.

To fulfill these objectives and as part of the activities envisioned by Task 4.1, UCAM, ATHENA (formerly DUTH) and UCL have implemented independently different mechanisms that operate at different levels of the software stack when offering UMOBILE services to users.

In the UMOBILE project, we have defined and developed another set of mechanisms aimed at prioritizing emergency traffic limited in certain areas and time space, called Name-based replication priorities, for opportunistic communications without infrastructure using Device-to-Device communications. However, these QoS mechanisms for opportunistic communications are developed in Task 4.3 and not in Task 4.1 of the WP4 and have already been defined and detailed in D3.1 deliverable [1]. Therefore this document does not include the Name-based replication priorities mechanisms and is focused on QoS mechanisms of UMOBILE services. The set of QoS mechanisms that this document describes are the following:

- Application level QoS mechanisms

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

   - Service Migration Platform (SM)
- Network level QoS mechanisms
   - Delay-Tolerant Framework (DTN)
   - Flowlet Congestion Control (INRPP).

These mechanisms have been implemented to operate independently in the sense that they use their own algorithms to address QoS requirements. However, they can also be integrated to operate in collaboration mode offering a unified approach towards QoS. Their collaboration is highly desirable in applications that involve QoS parameters that are visible and amenable to manipulation at different levels of the software stack. In account of potential integration, we have built interfaces that a designer can use to integrate one with another.

We have organized the document as follows: Section 1 (Introduction) discusses some core concepts on which the subsequent sections are based. Section 2 presents a summary of the UMOBILE architecture. In Section 3 we explain the layered approach that we have taken to deploy QoS mechanisms in the UMOBILE project. Section 4 explains the architecture of the Service Migration Platform. The focus of the discussion is on the components of the architecture and the algorithms and interfaces that the decision engine uses. It also explains how the service migration platform can work collaboratively with the DTN-framework and INRPP. Section 5 presents and overview of the architecture of the DTN-framework. The focus is on the interfaces that it offers to other modules, in particular to the Service Migration Platform. Section 6 presents and overview of the architecture of the Flowlet Congestion Control mechanism. The focus is on how this module can collaborate with Service Migration Platform and DTN-framework in the fulfillment of common QoS requirements. In Section 7 we present the results of a series of laboratory experiments that we have conducted to identify the parameters and their critical values (thresholds) that the algorithms of the Service Migration platforms needs to take into consideration to decide on opportunistic deployment of service instances. In section 8 we present examples of results that we have achieved to proof the impact of the QoS mechanisms that we have developed. We include examples that show how these QoS mechanisms can be integrated to offer a unified approach towards QoS. The examples show how they can collaborate with each other in the fulfillment of common QoS requirements. Finally, in the Conclusion section (Section 9) we summarize our findings and discuss open questions that we are currently addressing.

1. Introduction

In this document, we define a **service** an application that after being instantiated (from an image) is capable of receiving requests from remote users, processing them and responding with the corresponding results. We use services and applications as synonymous. In the implementations conducted in this projects, we use lightweight

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

virtualization technology (i.e., docker[1]) to compress the service as lightweight container.

Services are categorized into stateful and stateless [2]. The application level mechanisms (explained in Section 3) that we have developed QoS are aimed at **stateless services,** that is, services that, upon request, provide information to clients but do not keep information (state) about their interactions. In stateless services, each request contains all the information that the service needs to process it. Consequently, the service can treat each request independently from others. A positive effect that results from this interaction model is the decoupling between clients and services. Stateless services are quite common and comparatively simple to design and manage. Examples of services that fall within this category are the conventional RESTful web services.

Another central feature of the services of our interest is that they are **single purpose** in the sense that they are built to perform very specific functions for an arbitrarily short time. For example, a service is instantiated to respond to a single or few user requests and discharged. Single purpose services can be assembled into small size images, of the order of a few Mbytes. When additional functionalities are required, independent services are built rather that aggregating them in a single service. We have focused on stateless services of small size because they are emerging as promising technology to be used in edge network computing. Also they match the services used is the scenarios covered by the UMOBILE project such as the web services to be deployed in emergencies scenarios.

One of the salient advantages of stateless services of small size images is their flexibility in deployment. They can be instantiated, migrated, replicated and discharged by the service provider transparently, that is, without serious consequences to the end-users. Transparent mechanisms are simple to deploy, for example, it is not difficult for a service provider to deploy request redirection mechanisms (also called proxies). In this manner, an end-user that places a request against an instance that has been discharged can simply resend the request on the hope that it will be redirected towards an equivalent instance. These operations are central to the service migration platform that we have implemented.

A central concept in this deliverable is **service deployment** which we understand as the act of: retrieving an image of a service from a repository, copying the image to a computer that act as a service host and, instantiating it, always from scratch, as opposed to resuming it from a previously suspended image.

We assume that one of more instances of a given service can be grouped within a single host or spread over several of them. Likewise, a service host can run several instances that are not necessarily of the same service. Deployment of stateful images is discussed in [3].

In this deliverable, we consider a service delivery scenario where a group of remote end–users are interested in accessing services run by a service provider over a network infrastructure that suffers from typical network impairments such as latency, jitter, congestion, limited bandwidth, packet loss and temporary disconnections.

The end-users are assumed to pay for service access and, consequently, expect some levels of QoS from the service provider. The QoS parameters expected vary and are

[1] https://www.docker.com

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

determined by their applications. However, to frame our discussion, in this document we will focus on service availability and latency constraints on the basis that additional QoS constraints can be gradually incorporated to the platform presented in this document.

In addition to the service provider and the end-user, other parties are involved in the delivery of the service that interact with each other on the basis of an agreed business model.

The **business model** determines who is in control and responsible for what and, more importantly, how value is created and revenue generated. There are several alternatives [4,5,6]. In this document, we assume the emerging content access business model where the network provider (also called the ISP provider) offers both network and application services to end–users. The choice of the business model is in line with the latest trends in content delivery that show that traditional ISP are gradually deploying their own content distribution networks. A conceptual view of this model is shown in Figure 1.

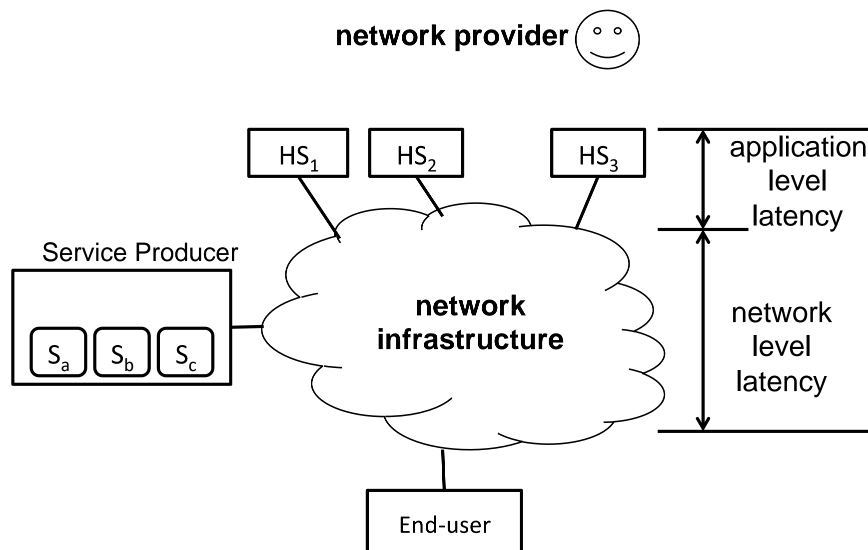


Figure 1: Business model.

This model is network provider centric. The network provider is in possession of the network infrastructure, computational and storage resources (HS_1, HS_2 and HS_3) that he is willing to virtualise to host and deliver services to end-users. The Service Producer is a party in possession of several services (S_a, S_b and S_c) that he is willing to offer to end-users but not directly, say, because he does not have the interest or means to deal directly with end-users, consequently, the Service Producer strikes a bargain with the network provider and delegates to the latter the responsibility of offering the services to end-users.

In this business model, the Service Producer delegates the responsibility of deploying the services under the needed and well–specified QoS requirements to the network provider. For example, a given service S_a might require guaranteed QoS whereas S_c might require

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

less-than-best-effort. QoS parameters are impacted at different parts of the infrastructure, as shown in the figure the latency perceived by and end-user is the result of the latency caused by the application (for example, the time it takes the server to process and present the request to the network interface) and the network (the time it takes requests and responses to transverse the network).

To face the challenge, the network provider relies of his pool of virtualisable resources, namely a set of hosting computers. He is also in possession of the technical description of his resources such as their cpu, memory and disk capacities. In addition, the network provider is in a position to deploy run-time monitors (for example, in each hosting computer and network routers) that provide him with metrics about current usage and demand placed by end users.

On the basis of this information, the network provider can develop and deploy mechanisms that operate at different levels of the software stack to help him deliver the services with the expected QoS even in the presence of temporary disconnections and other network impairments as discussed above. These mechanisms are not available yet in ICN networks. As explained later, a key feature of these QoS mechanisms is the level of the software stack where they are deployed. At one end of the spectrum, we can deploy network level mechanisms whereas at the other extreme, application level mechanisms can be deployed. Examples of such mechanisms are the three mechanisms that the UMOBILE consortium is currently developing: Service Migration Platform, Delay Tolerant Framework and Flowlet Congestion Control.

2. UMOBILE architecture

To set the scenario, this section presents a brief summary of the UMOBILE network architecture explained at large in the D3.1 deliverable [1].

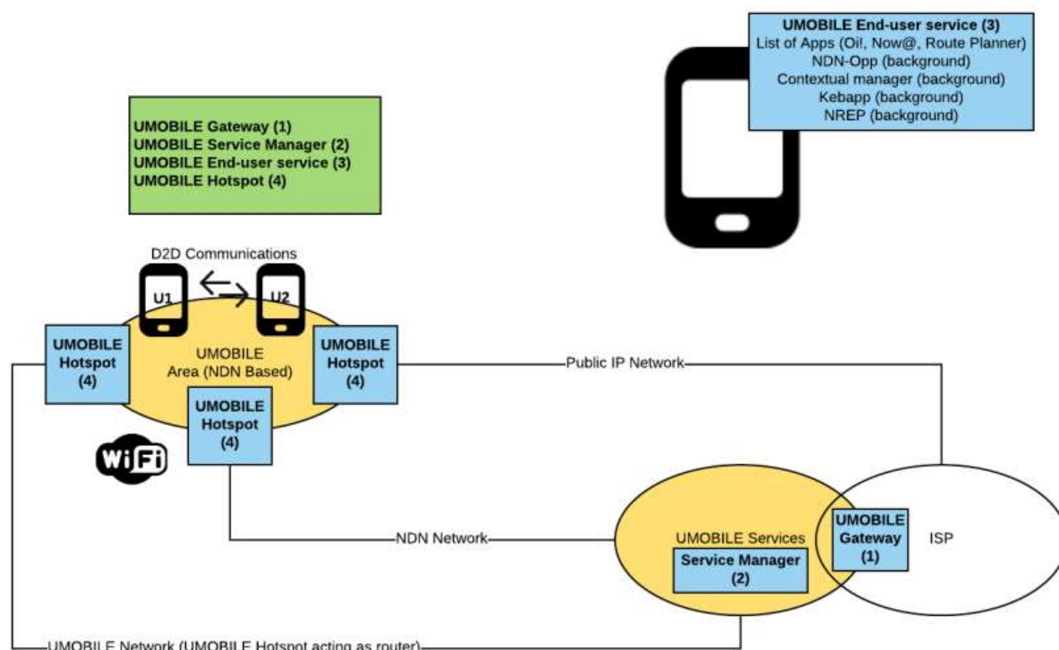


Figure 2: UMOBILE architecture.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

In the figure, the UMOBILE Area is a network built on the basis of the services that the UMOBILE project is developing (see Figure 2). As indicating in the figure, it is NDN based since our development is based on the latest release of the NDN software [7]. The UMOBILE Hotspots connected to the UMOBILE Area network represent the computers that the network provider uses for hosting his services. Two of them are shown in the figure, yet, there can be as many as necessary. The UMOBILE Hotspots are enabled with WiFi communication facilities that they use to offer access to the end user devices. Two of them are shown in the figure (U1 and U2). The end user devices are in possession of D2D (Device to Device) communication facilities.

The left-most UMOBILE Hotspot is connected to a UMOBILE network which is represented by a line. In the UMOBILE project such a network is built out of several UMOBILE Hotspots with routing functionalities. A central element of this UMOBILE network is a computer with disk facilities for storing UMOBILE services and represented by the yellow ellipse located on the left. A key service that we have developed in the UMOBILE project is the Service Manager. Observe that this computer can communicate with the conventional TCP/IP Internet through a gateway (UMOBILE Gateway).

3. QoS mechanisms

Figure 3 shows an end-user placing a request against a remote service with QoS constraints and deployed in a host. The QoS perceived through the response is determined by several parameters that are accessible at different levels of the software stack ranging from the actual application (a service) to the network. Latency for example, is determined by different sources of delays located between the end–user and the service (see Figure 1). For example, the host that runs the service might be overloaded, the actual service might be struggling to handle an unexpected burst of requests, the network might be suffering from congestion, etc. The sources of delays can be grouped into application level and network level. The separation allows designers to address QoS requirements by means of mechanisms deployed between the application layer and the network layers. The motivation for this approach is that it is widely acknowledged that some parameters that impact the QoS as perceived by the end user, are easy to measure and manipulate at different levels of the software.

3.1 Application level QoS mechanisms

At this level, the issue can be addressed by mechanisms that manipulate the actual application and the resources where the application is executed, for example, the hosting servers. These mechanisms can be based on metrics about the configurations of the hosts and their current status of resources such as the current load inflicted on the CPU. As elaborated in subsequent sections, a well-known technique that we are exploring is service replication. Notice that these mechanisms rely on both, the optimisation of the usage of available resources and the possibility of deploying additional ones such as more virtual machines.

3.2 Network level QoS mechanisms

At this level, the issue can be tackled by mechanisms that manipulate network packets. The idea is to use traffic engineering techniques aimed at the optimisation of the use of network resource in existence, mainly by means of monitoring and manipulating network

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

traffic. Consequently, their effectiveness depends on and is bounded by the amount of resources available. In other words, congestion control mechanisms can do nothing when the available network resources are already exhausted. A representative example of such mechanisms is congestion control which involves queue monitoring with subsequent manipulation of network packet within routers. A good introduction into these techniques is presented in [8]. The document introduces conventional TCP techniques such as end-to-end flow control based on bandwidth split in accordance with the slowest link on the path. Other techniques discussed are the In-Network Resource Pooling Protocol (INRPP) that takes advantage alternative sub–paths and in–network cache to detour traffic that exceeds the capacity of links.

3.3 Layered QoS mechanisms

In the UMOBILE project we take a layered approach to address QoS requirements (Figure 3). As shown in the figure, we have implemented QoS mechanisms that can be deployed and operated independently but alternatively they can operate simultaneously and complement each other. For instance, a service provider with powerful hosts and access to a highly reliable network infrastructure might decide to bypass QoS mechanisms and let end-user's requests reach its service directly through the double arrowed dashed line. In contrast, a service provider with resource—constrained hosts and a questionable network infrastructure is very likely to opt for *application level QoS mechanisms* running on top of *network level QoS mechanisms* as shown in the figure.

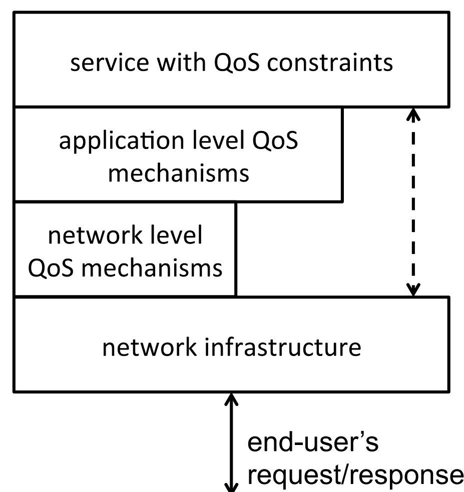


Figure 3: Layered QoS mechanisms

3.4 UMOBILE QoS mechanisms

The UMOBILE project has developed three QoS mechanisms that operate at the application and network level layered as shown in Figure 3.

- Application level
    - Service Migration Platform
- Network level

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

- Delay Tolerant Framework (DTN)
- Flowlet Congestion Control (INRPP).

The overall functionalities of these mechanisms have been discussed in deliverable D3.1 [1]. The focus of this deliverable (D4.4) is on explaining and demonstrating how they can collaborate in the fulfillment of QoS requirements.

4 Service migration platform

The application level QoS mechanisms that we have implemented is a service migration platform that takes advantage of three fundamental concepts:

- Light weight virtualization technology
- Opportunistic service deployment
- Edge network computing

In brief, the central idea is to deploy services instantiated from Dockerised images upon request and at hosts located at the edge of the network and with enough resources to run them with assurance that the services will satisfy their QoS constraints. The architecture of the Service Migration Platform is shown in Figure 4. The figure resulted from Figure 1 after highlighting the components involved in service migration.

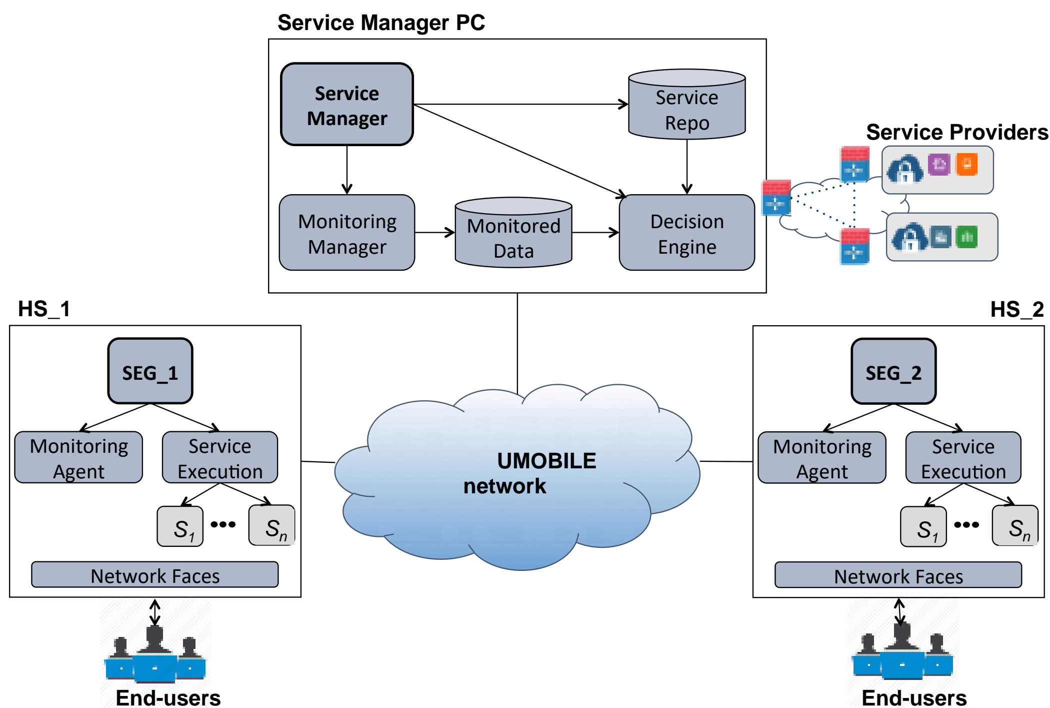


Figure 4: Service migration view of the UMOBILE architecture.

The functionality of the components included in Figure 4 is as follows:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

UMOBILE Network: The UMOBILE network is an NDN network with the software components developed by the UMOBILE project that extend NDN with the QoS mechanisms under discussion in this deliverable (Service migration Platform, DTN framework and INRPP).

Service Providers: A service provider is a business entity that is responsible for delivering services. In the UMOBILE project we assume a provider-centric business model where the service provider is in full control of the communication infrastructure (i.e., routers, access point, IoT sensors) and several source constrained edge nodes such as single board computers. The rationality behind this decision is that this business model is increasingly gaining acceptance in the service provisioning market. The service migration platform is meant to deploy services programmatically, that is, with as least as possible human intervention.

Service Manager PC: it is an ordinary computer used by the service provider to deploy the Service Manager and its components.

Service Manager: is the piece of software that is responsible for making informative decisions on the deployment of services. Being the master module of the Service Migration Platform, it is launched by Service Provider (a human being) when he or she wishes to deploy services with QoS awareness.

The Service manager relies on four modules: Service Repo, Monitoring Manager, Decision Engine and Monitored Data. Once lunched, it initiates (for example, it creates instances of their Python classes) and coordinates their operations.

Decision Engine (DE): implements the logic for opportunistic deployment of instances of services to meet QoS requirements. It operates on the basis of the QoS requirements of individual services (as expressed in their service descriptions), service demand expressed by end users, current status of resources (network and Hot Spots) and algorithms that help determine where and when to deploy a requested service.

Monitoring Manager: is responsible for placing pull requests against the *Monitor Agents* deployed in each Hotspot to collect information about the current status of their resources and the current demand imposed on the service. The monitoring manager stores the monitored that it collects about all Hotspot in the Monitored Data after formatting it as json objects.

Service Repo: is a repository where dockerized compressed images of the services are stored augmented with specification about their QoS requirements. The service specification is a text file that describes the service (for example, service name, image size), QoS parameters (for example, response time and availability) and class of the service. Regarding the class of the service, in this delivery we consider only three classes of services: less-than-best-effort, best-effort and premium.

Service Name	Service Description

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

busybox web server	Service features: image name, image size, …
	QoS parameters: response time < 1 sec, availability: 99.99, …
	Class: premium
tomcat web server	Service features: image name, image size, …
	QoS parameters: response time < 2 sec, availability: 99.99, …
	Class: best-effort

Table 1: Examples, of services with their service description.

Monitored Data: is a permanent storage where the Monitored Manager store the monitored data that it collects for the benefit of the Decision Engine.

Hotspot (HS): A Hotspot (also called an edge node) is a single board computer such as a Raspberry Pi (RPi) or Internet home router with storage, CPU and software facilities for hosting (upon request of the service controller) the execution of virtualised services. Two of them are shown in the figure (HS_1 and HS_2) but there might be an arbitrary large number of them. They are used for hosting the execution of service instances deployed by the service migration platform (shown as S_1 … S_n in the figure) for the benefit of the End-users. To accomplish this task, the HSs run four modules that we have implemented: SEG, Monitoring Agent, Service Execution and Network Faces.

Service Execution Gateway (SEG): A SEG is the master module implemented in Python to run inside a hostspot. We show two of them in the figure (HS_1 and HS_2). The Service Provider launches a SEG in each Hotspot to coordinate the execution of the local Monitoring Agent and Service Execution.

Monitoring Agent: is a piece of software that is responsible for measuring the current status of resources and the current demand imposed on the services. By resources we mean, the actual hardware of the Raspberry Pis used to implement the Hotspot and the Docker containers instantiated in the Raspberry Pis. The monitored data provided by each Monitoring Agent is formatted as json objects.

Service Execution: is a piece of software that we have implemented to instantiate containers automatically upon receiving requests from the Decision Engine.

End-user: End users are individuals in possession of mobile devices used for placing service requests. They use the HSs to access the UMOBILE network.

4.1 Reactive and proactive (pre-fetching) service migration

The modules that compose the service migration platform (described above) are flexible enough that they can be used to deploy services in accordance with different strategies. Our research effort has focused on two of them: reactive service migration and proactive service migration (pre-fetching). In the deliverable D3.4 we presented the general principles of these strategies, in this deliverable we present a detailed discussion supported with specific examples and results.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

4.1.1 Reactive service migration

Reactive service migration is a mechanism that a service provider can use to deploy instances of services when and where they are needed to meet the QoS expected from the services. It is a reactive mechanism in the sense that it is used at service delivery time to deploy instances in response to variations (between “vigorously running” and “showing signs of exhaustion) of the status of the resources involved in the execution of running instances. Examples of resources are the computers (Raspberry Pi) that we use in the UMOBILE project for realising the hotspots and the Docker containers used for creating the instances of the services. Information about the current status of the resources is provided by the Monitoring Manager.

4.1.2 Proactive service migration (pre-fetching)

Service pre-fetching is a proactive mechanism that a service provider can use to cache images of services before they are needed. It is a proactive mechanism in the sense that images of the services are cached in advance (for example at midnight) and as close as possible to where they will be subsequently instantiated and used. The cached instances are not necessarily instantiated immediately after caching. Enough for the service provider is to have them ready for instantiation when needed to meet the expected QoS requirements. As opposed to reactive service migration, service pre-fetching is not concerned with the current status of the resources where the instance will be subsequently instantiated because this status is likely to change by the time the instance is actually instantiated.

4.1.3 Introductory example

To explain the operation of the service migration platform let us discuss an example where we use it as a reactive mechanism. Let us take Figure 4 and imagine that the Service Provider is responsible for providing service S_1 with certain QoS constraints. The nature of the QoS parameters is irrelevant in this example, but one can think of response time, availability and similar QoS parameters. Also, imagine that an End-user that accesses the UMOBILE network through HS_1 is interested in S_1. No instances of S_1 are currently running.

To meet the expected QoS constraints, the Service Provider uses the service migration platform and proceeds to launch it as follows:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

1. The Service Provider launches the SEG_1 and SEG_1 in HS_1 and HS_2, respectively.
2. The Service Provider launches the Service Manager in the Service Manager PC.
3. The Service Provider stores an image of S_1 and its service specifications (see Table 1) in the *Service Repo*.
4. The Service Manager periodically (for example, every 60 secs) executes the Monitoring Manager to collect information about the latest status of the resources of the HS_1 and HS_2.
5. The End-user places a request to access S_1 that is received by the Service Manager.
6. The Service Manager initiates the Decision Engine to deploy S_1.
7. The Decision Engine accesses the service description file of S_1. From the service description, the Decision Engine learns, the class of the service. To motivate this example, let us assume that S_1 is a premium service.
8. The Decision Engine consults from the Monitored Data the latest status of the available resources.
9. On the basis of the service description of S_1, the Decision Engine selects a decision deployment algorithm from a repository, provides it with the QoS specification and the current status of resources of the HS nodes and executes it.
10. The decision deployment algorithm outputs the name of a candidate HS node to host an instance of S_1. For the sake of this explanation, let say the output is HS_1.
11. The Decision Engine accepts the output, retrieves the image of S_1 and composes a deployment description text file.
12. The Decision Engine sends both, the image of S_1 and the deployment description file to the Service Execution component of the Hotspot selected, to HS_1 in this example.
13. The Service Execution under the control of SEG_1 creates an instance of S_1.
14. S_1 responds to the End-user.

It is worth emphasizing the service migration platform operates in reactive mode in the sense that the deployment of the instance is based on the current status of the resources (point 8 and 9). No image pre-fetching takes place.

4.1.4 Decision engine algorithms and interfaces

The Decision Engine includes ancillary functions and decision making functions that implement the decision making algorithms. The ancillary functions help in the manipulation of the about resource consumption, specification about services, specification of service deployment and so on. The decision making algorithms are the code that decide on service deployment.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

To illustrate the point, we will show examples of both ancillary functions and actual algorithms. We will show their usage in Section 8.

In this implementation, we use json objects, namely json dictionaries, to format information about the status of the resources of the Hotspot nodes. In this order, each Monitoring Agent collects the status of its local resources, produces a json object (*json obj with monitored data*) and presents it to the Monitoring Agent. The Monitoring Agent aggregates these objects into a single *json obj with aggregated monitored data* which is in fact a list of *json obj with monitored data* objects (a list of json dictionaries).

The following code is the skeleton of the *json obj with monitored data* that we use for collecting information about the status of a Pi. Observe that it can have zero or more containers. In the following snippets of Python code, we use the variable *json_lts_dict* (list of json dictionaries) to hold the *json obj with aggregated monitored data*.

```
pi_status_default= {
# Identification of the Hotspot node controlled by a SEG
 'PiID': 'ID',
 'PiIP': 'IPaddress',

#Hard configuration of the Hostpot node controlled by a SEG
 'hardResources':
    {'cpu': 'cpuType',
    'mem': 'MemorySize,
    'disk': 'diskSize'},
#Soft configuration of the Hostpot node controlled by a SEG
 'softResources': {'OS': 'OperatingSystemType},
 'resourceUsage':  {'cpuUsage': 'HostCPUusage',
                   'cpuLoad': 'HostCpuLoad,
                   'memUsage':'HostMemoryUsage},
#Resources consumed by containers instantiated in the Hotspot controlled by a SEG.
 'containers':
    [
    # Resources used by first container
    {'id':      'containerID',
    'cpuUsage':  'containerCpuUsage',
    'memUsage':  'containerMemoryUsage',
    'name':      'nameOfTheServiceInContainer',
    'status':    'runningTimeOfTheContainer',
    'image':     'nameOfImageOfContainer',
    'port_host':  'portNurOfEdgeNode',
    'port_container': 'portNumOfContainer'},

    # Resources used by second container
    {'id':      'containerID',
    'cpuUsage':  'containerCpuUsage',
    'memUsage':  'containerMemoryUsage',
    'name':      'nameOfTheServiceInContainer',
    'status':    'runningTimeOfTheContainer',
    'image':     'nameOfImageOfContainer',
    'port_host':  'portNurOfEdgeNode',
    'port_container': 'portNumOfContainer'},

    # More containers can be included in the list
    ]}
```

The following functions are examples of ancillary functions that we have implemented.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

- get_num_of_containers_of_pi(): A Pi used for realizing a hotspot can host the execution of zero or more containers. This function produces the number of containers currently running in a Pi.
- get_pis_with_cpuLoad(): The Pi can experience different levels of cpu load. This function produces the current cpu load of all the Pis and sort them is increasing g order.
- get_pi_cpuLoad(): This function produces the current cpu load of a given Pi.
- get_pis_with_min_cpuLoad(): To decide on deployment of containers, it is useful to identify the Pi that is currently less loaded. This function produces the ID of such a Pi. Note that the Pi is not necessarily unique, there might be one or more of them experiencing the same load. The function selects one of them arbitrarily.
- selectHost_to_deploy_firstInstance(): When the first instance of a service is to be deployed a hotspot needs to be selected on the basis of the requirements of the services and the current status of the hotspot that can potentially host the instance. This function produces the ID of a Pi that can potentially host the instance.
- try_localReplication_of_additionalInstance(): Some times it is convenient to deploy additional instances in the same hotspot where a previously deployed instance is already running, provided that the hotspot has enough resources to host another instance, in the contrary, a remote replication should be applied. This function determines whether an additional instance can be deployed locally (that is, in the same Pi).

5 DTN framework

The architecture of the DTN framework is explained at large in the D3.1 deliverable [1]. In this section we present only a summary of its functionality with focus on the interfaces that the DTN framework offers to other modules, in particular, to the Service Migration Platform.

5.1 Architecture and functionality of the DTN framework

The DTN framework enhances the UMOBILE architecture with a delay tolerant protocol (Delay Tolerant Networks-DTN) that can be activated (manually or automatically) when the conventional ones (for example TCP/IP) fail due to network problems. To implement delay tolerance, the DTN framework takes advantage of caching resources. The DTN framework seems to be suitable for dealing with QoS parameters related to availability and not particularly convenient to address stringent QoS requirements related to timing such as response time and latency. Essentially, on it own, DTN can be used to enhance the operation of the network by means of the following facilities:

1. less-than-best-effort service: This class of service can be used to provide communication to services with relaxed response time requirements.

2. Reliable communication service: This class of service can be used in situations where service availability is essential (e.g. in emergency cases).

This project has received Funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

3 Finer-grained services, by opting in and out of the above.

4 A congestion control mechanism: Communication over the DTN protocol can be activated reactively to off load traffic main communication links suffering from congestion.

5 A congestion avoidance mechanism: proactively scheduling traffic to reach its destination only when links are underutilized.

5.2 DTN integration interfaces

The DTN forwarding mechanism can be employed in cases when one or more of the following conditions are met:

- There exist expected or unexpected connectivity disruptions in some parts of the network.
- Reliability must be ensured to guarantee the delivery of large packets in cases when the small packet size of NDN causes unwanted delays (e.g., in the case of a mobile node used as a data mule).
- There is congestion in the network.
- A less-than-best-effort service class must be used.

Observe that the integration of the service migration platform and the DTN framework is facilitated by the naming system of the NDN network: Interest packets with DTN prefixes in their names are forwarded through DTN interfaces. The service provider can activate it (see Figure 23) by means of executing procedures to make Hotspots DTN aware. These procedures involve the configuration of the FIB (Forward Interest Base) of the Hostspots to set DTN faces and can be done either programmatically by the Service Manager or manually by the service provider.

In this order, if one or more of the above conditions are met, the Service Migration platform can leverage the DTN forwarding mechanism by employing the corresponding naming scheme that is tied with the DTN interface, based on the node configuration. This way, any node configured to support the DTN forwarding mechanism, will forward packets with the DTN-related naming scheme using the DTN interface. It must be noted that in case a node is not DTN-enabled, it will fall back to the typical forwarding mechanisms that use TCP or UDP.

6 Flowlet congestion control

The architecture of Flowlet is explained at large in the D4.1 [9] and D4.2 [10] deliverables. In this section, we present only a summary of its functionality with focus on the interfaces that the framework offers to other modules, in particular, modules related to QoS mechanisms.

6.1 Architecture and functionality of the INRPP congestion control

As discussed in D4.1 [9] and D4.2 [10] one can use congestion control mechanisms to

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

prevent congestion problems that might result in packet loss, latency and low throughput. Congestion control mechanisms address QoS by means of manipulating information at network level such as monitoring of queues with subsequent manipulation of network packet within the routers. Examples of these techniques are traffic management by means of end-to-end or hop-by-hop congestion control.

Within the UMOBILE project, we aim to design and evaluate the In-Network Resource Pooling Protocol (INRPP), which pools bandwidth and in-network cache resources in a novel congestion control framework to reach global fairness and local stability. There are two main uncertainty factors that fuel fear of instability and with which any reliable congestion control protocol has to deal with: i) the input load factor: the network does not know how much data the senders will put in the network next, and ii) the demand factor: there might be excessive demand for bandwidth over some particular area/link. TCP, for instance, defends against the input load factor through the Additive Increase/Multiplicative Decrease transmission model, while it deals with the demand factor by adopting the "one-out, one-in" packet transmission principle (only when a packet gets out of the network is a new one allowed in). Those two mechanisms are closely linked and interrelated and lead to TCP's defensive behaviour by effectively (proactively) suppressing demand. In this essence, the end-points have to speculate on the available resources along the end-to-end path and move traffic as fast as the path's slowest link.

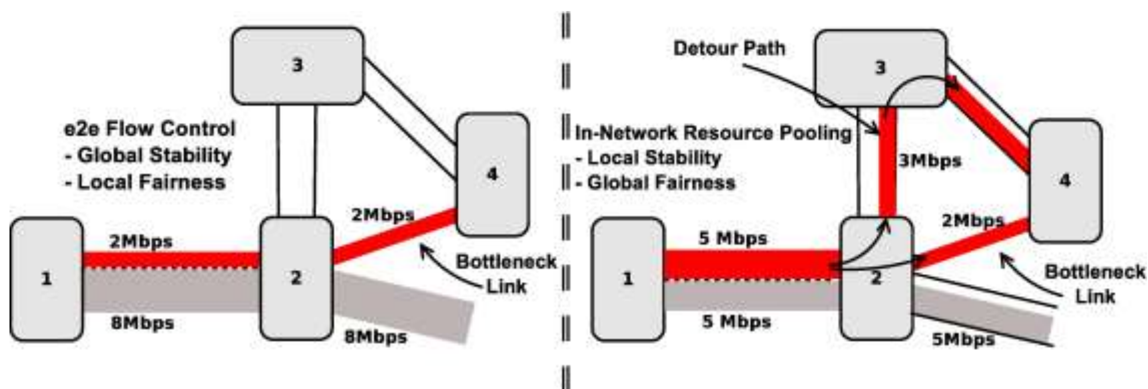


Figure 5: Left: e2e Flow Control: Bandwidth is split according to the slowest link on the path. Right: INRPP: Bandwidth is split equally up to the bottleneck link (global fairness). Detour applies to guarantee local stability.

Given the single-path nature of TCP, moving traffic according to the path's slowest link guarantees global stability (i.e., stability along the e2e path through e2e rate-adaptation). Fairness on the other hand, is guaranteed locally (i.e., based on the capacity of the bottleneck link). We argue against this relationship and in the spirit of INRPP propose that: i) stability should be local, and ii) fairness should be global. Local stability demands that the node before the bottleneck link takes appropriate action when conditions deteriorate. Global fairness on the other hand requires that all resources up to and including the bottleneck link are shared equally among participating flows. Consider two flows in the topology of Figure 5. According to the e2e flow control of TCP (left part), the flow that traverses the bottleneck link (2-4) would achieve 2Mbps throughput (global stability), while the second flow would dominate the shared link (1-2) and achieve 8Mbps throughput. According to Jain's Fairness Index [11], given by $F = \frac{\sum(T)^2}{n\sum(T^2)}$, where T is each flow's throughput and n is the total number of flows, the system fairness in this case is

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

0.73. In case more than one flows traverse the bottleneck link (2-4), they would share equally the available bandwidth (local fairness). In contrast, according to the global fairness, the shared link (1-2) is split equally among the two flows. Node 2 has two options in this case: i) find alternative routes to reach node 4 (local stability), or ii) notify node 1 to reduce its sending. In the topology of Figure 5, node 3 can accommodate the extra 3Mbps. In this case, Jain's index indicates perfect system fairness equal to 1.

Taking profit of the hop-by-hop design and the caching capabilities inherent in the NDN networks, or adding caches (i.e., temporary storage) and breaking the end-to-end principle in TCP/IP networks, we argue that the demand factor can be tamed, providing global fairness and local stability. Given this functionality of in-network storage, INRPP comprises three different modes of operation:

1. push: content is pushed as far in the path as possible in an open-loop, processor sharing manner, based on the path's hop-by-hop bandwidth resources to take advantage of under-utilised links;

2. store and detour: when pushed data reaches the bottleneck link, the excess data is cached and simultaneously forwarded through detour paths towards the destination;

3. backpressure: if detour paths do not exist or have insufficient bandwidth, the system enters a backpressure mode of operation to avoid overflowing of the cache. During the backpressure mode, the nodes enter a closed-loop mode, where an upstream node sends one data packet per one received ACK to the backpressuring downstream node.

Note that congestion control mechanisms are based to the optimisation of network resource usage, consequently, their effectiveness depends on and is bound by the amount of resources available. In other words, congestion control mechanisms can do nothing when the network resources are already exhausted. In that case, application-level QoS mechanisms are required.

In summary, INRPP congestion control features the following characteristics to provide QoS network-level mechanisms:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

- Hop-by-hop congestion control using in-network caches as a temporary custodian to alleviate temporary congestion without slowing down the source.

- Use of in-network multipath, being able to use unused bandwidth in one-hop detour paths (i.e., paths that can be used to reach the next node only with a difference of a one hop more).

- It provides network stability using backpressure mechanisms and perfect fairness.

- Improves flows completion time and better use of resources.

- Using differentiated level services in the caches, we add priorities to deliver packets from the temporary custodian cache to the link at different rates.

6.2 INRPP integration interfaces

We envision to operate Flowlet as a mechanism that can be activated by the Service Manager (see Figure 23) as an executable file. Once activated the process can remain operating in the background until the Service Provider instructs the Service Manager to stop it.

The INRPP congestion control can be used along with predefined priorities in order to transfer critical services from the service repository to any point of the network with the minimum transfer time and reduce the latency that the user perceives when instantiating the service.

- There exist expected or unexpected connectivity disruptions in some parts of the network.
- Reliability must be ensured to guarantee the delivery of large packets in cases when the small packet size of NDN causes unwanted delays (e.g., in the case of a mobile node used as a data mule).
- There is congestion in the network.
- A lower-than-best-effort service class must be used.

Observe that the integration of the service migration platform and the INRPP framework is, as with the integration with the DTN mechanism, facilitated by the naming system of the NDN network: Interest packets with specific prefixes in their names are forwarded by INRPP with different priorities. For example:

- Premium service: /premium/service1/image1
- Best-effort service: /be/service2/image2

In case of a service using a DTN prefix, as stated in the previous section it falls back to normal best-effort service, being totally agnostic for INRPP and treated accordingly for DTN enabled devices. Therefore, INRPP can be considered as a mechanism that is used for in-path devices between DTN-enabled end-points at a lower level than DTN, but both are considered network-level QoS mechanisms.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

7 Evaluation of QoS-related parameters

When the service migration platform is used as a reactive service migration mechanism it relies on the Decision Engine to make decisions on the deployment of service instances. The responsibility of the Decision Engine is to prevent the violation of the QoS parameters of the services by means of deployment of additional instances of services showing signs of exhaustion, for example, of services showing signs of unacceptable latencies. For this to be possible, the Decision Engine needs accurate knowledge of the key parameters that determine the level of QoS of services and their critical values (thresholds). Also, the Decision Engine needs to weight the expected benefits against the cost of the deployment of additional instances of the service. We have observed that these parameters can be categorized into:

- The current status of the hosting hardware,
- The current status of the existing instances of the service and
- The particularities of the service.

Once the parameters and their critical values are identified, they can be included as input to the algorithms of the Decision Engine and in the list of the parameters that the Monitoring Agents need to monitor. It is worth explaining that in the implementation of the service migration platform we take a service-side approach in the sense that the monitored data is collected exclusively from within the resources under the control of the service provider. We do not rely on support from the end-users. For example, we do not rely on information provided by end users to determine response time or deploy dummy users to probe the services. These alternatives fall outside of our research interest and are not discussed further.

Some of these parameters can be retrieved from the documentation, for example, from the specification of the services. However, the retrieval of others is far more problematic as it demands actual experimentation. We will explain now a series of laboratory experiments that we have conducted in pursuit of this endeavour.

7.1 Experiment settings

In this section we describe the configuration of the software and hardware components that we use in our experiments.

HSs*:* To implement the HS nodes we use Raspberry Pi 3 (RPi-3). The Pis used in our experiments run the Hypriot OS Version 1.2.0[2] in default mode that allows containers to compete for the RPi-3 resources. To implement the virtualized services, we use Docker containers [11,12].

Services: The services that we use in the deployments are web services. They help us to demonstrate how the configuration and the implementation of an application impact the QoS as perceived by the end users. We have selected four of the most popular web servers from the docker hub[3]. Table I shows the images that we used in the experiments.

Image name	Image size

[2] https://blog.hypriot.com/downloads/
[3] https://hub.docker.com/explore/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

hypriot/rpi-nano-httpd	88Kbytes
hypriot/rpi-busybox-httpd	2.16 Mbytes
gordonff/rpi-tomcat	251 Mbytes
armhfbuild/nginx	368 Mbytes

Table 2: Sizes of Dockerized images of four popular web services.

In accordance with the amount of bytes involved in the response, we regard *hypriot/rpi-nano-httpd*, *hypriot/rpi-busybox- httpd* and *armhfbuild/nginx* as light weight web servers. The three of them deliver a single html document that consists of html text of 300 bytes with a link to a local jpeg image of 80 kB. We deliberately use a small html document to reduce the memory consumed by the document and leave it entirely at the disposition of the Docker containers. On this account, we regard *gordonff/rpi-tomcat* as a heavy weight web server since its front page consists of multiple items (e.g., photo, external links, java web applications).

Apache benchmarking (ab) tool[4]: Our experiments involve the generation of artificial sequential http requests. We generate them with the Apache Benchmarking (ab) tool run in a laptop (lenovo E560: Intel Core i5-6200U 2.3GHz, 8GB RAM, Ubuntu 14.04) that we use as a test machine. By sequential http requests we mean that a user places a request and waits for the arrival of the corresponding response. Upon the arrival of the response the user proceeds immediately to place another request. We use the ab tool to place concurrent http request to web services in the following manner. We create linux shells in the Lenovo computer and run the ab tool configured to simulate different numbers of concurrent users. For example, a shell with the ab tool simulating a single user is equivalent to a single request arriving at the web service, a shell with the ab tool simulating two concurrent users is equivalent to two requests arriving concurrently at the web service, a shell with the ab tool simulating 100 concurrent users is equivalent to 100 requests arriving concurrently at the web service.

7.2 Impact of the number of concurrent requests on QoS

The aim of these experiments is to determine how many concurrent requests a given service can handle without compromising response time. This value is application dependent. We examine the response time of the web services of Table I instantiated as Docker containers in the Hot Spot nodes and exposed to different numbers of concurrent requests.

We configured each of the four web servers with the necessary libraries to serve a single web page and hosted each of them in its own container. We deployed the four resulting containers in four RPis (one each) and exposed each of them separately to a total of 10,000 http requests.

We generated the http request from the Apache ab tool in the following manner. We created linux shells in the test machine. In each shell, we run an instance of the ab tool

[4] http://httpd.apache.org/docs/current/de/programs/ab.html

This project has received Funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

and configured it to create a number of concurrently active users: each user generated a number of sequential http requests.

Under this configuration, we conducted individual response time stress tests on each container increasing the number of users from 5 to 1000. In this order, in a five concurrent users experiment, each user generates 10000/5=2000 requests sequentially, in a 100 concurrent users experiment each user generates 10000/100=100 sequential requests only, and so on. It is worth recapping that the number of concurrent users determines the number of concurrent http requests received by the container. For instance, with 1000 concurrent users, the container receives and handles 1000 requests concurrently. We measured the success rate at the test machine by comparing the number of http requests sent by the ab tool and the number of responses received. Figure 6. shows the average number of http requests successfully served by the nano, busybox, nginx and tomcap containers under different concurrent user levels. We repeated each experiment for 30 times.

Let us examine the nano container firstly (red bars), from five concurrent users onwards, it exhibits a success rate of 99.98% as it fails to respond to some of the 10000 requests. The success rate significantly decreases to 24.59% when the number of concurrent users is increased to 100. The performance of the busybox container (white bars) is similar to that of the nano container. Busybox success rate falls to 34.85 % when the number of concurrent user level increases to 500 users. As for the nginx (blue bars) and tomcat (black bars) containers, both can serve up to 1000 concurrent users with a success rate of 100%.

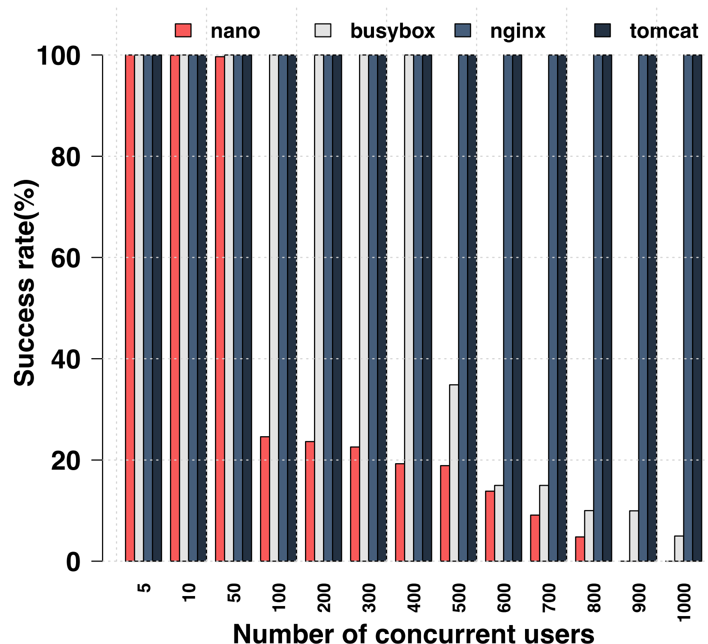


Figure 6: Average success rate of four containers.

The results show that nano containers cannot serve more than five concurrent users with acceptable response time. This parameter needs to be taken into account by the Decision

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

Engine which might decide to deploy additional instances of the containers to avoid compromising the QoS. The creation of additional instances is reasonable since these containers consume only a few Kbytes of memory of the underlying hardware. In previous work, we have proven that a single RPi can run more than 2400 instances of the nano container simultaneously.

7.3 Impact local container replication

In some situations, it is convenient to deploy additional instances of a service to share the load of existing ones. The three experiments that we discuss in this section were performed to investigate how many replicas a *Hot Spot* node can handle without exhausting its resources (See Figure 7). This parameter is crucial for Decision Engine. We use the ab tool to create clients that independently send a number of sequential http requests against web services.

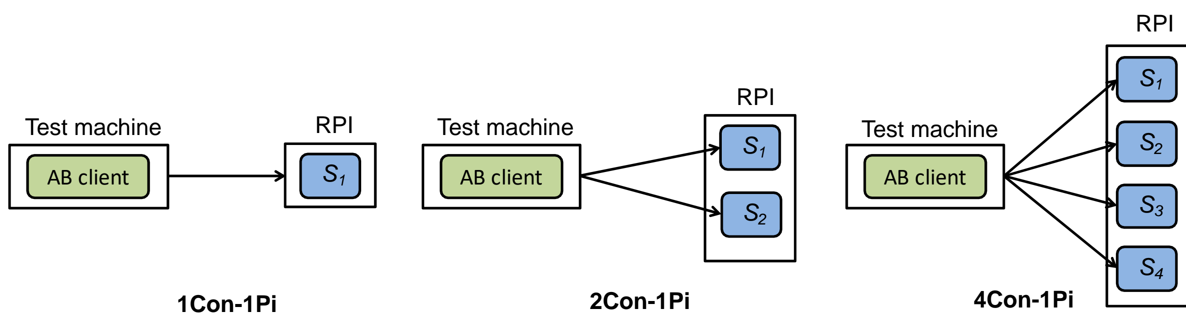


Figure 7: Experiment settings.

In the 1Con-1Pi experiment, we instantiated a single container in a RPi and a linux shell in the test machine. In the shell, we configured the ab tool to simulate 100 clients instructed to send 100 http sequential requests each, to retrieve an html document from the container. The 100 clients operate concurrently, consequently, at any time, the container has 100 requests to processed concurrently. By the end of the experiment, the container would have processed 100x100= 10000 requests in total.

In the 2Con-1Pi experiment, we instantiated two containers in the RPi and created two linux shells in the test machine. In the first shell, we configured the ab tool to simulate 50 clients instructed to send 100 http sequential requests each, to retrieve an html document from the S_1 container. The second shell was configured similarly but targeted the S_2 container. Due to clients concurrency, at any time, each container has 50 requests to process. Like in the previous experiment, by the end, the RPI would have processes 10000 request in total (5000 by each container).

In the 4Con-1Pi experiment, we instantiated four containers in the RPI and created four linux shells in the test machine. In the first shell we configured the ab tool to create 25 clients instructed to send 100 http sequential requests each to retrieve an html document from the S_1 container. Consequently, S_1 received 2500 requests in total. The second, third and fourth shells were configured similarly but targeted the S_2, S_3 and S_4 containers, respectively. Due to the clients' concurrency in each shell, at any time, each container has to process 25 concurrent requests. Like in the two previous experiments, by the end, RPi would have received 10000 request in total (2500 by each container).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

As shown in Figure 8 -13 , we conducted the 1Con-1Pi, 2Con- 1Pi and 4Con-1Pi independently with the busybox, nginx and tomcat containers with aim of measuring how the resources of the RPI are impacted by the local replications of containers. We left out nano container because its inadequacy to support large numbers of concurrent clients renders it unsuitable for these experiments. The results demonstrate that the CPU utilization, CPU load and memory usage of the RPi increase significantly when the number of containers increases. This is because the RPi allocates independent resources (for example, memory buffers and CPU cycles) to each container to handle the communication with the clients. As a result, the creation of an additional container replicates the consumption of RPi resources. This results need to be taken into consideration by the Decision Engine. For instance, in Figure 8, the CPU utilization of busybox container in the 1Con-1Pi experiment exhibits a sharp increase after 10s. This is a sign of exhaustion of the container. It might decide to deploy an additional instance of the container before the QoS is compromised.

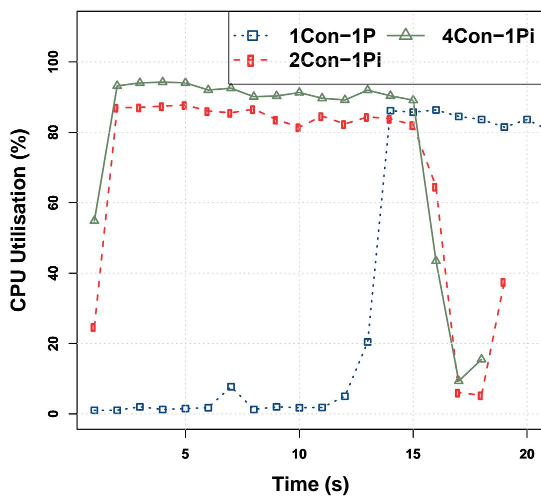


Figure 8 : CPU utilization of busybox on the RPi.

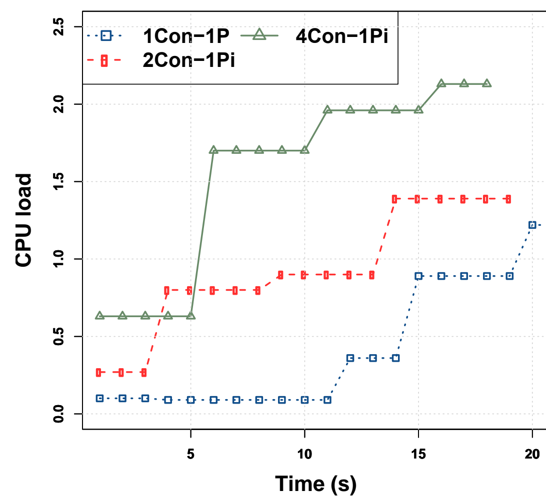


Figure 9: CPU load of busybox on the RPi.

This project has received Funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

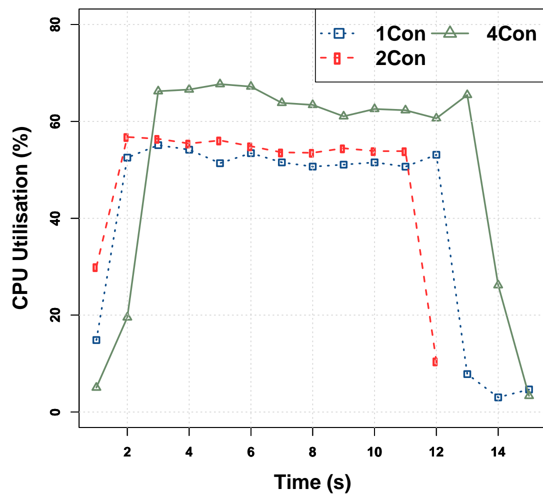


Figure 10: CPU utilisation of nginx on the RPi.

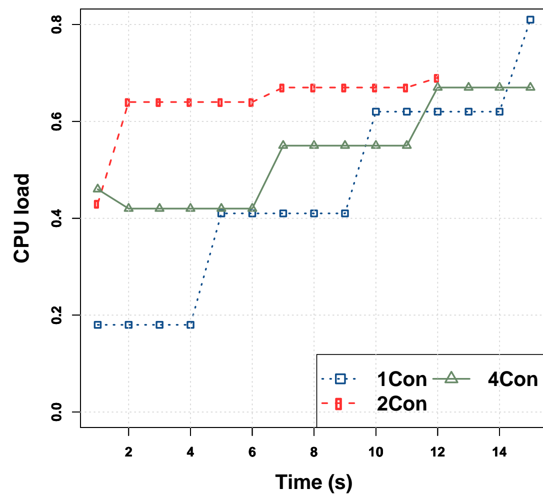


Figure 11: CPU load of nginx on the RPi.

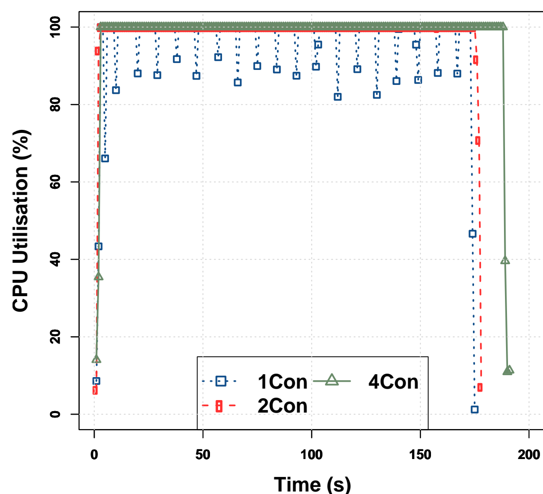


Figure 12: CPU utilization of tomcat on the RPi.

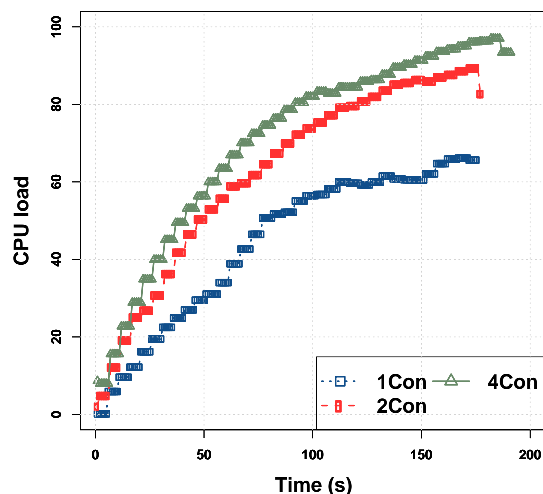
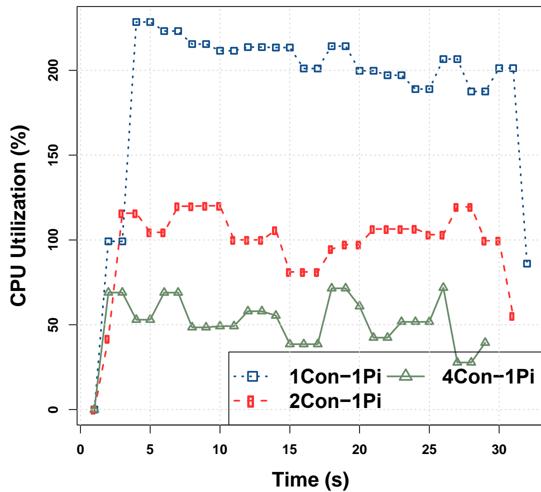


Figure 13: CPU load of tomcat on the RPi.

In practice, there will be several containers running in the same physical host sharing the common pool of resources. Because of this, the information about the status of the RPi's resources is not sufficient to program the Decision Engine. In addition to that, the Decision Engine needs to be aware of the status of resources consumed by each container. The results shown in Figure 14, 15 and 16 which compare the CPU utilization of three containers with different configurations, support our argument. The CPU utilization of all three containers fall to around 50% and 75% when two (2Con-1Pi) and four (4Con- 1Pi) containers are deployed in the RPi. The three plots show that each service exhibits different level of exhaustion. The CPU utilisation inflicted on a single nginx container is only about 60%; this finding indicates that a single instance of nginx can handle the load. In contrast, the busybox container requires four containers to keep CPU utilization under 50%. The tomcat containers exhibit instability over all experiments that drove CPU

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

utilization to the extremes (over 100%).



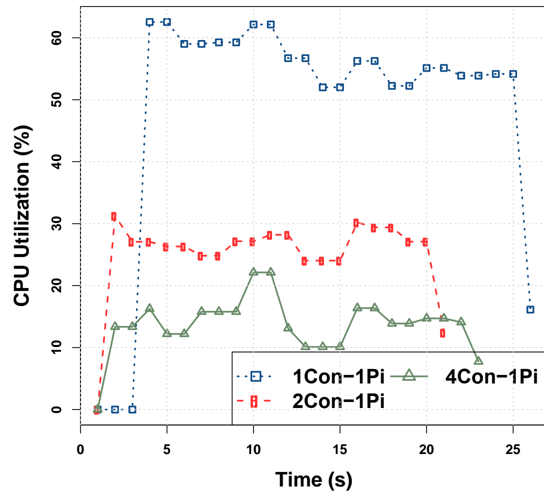


Figure 14: CPU utilization of busybox containers.

Figure 15: CPU utilization of nginx containers.

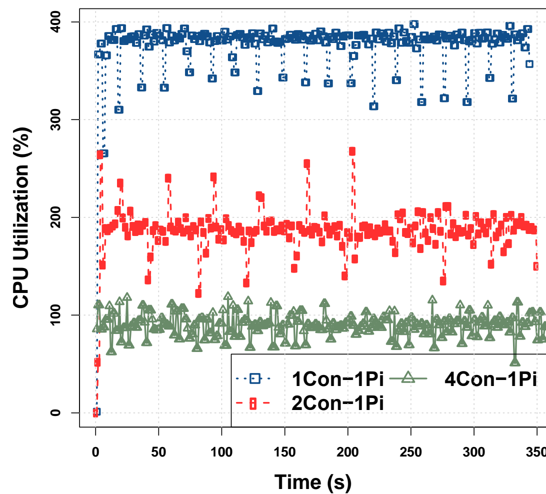


Figure 16: CPU utilization of nginx containers.

7.4 Local vs Remote Container Replication

We conducted experiments to understand how exhausted resources impact the QoS perceived by clients, we focused on response time. In addition to the three experiment configurations mentioned in the previous section (local replication), we include now configurations 2Con-2Pi and 4Con-4Pi aimed at showing the impact of container replication on alternative Pis (remote replication). In experiment 2Con-2Pi, we instantiated two containers (S1 and S2) on two RPis (RPi1 and RPi2)— one container each. In the test machine, we created a linux shell where we used the ab tool to create 50

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

clients operating concurrently. Each of them sent 100 sequential requests to S_1. Thus at any time, S_1 had 50 requests to process. We associated S_2 to another shell similarly. The 4Con-4Pi experiment is similar but aimed at reducing the level of client concurrency. We created four containers on four RPis (one each). Each container was exposed only to 25 concurrent clients instructed to generate 100 sequential requests each. To setup the exper- iments, we connect four RPis and the test machine with a D-Link DES-1008D switch via ethernet cable. The average round trip time between test machine and each RPi is 5ms.

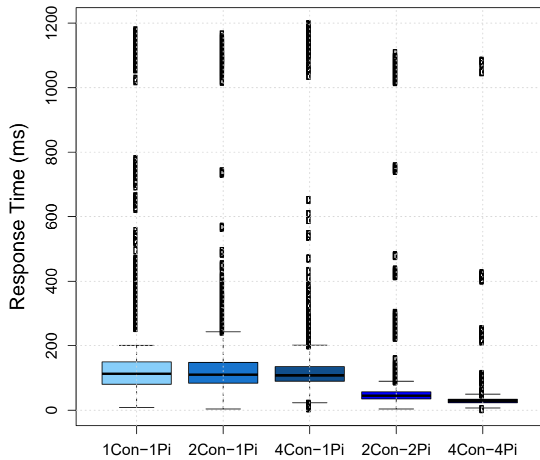


Figure 17: Response time of the busybox web server.

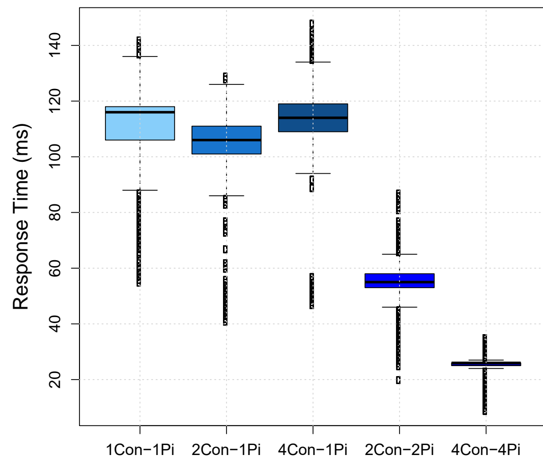


Figure 18: Response time of the nginx web server.

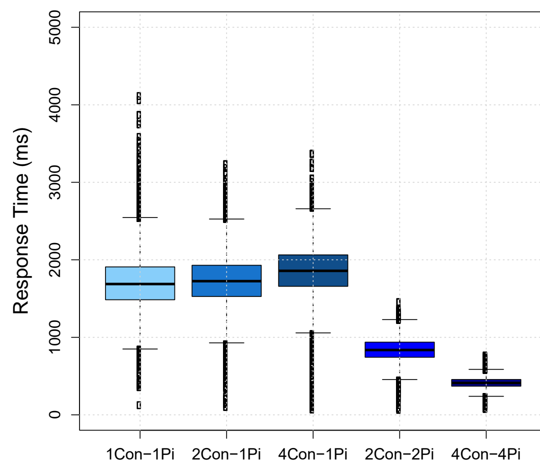


Figure 19: Response time of the tomcat web server

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

Figure 17, Figure 18 and Figure 19 show the average response time of web containers with different service deployment configurations. The computation load of 2Con-1Pi and 4Con-1Pi are analytically reduced by 50% and 75% compared to a single container case (1Con-1Pi) as a number of concurrent users placing the request against each container is divided to 50 and 25 respectively. However, in both configurations, the end-users cannot achieve better response time. In case of busybox and nginx, the end-users achieve almost similar results for all three configurations. As for the tomcat, the average response time is slightly increased when more containers are replicated in the same RPi. On the other hand, applying the remote replication strategy (2Con-2Pi and 4Con4-Pi) significantly improves the performance of response time. For instance, in case of busybox container, the average response time is improved up to 55.01% (2Con-2Pi) and 77.25% (4Con-4Pi) compared to 1Con-1Pi case. Similar tendency is also applied to nginx and tomcat containers.

The implication behind these results is related to resource exhaustion of the RPi. The measurements of CPU utilisation and CPU load are presented in Figure 20 and Figure 21. As shown in Figure 20, the deployment of two and four containers in a single RPi (2Con-1Pi and 4Con-1Pi, respectively) cause higher CPU utilization and CPU load than the deployment of a single container (1Con-1Pi). However, when the containers are deployed in another RPi (2Con-2Pi and 4Con-4Pi), the CPU usage gradually decreases. The experiment with the tomcat container is an example of extreme resource exhaustion where the CPU is fully utilized and CPU load increases up to 70. Such a load exhausts the CPU of the RPi and severely affects its average response time which reaches up to 1847 ms (4Con-1Pi). The Decision Engine needs to be aware of these parameters and remedy the situation, for example, by deploying an additional instance of the container in another RPi to take the excessive load.

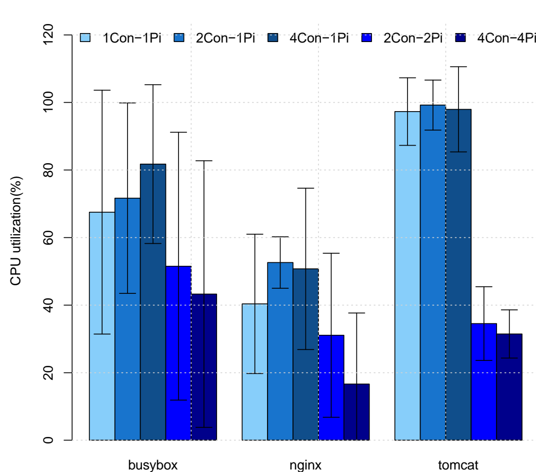


Figure 20: CPU utilization of a RPi under stressing loads.

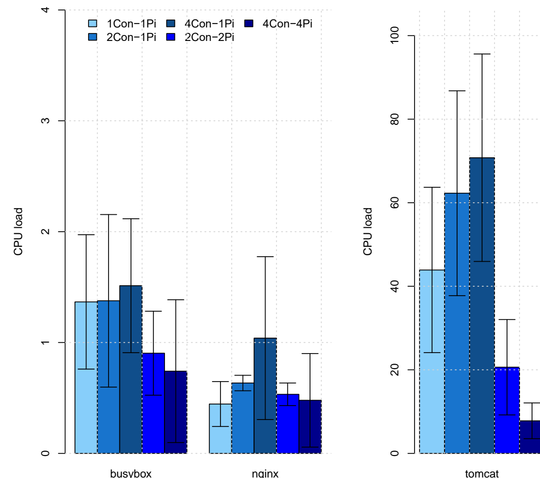


Figure 21: CPU load of a RPi under stressing loads.

7.5 Container replication cost

Deployment of additional instances can help to meet QoS requirements but at a cost. We

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

have identified two: network traffic and instantiation time. The network traffic cost is the traffic generated by the transfer of the service image from the *Service Repo* to the *Hotspot* node (see Figure 4). It mainly depends on the size of service image and the bandwidth of network link. There are several approaches to estimate this cost, since the issue fall outside the scope of this deliverable we leave it aside and focus on instantiation time. The instantiation cost time is the time that that it takes to have a newly deployed instance ready for serving. In our experiments, we measured the booting time of the docker engine on a RPi and the time it takes to create a container. Figure 22 shows the service instantiation cost of the four web server containers. It takes about 3.37 s to boot up a docker engine in a RPi 3. This cost is zero when the Docker engine is already running in the RPi. As for the containers, it takes about 1.42, 1.46, 1.44 and 1.48 s to instantiate the nano, busybox, nginx and tomcat web server containers, respectively.

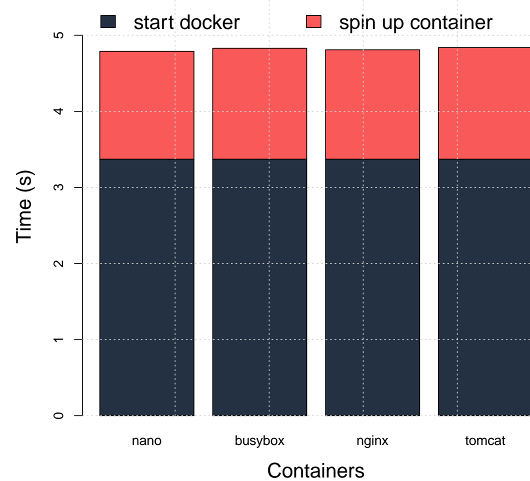


Figure 22: Container replication cost.

8 Fulfilment of QoS requirements

To recap, the service provider is responsible for deploying services with specific QoS requirements. The QoS requirements associated to services result from negotiations and contractual agreements drawn between the Service Provider and the Service Producer (see Figure 1). The Service Manager is the tool that the Service Provider has at his disposition to help him fulfill the QoS requirements associated with his services. He can use it to activate his QoS mechanisms. Figure 23 shows the relationship between the Service Manager and each mechanism. It also shows potential collaboration relationship among the mechanisms.

This project has received Funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

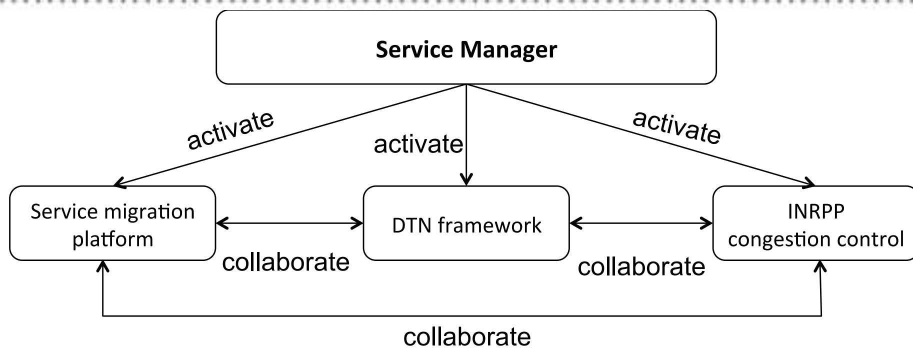


Figure 23: The Service Manager and its QoS mechanisms.

As shown in the figure, the Service Manager can activate each of the mechanisms to work individually; for example, it can activate the Service migration platform and nothing else. Alternatively, it can activate two of them and make them collaborate; for example, it can activate the Service migration platform and the DTN framework and make them collaborate. The Service Provider can devise different combinations to meet different QoS parameters associated to different classes of services.

Table 3 shows some examples. In the table, SM, DTN and INRPP stand for Service Migration platform, DTN framework and INRPP congestion control, respectively.

Classes of services	**QoS parameters**		
	Response time	Availability	Num. of packets loss
premium	SM + DTN + INRPP	SM + DTN + INRPP	INRPP
best-effort	[SM + DTN + INRPP]	[SM + DTN + INRPP]	[INRPP]
less-than-best-effort	DTN	DTN	INRPP

Table 3: QoS mechanisms used for supporting services of different classes with specific QoS parameters.

Each service is expected to meet thresholds of a set of QoS parameters such as response time, availability, number of packets loss, deployment time, etc. The value of a threshold varies from service to service, in this manner some services will demand low response time (for example, below 3 sec) whereas others will tolerate arbitrarily long response times. The same holds for availability, important services will demand high availability (of the other of 99.99) whereas other services will tolerate long down times (say, several hours).

A service with a stringent requirement of a QoS parameter will be classed as premium with respect to that specific parameter. For example, a service that is required to be

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

deployed within seconds of receiving a request will be classed as premium regarding deployment time. In contrast, a service that is expected to be deployed within 24 hrs of being requested will be classed as best effort regarding deployment time. It is worth emphasizing that a service with stringent deployment time does not necessarily has stringent response time. Equally important, a QoS mechanism that helps satisfy a given QoS parameter is not necessarily adequate to satisfy another. This is why we classify services as premium, best-effort and less-than-best-effort regarding specific QoS requirements. To mention a specific example, we can imagine an emergency service that is expected to be deployed within seconds (premium deployment time) after a disaster strike, offer a response time of 2-3 seconds (premium response time) and remain operational for as long as necessary (premium availability).

As a second example, we can think of a service used to deliver news in a remote rural area for free. Such a service is likely to be negotiated (between the service owner and the service provider) with relaxed availability requirements (less-than-best-effort) and no notion of response time or deployment time. To explain the potential use of the QoS mechanisms as shown in

Table 3, let us assume that the service provider is operating a network that is prone to suffer from congestions, partitions and unpredictable and arbitrarily long delays. Let us examine response time requirements firstly. We define response time as time perceived by the and-user. It is measured as the time elapsed between the placement of a request and the arrival of the last bit of the corresponding response. As suggested by the last row of the table, to meet relaxed response time requirements (less-then-best-effort class) the service provider would probably activate DTN alone. The service provider will resort to DTN to handle extreme situations where the end-users and the service are physically separated due to a total absence of communication infrastructures or temporary network partitions. In these situations, DTN will be used to eventually deliver requests/responses placed by end-users against the service, irrespectively of the locations of the service and the end-users. DTN can be deployed in a mobile device (for example a UAV) capable of carrying data between the two communicating parties.

At the other end of the spectrum, to support the operation of a service with stringent response time requirements (premium class), the service provider would activate SM, DTN and INRPP. INRPP will be used to remedy potential congestion problems that can impact response time. To reduce response time, SM will be used to deploy the service as close as possible to the end-users. In extreme situations where the end-users are left isolated in a network segment due to a network partition or where communication infrastructure is absent, DTN will be used to carry a binary image of the service and deploy it where the users are and instantiate it under the assurance that proximity to end-users will result in short response times.

Let us examine availability requirements now. Availability is defined as the readiness to deliver a correct service. Availability is measured as the percentage of the time that the service is delivering the expected service. Consequently, availability depends on the outage duration: *availability= ((agreed service time – outage time)/agreed service time)x100.* To support the operation of a service with availability requirements classed as less-than-best-effort, the service provider would probably activate DTN and nothing else. In scenarios where the communication infrastructure is absent or temporarily impaired by a network partition, DTN can be used to transport data (requests placed against the

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

service and responses sent to the end-users) between the two communication parties and make the service to remain available. Due to the long response time, the service outage will be extremely long and cause a low availability, but for a less-than-best-effort class, this availability should be acceptable. As shown in the table, services that require high availability (say, in the region of 99%) are very likely to be classed as premium services and supported by the activation of SM, DTN and INRPP operating in collaboration. As before, in scenarios with no network infrastructure or impacted by network partitions DTN can be used either as a transport (mule) mechanism to ferry requests/responses between the end-users and the service or alternatively to transport an image of the service, download it where the end-users are and instantiate it to make the service available. A parameter that impacts the availability of service is the time it takes to deploy it. Recall that to deploy a service, its image needs to be transferred from a repository (or cache) to the hotspot where it is instantiated. The transfer time is determined mainly by the size of the image of the service and the congestion of the network used for transferring the image. To minimize transfer time of services with premium availability requirements, the service provider is likely to activate his INRPP and instruct it to differentiate classes of services and prioritize the transfer of services classed as premium. A thorough analysis of this measure is discussed in the example of Section 8.3.2.

Finally, let us examine packet loss requirements. A packet is considered lost if it is dropped by an intermediate node before the packet reaches its final destination. Packet loss is measured as a percentage of packets lost with respect to packets sent. Packet loss rate is particularly high when the network is congested unless congestion control mechanisms are in place. As shown in

Table 3, to keep the rate low for services classed as premium regarding number of packet loss, the service provider is likely to activate INRPP alone. Packet loss is a network issue that can be naturally handled by QoS mechanisms such as INRPP. One can argue that SM can be used to reduce the rate of packet loss by re-deploying the service as close as possible to the end user to reduce network traffic, though we do not preclude this alternative we regard it as an indirect measure to address the problem at hand—this is why we do not include SM in the fourth column. Regarding services classed as less-than-best-effort regarding number of packets loss, the service provider might, as a precaution activate INRPP. It is worth clarifying that there are no technical difficulties to activate INRPP as a precaution measure in the second and third columns of the less-than-best-effort as well. We did not include it in the table because this measure is optional. Similarly, and as mentioned in Section 5, one can rightly argue that DTN can be used as a congestion control mechanism (to carry some traffic load of low priority) as well to prevent packet loss of premium services. We did not include it in the table because INRPP is more adequate than DTN for addressing this issue.

This discussion is meant to give ideas about how the QoS mechanisms can be used. The three QoS parameters that we use in the discussion are only examples. There are others parameters (for example, throughput and time to repair) that are likely to demand different combinations of the QoS mechanisms. Observe that we suggested mechanisms for dealing with the most contrasting classes of services (premium and less-than-best-effort). Best-effort services fall somewhere in between these two classes. As such, to handle them, the service provider will deploy mechanisms at its discretion depending on how he regards best-effort. For instance, he can be lenient, made no commitments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

towards their QoS expectations and use the same QoS mechanisms as for less-than-best-effort. On the other extreme and as suggested in the table by the parameters shown in brackets, the service provider might regard best-effort services nearly as important as the premium ones and deploy similar QoS mechanisms for handling them.

From the above discussion it follows that a crucial piece of information that the Service Manager needs to know about the services under its responsibility is their classes (premium, best-effort, less-than-best-effort) which can be retrieved from the service description associated to each service. On the basis of this information the Service Manager determines how to operate its Decision Engine. For instance, the Service Manager might follow a policy that to deal with services classed as less-than-best-effort, it launches the Decision Engine only to deploy the first instance of the service and hope for the best. Similarly, to deal with services classed as best-effort, the Service manager, will launch the Decision Engine to deploy the first instance of the service and occasionally, but leniently, re-launch it to assess the situation and deploy additional instances of the service if needed in an attempt to keep the service running smoothly. However, to deal with services classed as premium, the Service Manager is likely to launch the Decision Engine to deploy the first instance of the service and re-launch it frequently enough to keep track of the operation of the running instances and deploy additional ones as needed to guarantee its QoS. The categorization of services into discriminatory classes is also central to the Decision Engine. Its algorithms prioritize attention in the sense that problems related to premium services are addressed before problems related to best-effort and less-than-best-effort services. For instance, instances of premium services are deployed before the Decision Engine pays attention to instances of lower priority.

8.1 Handling numbers of concurrent requests with local replication

In this example, we use the service migration platform operating as a reactive service migration mechanism. The example shows how the Service Provider can use the Service Manager to operate individually (without support from the DTN framework or INRPP) to meet requirements imposed on the number of concurrent requests that a service needs to handle. The service is assumed to be of premium class.

Let us assume that the Service provider is made responsible for the provisioning of a busybox web service for the benefit of end users located near HS_1 (Figure 4). He is expected to honour the following QoS requirements:

- ***Requirement1:*** *The service shall provide a response time no larger than 500 ms.*
- ***Requirement2:*** *The service shall provide availability of 99.99 %*
- ***Requirement3:*** *The service shall be able to handle at least 700 concurrent requests.*
- ***Requirement4:*** *The service shall be classed as premium service.*

To comply with its obligations, the Service Provider proceeds as follows:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

1. The Service Provider builds the *Sbusybox_spec.json* shown below on the basis of the QoS requirements (*Requirement1, … ,Requirement4*) and experimental results of Figure 6. From these results, he knows that the number of concurrent requests that the busybox service can handle without showing signs of exhaustion is 500. Likewise, from the results of Figure 17 he knows that two instances of a busybox container created in a single Pi can offer an average response time below 200 ms. The Service Provider includes these two parameters in the *Sbusybox_spec.json* file.

```
S_busybox= {
    'par':{
        'serviceName':  'nano',
        'imageName':    'hypriot/rpi-busybox-httpd',
        'imageSize':    '88',
        'maxConReqs':     '500',
        'startUpTime':  '5'
        },
  'QoS':{
        'responseTime':  '500',
        'availability': '99.99',
        'numConReqs":     '700'
       },
 'class':{
        'serviceClass': 'premium'
       }}
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

2. To deploy the first instance of the busybox service, the Decision Engine needs to select a Hostspot that can handle it. In pursuit of this aim, the Decision Engine executes the following algorithm:

```
selectHost_to_deploy_firstInstance(json_lst_dict, json_server_Spec)

        # json_lst_dict: json list of dictionaries with aggregated monitored data

        #json_server_Spec: server specification in json
```

As a result, the function produces:

```
The host selected to deploy first instance is: SEG_1
```

which suggests SEG_1 to the Decision Engine.

3. As a precaution, the Decision Engine can verify the number of containers running in SEG_1.

```
get_number_of_containers_of_pi('SEG_1')
```

In this example, the output from the function is:

```
Pi identified as SEG_1 is running 0 containers
```

If necessary the Decision Engine can also verify the status of the Pi with SEG_1 ID with the help the following function:

```
get_pis_status(json_lst_dict, 'SEG_1')
```

The output from the function is:

{"softResources": {"OS": "Linux"}, "hardResources": {"mem": "1 GB", "disk": "16 GB", "cpu": "A 1.2GHz 64-bit quad-core ARMv8 CPU"}, "resourceUsage": {"cpuLoad": "0.07", "memUsage": "12", "cpuUsage": "0.374996785177"}, "PiID": "SEG_1", "PiIP": "192.0.0.2", "containers": [] }

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

Observe that list of containers is empty, which indicates that SEG_1 is currently running no containers. Consequently, SEG_1 is definitely selected to host the instantiation of the service. In addition, the Decision Engine can find out what is the Pi with the lowest cpuLoad:

```
get_pis_with_min_cpuLoad(json_lst_dict)
```

The execution of the function above will output the following response

```
PiID with min cpuLoad
PiID= SEG_1 cpuLoad= 0.01
```

4. Let us assume that the Decision Engine decides to deploy the first instance of the busybox service in SEG_1. It proceeds to produce the following *json obj with deployment* descriptor containing the deployment specification of the image of the busybox service.

```
deploy_descriptor = {' rpi-busybox-httpd.tar':
                        {'image_name':'hypriot/rpi-busybox-httpd:latest,
                 'port_host':8081,
                 'port_container':80}}
```

5. The Decision Engine executes the following function to send the deployment descriptor and the image of the busybox service to the Service Execution of HS_1.

```
deploy(HS_1, 'rpi-busybox-httpd.tar', deploy_descriptor)
```

The Decision Engine can verify the number of container currently running in SEG_1:

```
get_number_of_containers_of_pi('SEG_1')
```

In this example, the output from the function is:

```
Pi identified as SEG_1 is running 1 containers
```

If necessary the Decision Engine can also verify the status of the Pi with SEG_1 ID with the help the following function:

```
get_pis_status(json_lst_dict, 'SEG_1')
```

The output from the function is:

{"softResources": {"OS": "Linux"}, "hardResources": {"mem": "1 GB", "disk": "16 GB", "cpu": "A 1.2GHz 64-bit quad-core ARMv8 CPU"}, "resourceUsage": {"cpuLoad": "0.08", "memUsage": "13", "cpuUsage": "0.366414641056"}, "PiID": "SEG_1", "PiIP": "192.0.0.2", "containers": [{"status": "Up About a minute", "port_host": "8002", "memUsage": "2023424", "name": "/busybox-FirstInstance", "image": "hypriot/rpi-busybox-httpd:latest", "port_container": "80", "id": "dd8aa8df26370358188b5283c828c919f393c300d25161d61a5d07086905db0e", "cpuUsage": "39956775"}]}

Observe that list of containers has one element, namely a container named busyboxFirstInstance which corresponds to the instance previously created.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

6. From the results shown in Figure 6, the service provider knows that an instance of a busybox service can handle 500 concurrent requests only. Since the service is premium class (*Requirement4*) the Service Manager decides to play safely and deploy another instance immediately, that is, without waiting to see signs of exhaustion of the recently deployed instance. The question that arises here is where to deploy the second instance: local replication (collocated with the first instance in HS_1) or remote replication (in an alternative HS node such as HS_2). The Decision Engine executes the following algorithm to verify if SEG_1 can be used to instantiate the second instance of the busybox service.

```
selectHost_to_deploy_additionalInstance(json_lst_dict, json_server_Spec, PiID)

# json_lst_dict: json list of dictionaries with aggregated monitored data

#json_server_Spec: server specification in json
```

As a result, the function produces:

```
The host selected to deploy an additional instance is: SEG_1
```

which suggests a local replication, that is, a deployment in SEG_1.

As shown in Section 4.1.4, the algorithm takes into account the results shown in Figure 9. From these results the Service Provider knows that HS_1 is capable of hosting two instances of the busybox service. The results show that the deployment of the second instance will not be detrimental as it will not afflict a cpu load above 1.5. As a precaution measure, the Decision Engine can always execute the following function to determine how many containers are currently running in SEG_1.

```
get_number_of_containers_of_pi('SEG_1')
```

7. Let us assume that the Decision Engine decides to deploy the second instance of the busy-box service locally, that is, in SEG_1. Then the Decision Engine produces the following *json obj with deployment* descriptor that contains the parameters for the instantiation of an additional instance of the busybox web service.

```
deploy_descriptor = {' rpi-busybox-httpd.tar':
                          {'image_name':'hypriot/rpi-busybox-httpd:latest,
                          'port_host':8084,
                          'port_container':80}}
```

8. The Decision Engine sends the *json obj with deployment* to the Service Execution of the HS_1 node. Note that, the Decision Engine specifies a 'port_container' that is free ('8084'), port 8082 is already in use by the first instance.

```
deploy(SEG_1, ' rpi-busybox-httpd.tar', deploy_descriptor)
```

Observe that the image of the web service doesn't need to be sent since HS_1

This project has received Funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

already has them. However, the deploy_descriptor is required to be delivered, since the description to deploy the service is updated.

9. The Service Execution of HS_1 creates another instance of the busybox service. This instance can handle another 500 concurrent requests. The two collocated instances of busybox should be able to meet *Requirement3.*

   As a result, two instances are running in the HS node, one in port 8002 and another one in port 8004.

10. The Decision Engine can verify the number of container currently running in SEG_1:

    get_number_of_containers_of_pi('SEG_1')

    In this example, the output from the function is:

    Pi identified as SEG_1 is running 2 containers

    If necessary the Decision Engine can also verify the status of the Pi with SEG_1 ID with the help the following function:

    get_pis_status(json_lst_dict, **'SEG_1'**)

    The output from the function is:

{"softResources": {"OS": "Linux"}, "hardResources": {"mem": "1 GB", "disk": "16 GB", "cpu": "A 1.2GHz 64-bit quad-core ARMv8 CPU"}, "resourceUsage": {"cpuLoad": "0.04", "memUsage": "13", "cpuUsage": "0.365180685999"}, "PiID": "SEG_1", "PiIP": "192.0.0.2", "containers": [{"status": "Up 30 seconds", "port_host": "8003", "memUsage": "221184", "name": "/busybox-SecondInstance", "image": "hypriot/rpi-busybox-httpd:latest", "port_container": "80","id":"78c7f9050888ec9abf5868fb501f67abd477542f63a52f2097bff707e8e47a2e", "cpuUsage": "56309687"},{"status": "Up 5 minutes", "port_host": "8002", "memUsage": "1859584", "name": "/busybox-FirstInstance", "image": "hypriot/rpi-busybox-httpd:latest", "port_container": "80", "id": "dd8aa8df26370358188b5283c828c919f393c300d25161d61a5d07086905db0e", "cpuUsage": "39956775"}]}

    Observe that the list of containers has one element, namely a container named busybox-SecondInstance which corresponds to the instance previously created.

11. On the basis of the results from Figure 6, the service provider can confidently expect that the two instances of the busybox service running in the SEG_1 Hotspot will comply with *Requirement3* and simultaneously with the requirement of a premium service (*Requirement4*).

8.2 Meeting of availability requirements with DTN support

In this example we use the service migration platform operating as a reactive service migration mechanism. The example shows how the Service Manager can collaborate with the DTN framework to meet availability requirements demanded by an emergency service classed as premium.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

8.2.1 QoS requirements and scenario

The Service provider is assumed to be responsible for the provisioning of a busybox web service :

- ***Requirement1:*** *The service shall be available regardless of the network conditions in the area.*
- ***Requirement2:*** *The service shall be in operation within 60 sec after requested by end users.*
- ***Requirement3:*** *The service shall be classed as premium.*

To show how we address the challenge with the help of the service migration platform and the DTN framework, we use the scenario of Figure 24. As in the previous example, we assume that the Service Manager builds a service description on the basis of the information that he has about the service and the three requirements mentioned above. The challenge in this example, is to meet the requirements (*Requirement1 ,... , Requirement3*) when potential network problems (network partitions) materialize.

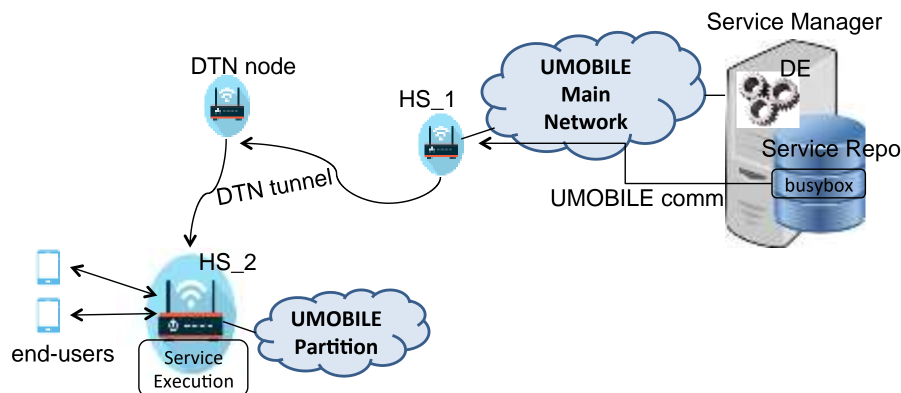


Figure 24: Integration of Service Migration and DTN to meet availability requirements.

The figure is a simplified version of the Service Migration platform shown in Figure 4. It includes only the components needed to demonstrate how the QoS requirements mentioned above can be meet.

The Service Manager includes a Decision Engine and a Service Repository (Service Repo) but it does not include a Monitoring Manager or Monitoring Agents. Let us assume that HS_1 is at the edge of the network. HS_2 is miles away from HS_1 and disconnected from the main network, say due to a temporal network partition. The DTN node is physically mobile in the sense that it can travel backwards and forwards between HS_1 and HS_2; it can be physically attached to an UAV, but other means of transport will work as well. Busybox is the service that the Service Provider is expected offer to under the observance of QoS requirements mentioned above. Though we use RPi-3 to realize HS_1, HS_2, DTN node and Service Manager, the hardware is irrelevant. The RPi-3 used for the Service Manager is deployed with the UMOBILE platform discussed in Deliverable D3.3. However, HS_1 and HS_2 are deployed with the platform shown Figure 25. Figure 25 is an

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

augmented version of the UMOBILE platform aimed at integration with the DTN-framework, such components are highlighted in blue.

8.2.2 Service migration-DTN integration architecture

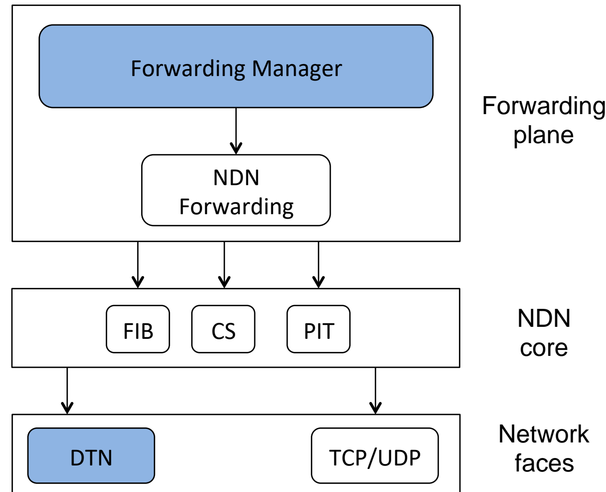


Figure 25: UMOBILE components involved in the integration of service migration and DTN.

To make the service migration platform collaborate with the DTN framework, we use the components of Figure 25 to integrate their functionalities. The figure resulted from the enhancement of the original NDN architecture with additional services including delay tolerance and QoS awareness. These services can be used to provide QoS and reliability in environments afflicted by intermittent disconnections. The new components developed in the context of UMOBILE are highlighted in blue.

In brief, the central idea is to operate the Decision Engine of Figure 24 in listening mode to listen to end-user's requests for services (for example busybox). These requests arrive as Interest messages. In response to a request, the Decision Engine fetches the corresponding image from the Service Repo and sends it back to the SEG requesting it (for example, to HS_2) where the local Execution Service instantiates it.

Forwarding plane: It is deployed in both, HS_1 and HS_2. It implements the logic that decides how nodes forward Interests towards their final destination. The *Forwarding Manager* is responsible for choosing the communication model (i.e., push or pull–based) that best meets the requirements of the application and the forwarding strategies. NDN provides several strategies such as Best Route (aggregated in NDN Forwarding) but it also allows customised strategies.

NDN natively follows the synchronous communication model where a consumer sends an Interest message and waits for the corresponding Data before sending another Interest. In addition, the maximum size of a Data message is set to 8 kB. These constraints are not suitable in disruptive scenarios where there is only a single communication link between the consumer and producer. As shown in our example (Figure 24), a DTN node (e.g., a raspberry Pi on board of an UAV) used for deploying a large server would need to fly several times between HS_2 and the main network to retrieve and deliver all the pieces of data chunks. For example, to transfer a busybox image of 2.16 Mbytes (see Table 2), the UAV would need 270 trips (*trip = image_size/Data_message_size*) which is impractical. To

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

remedy the problem, we implemented an *asynchronous multi-Interest forwarding model*: a consumer sends several Interest in a burst and waits for the corresponding data chunks to arrive. In the example, the RPi is capable of aggregating both the Interests and the data chunks. In this example, we aggregate 30 Interests (burst_size) in the second message as a result the number of trips that the UAV needs to make is only 9 (*trip = image_size/(burst_size * Data_message_size)*).

NDN core: aggregates the network services and functional data structures included in the original NDN architecture such as FIB (Forward Interest Base), CS (Content Store) and PIT (Pending Interest Table).

Network face: is the generalisation of a communication interface and covers physical and software interfaces. We have implemented the DTN face (also called DTN tunneling) that supports communications in environments that suffer from intermittent connectivity. It relies on store-and-forward and can operate over diverse underlying technologies. It supports data mulling and custody transfer services.

8.2.3 Practical demonstration

To verify that the integration of the service migration platform and the DTN framework can be used to deploy the busybox web server in the HS_2 when the latter is affected by a network partition (see Figure 24) we have conducted laboratory experiments. To simplify the equipment involved and focus on the core functionality of the solution, we have used the laboratory settings shown in Figure 26.

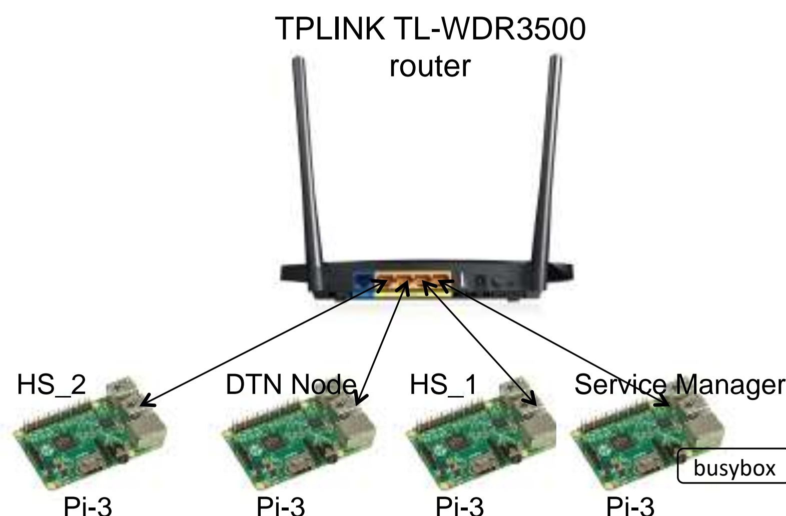


Figure 26: Laboratory settings for proving integration of service migration and DTN.

As shown in the figure, we used RPi-3 single board computers for realizing HS_1, DTN Node, HS_2 and Service Manager. Busybox is the Dockerised image of the busybox service and is stored in the Service Manager. The router is a commercial router manufactured by TP-Link. To simulate the network partition of the left side of Figure 24, we proceeded as follows:

i. We cable–connected HS_1, DTN node and Service Producer to a router. Pi, HS_1 and Service Producer can communicate with each other through the router. HS_2 remains unplugged to simulate that it was impacted by a network

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

partition that rendered it isolated.

ii. We unplugged HS_1 and Service Producer from the router and plugged HS_2. The effect is that DTN node has joined HS_2 in its partitioned network. Pi and HS_2 can communicate with each other through the router. This connection arrangement is equivalent to physically proximity between the HS_2 and the DTN node if the latter were mobile, say attached to an UAV.

iii. We returned to step i, and repeated the procedure as many time as necessary to transfer all the data chunks of the busy-box service.

At Interest message level, the solution takes advantage of the name-based routing of NDN to select forwarding strategy based on names. It operates as follows:

1. Let us assume that HS_2 is currently impacted by a network partition (step i). HS_2 receives a user's request for the busybox service. In response, the Forwarding Manager of HS_2 issues an Interest message with a name prefix */dtn/service/emergency* to request the service through its DTN face.

2. HS_2 forwards the Interest to the DTN node when DTN node becomes reachable to HS_2 (step ii). This is done through a DTN static link that we set between HS_2 and HS_1 via the DTN node. The static rule at HS_2 is set, by adding the following line to the ibr-dtn configuration file:

   route1 = dtn://HS1/[[:alpha:]] dtn://UAV.dtn

   The ' dtn://* ' parameters are the relevant DTN Endpoint IDs (EIDs). To be reachable, we also have to configure the local HS_2 NFD in order to register HS_1 as the next hop FIB entry for the namespace serving our Interest. This can be done by using the following command on a terminal, to configure NFD:

   nfdc register /dtn/service/emergency dtn://HS1/nfd

3. Although we use static routing in this laboratory experiments, several DTN routing algorithms can be enabled to support more sophisticated scenarios.

4. When the DTN node is able to communicate with HS_1 (step i), it delivers the Interest message to the latter. The NDN Interest packet has been encapsulated in one or more DTN bundles. When the two nodes (HS_1 and UAV, both running DTN implementations) come into contact, the DTN node ("UAV") delivers the bundles to its destination (HS1) by examining its EID - essentially closing the tunnel. After receiving the bundle(s), they are decapsulated and the resulting Interest packet can finally be processed by HS_1.

5. After examining the Interest, HS_1 tries to retrieve the busybox service from caches and through name-based routing.

6. The Service Manager or another intermediate node in possession of the image replies with one or more Data messages along a reverse path and using a delay-

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

tolerant forwarding strategy. A namespace dedicated to delay-tolerant Interests can optionally be used to trigger delay-tolerant forwarding strategies, as a per-namespace strategy selection is made by the intermediate forwarders. The naming, thus, can serve as an API.

7. The first Data message sent in response to an Interest is augmented with information about the total size (e.g., 1MB) of busybox. This allows HS_2 to determine how many Interests it needs to send to the Service Manager to retrieve busybox. HS_2 appends chunk IDs to the name prefix of each subsequent Interest, for example the prefix of a second Interest message is expressed as %02.

8. Again, when needed, the subsequent Data packets are forwarded through the DTN face (i.e. are encapsulated in DTN bundles and delivered to nodes according to their EIDs). The current implementation has to ensure that the FIB entries still exist by setting a large Interest Lifetime, so that the subsequent Data packets can follow the breadcrumbs route back to the consumer (HS_2).

9. When HS_2 receives all the chunks of busybox, it calls its Service Execution function to execute it. Observe that the number of cycles (trips of the DTN node between HS_2 and HS_1) depends on the size of the service.

8.2.4 Discussion of results

After repeating the cycle 9 times, the image of busybox, originally located in Service Manager, was instantiated in HS_2 and able to respond to users' requests. The availability of busybox in the partitioned network satisfies *Requirement1*.

Observe that in this experiment, HS_2 is free from running other instances. However, if HS_2 had other instances running, we would have taken into account the current status of the resources of HS_2 and verified its capability to run another busybox instance. This can be done with the help of a Monitoring Agent and Monitoring Manager as shown in Figure 4. Figure 9 shows that two instances of the busy-box service can comfortably run in the Raspberry Pi that we use for realizing HS_2, without inflicting a cpu load above 1.5.

In the same order, note that for the sake of simplicity, the experiment does not take into account the number of users requesting the busy-box service, consequently, the Service Provider deploys a single instance. In accordance with the results shown Figure 6, this instance should be able to handle about 400 concurrent requests without exhibiting signs of exhaustion. However, if an unexpectedly large number of requests arrive to exhaust the instance, the Service Provider can react by deploying a second instance to ease the load inflicted on the first one.

On the other hand, the service provider should have no difficulties in meeting the requirement on the deployment time (see *Requirement2*) since from Table 2 he knows that the size of the busy-box is 2.16 Mbytes which amount to 270 trips of the DTN node between the HS_2 and HS_1 to transfer the busy-box image. Once all the chunks of the image of the busy-box service are in HS_2, it costs, in accordance with Figure 12, less than five seconds to have to busy-box fully operational: about 3.3 seconds to start Docker and about 1.4 seconds to spin the container up.

The results prove that the solution can indeed be used to deploy services regardless of

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

network conditions or network coverage. It can also be used for deploying delay–sensitive services as close as possible to the end-users. Note that the busybox service of the example is assumed to be self–sufficient, i.e., capable of working independently from the main network. Services that need support from the main network can also be deployed, provided that the DTN node is used for transporting data between the server and the main network. Likewise, in the absence of hosting facilities in the partitioned network, the DTN node can be used to data–mule Interests and Data messages between the end users and the server deployed in the main network. However, in this setting the Pi will inevitably cause delays which can be either time constrained or arbitrarily long. In the latter case, the solution can still be used but for less-than-best-effort services and best-effort services with response time requirements. The observation here is that in practice the service provider is unlikely to agree to deploy premium services with stringent response time requirements on a network prone to failures.

8.3 Meeting of availability requirements with INRPP support

In this example we use the service migration platform operating as a proactive service migration mechanism (pre-fetching). The example demonstrates the potential use of the service migration platform in collaboration with the INRPP flowlet congestion control to migrate the service in advance before user request is generated. Unlike the reactive migration, in this example we do not consider the runtime resource exhaustion of the hosting hardware (e.g., hotspot). As the main benefit of service pre-fetching is to cache the service at the hotspot close to the end users and make it available for instantiation when needed, the deployment time is needed to be optimised to enhance the availability. The example assumes that the service provider is running services of different classes. Thus, INRPP is activated with its priority differentiation mechanisms in action (premium and best effort). The example includes evaluation results produced by the ndnSIM simulator [13].

8.3.1 Service migration-INRPP integration architecture

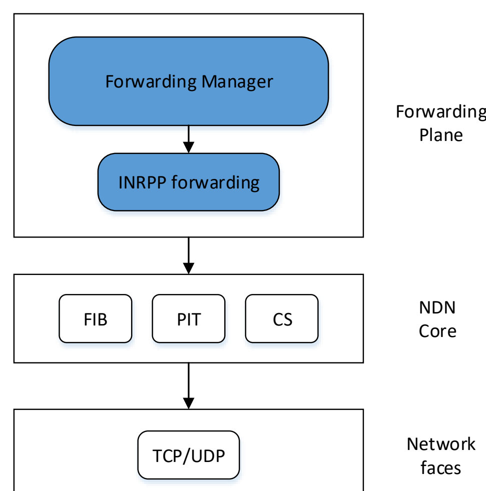


Figure 27: UMOBILE components involved in the integration of service migration and INRPP.

To make the service migration platform collaborate with the INRPP congestion control, we use the components of Figure 27 to integrate their functionalities. The figure resulted from the enhancement of the original NDN architecture with additional services including

This project has received Funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

delay tolerance and QoS awareness. These services can be used to provide QoS, hop-by-hop congestion, multipath communications control and service priority with INRPP. The new components developed in the context of UMOBILE are highlighted in blue. Basically are the same elements as in the service migration-DTN solution, except that the INRPP is implemented in the forwarding manager and includes the new congestion control in the forwarding strategy. INRPP needs to be deployed all any devices of the UMOBILE network, including the hotspots and the Service Manager.

8.3.2 QoS requirements and scenario

The Service provider is assumed to be responsible for the provisioning of a busybox web service:

- ***Requirement1:*** *The service S2 shall be available with full priority in contrast to other best-effort services like S1.*
- ***Requirement2:*** *The service shall be in operation within 0.5 sec after requested by end users.*
- ***Requirement3:*** *The service S2 shall be classed as premium and S1 as best-effort.*

The challenge in this example is to meet the deployment time of S2 as stipulated in *Requirement3* ---a crucial requirement to the service provider because S2 is classed as a premium service. Let us assume that the Service Provider is in possession of information that indicates that S2 and S1 will be needed in HS1 and HS2, respectively. On the basis of this hint and to avoid the risk of failing to meet *Requirement2* the Service Provider decides to pre-fetch the images of S1 and S2 with the assistance of the INRPP congestion control with its priority mechanism switched on.

To show how the Service Provider addresses the challenge, we use the scenario of Figure 28. As in the previous example, we assume that the Service Provider builds a service description on the basis of the information that he has about the service and the three requirements mentioned above.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

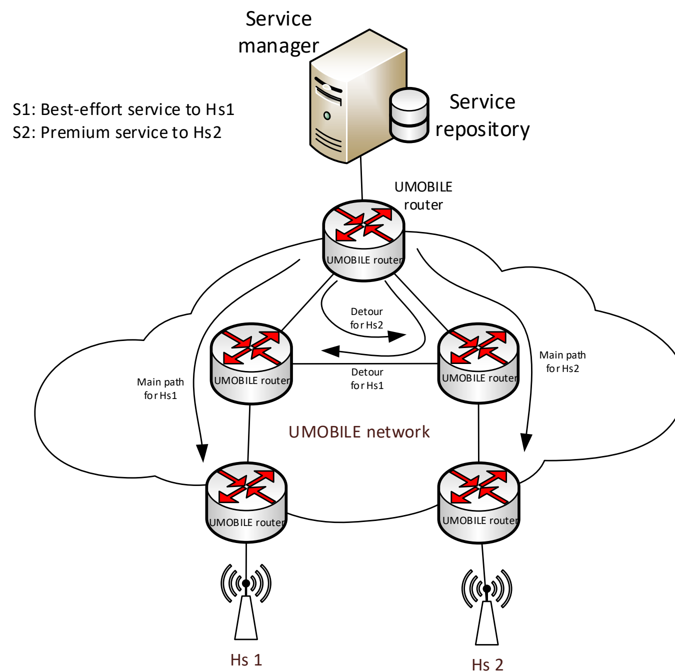


Figure 28: Integration Service Migration and INRPP to meet the QoS requirements.

The challenge in this example, is to meet the requirements (*Requirement1, ... , Requirement3*) even when congestion occurs in the UMOBILE network. In this case we set up an scenario according to Figure 28 using ndnSim [13] with the INRPP implementation. In the simulation we have two services, a best-effort service S1 and a premium service S2. These two services are treated differently in the cache that is acting as a temporary custodian for the congestion control giving priority to the premium service.

8.3.3 Service migration-INRPP evaluation results

In this section we include a set of experiments to evaluate the benefits of using the service migration platform along with INRPP to offer differentiated services that distinguish and treat service of different classes (premium, best-effort, less-than-best-effort) differently.

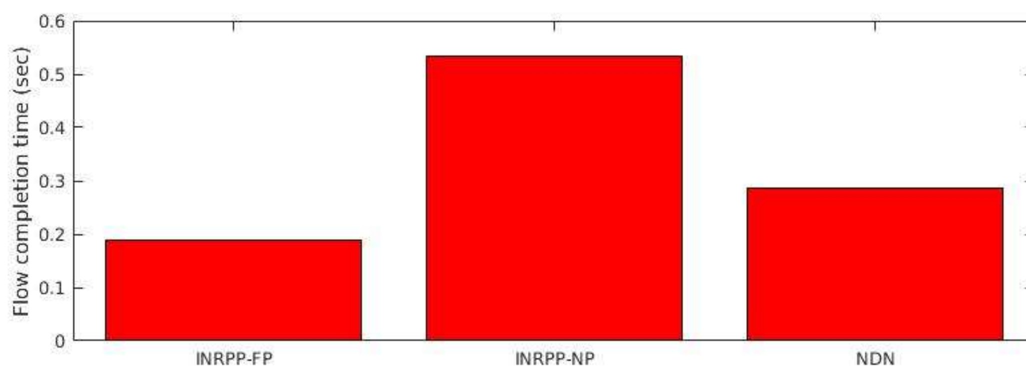


Figure 29: Average flow completion time for two docker images using INRPP with full priority (FP) and no-priority (NP) compared with random flows using plain NDN.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

The example evaluates the scenario depicted in Figure 28, where all links are 100Mbps except the link to the Service Manager which is 1Gbps. We evaluated how fast we can transfer two busybox web-service images (S1 and S2) of 2.16 Mbytes from the Service repository to HS1 and HS2, respectively. We use INRPP with no priority (INRPP-NP in Figure 29 and Figure 30) for service S1 then we use INRPP with full priority (INRPP-FP in Figure 29 and Figure 30) for service S2. In the simulation we added NDN random background traffic without using INRPP and pareto distributed flow sizes with shape equal to 1.2 and E[L]=45KB, where L is the size of the flow. In Figure 29 we can observe the flow completion time, i.e., the time necessary to transfer the image to the hotspot that wants to instantiate the service, and therefore, how fast we can move the service to the edge of the network. In this case we know that there is a requirement of 0.5 sec (*Requirement2*). In the figure, we can see that the transfer time of S2 is slightly below 0.2, achieving the maximum operating latency by far. In contrast, the transfer time of S1 is slightly above 0.5, having a transfer time close to twice of that of S2 because it is not a prioritized service. Here we can also observe the average completion time for normal NDN flows, seeing that despite being very small flows in average (E[L] = 45KB) the flow completion time is greater than of S2.

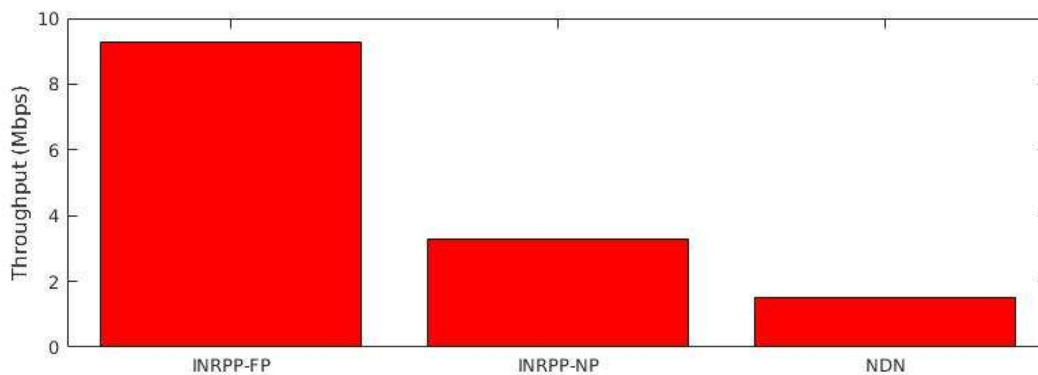


Figure 30: Average throughput for two docker images using INRPP with full priority (FP) and no-priority (NP) compared with random flows using plain NDN.

In Figure 30, we can observe the average throughput for the same evaluation analyzed in the last paragraph. In this case, we can see that the throughput achieved by S2 is close to three times (90Mbps vs 30Mbps) S1. Here we can see that S2 can take all the bandwidth available, while S1 is sharing the bandwidth with the rest of flows. However, we can see that S1 is getting more throughput than NDN flows, because it is able to use the residual bandwidth available in the detour path that regular NDN is not using because it is a single path solution.

Once the image of the premium service S1 is cached in HS1, the service provider can instantiate it, in no time, when it is eventually requested in no time to comply with *Requirement2*. Note that the image of S2 might also be cached in HS2 before it is needed, but this is secondary because S2 classed only as a best-effort service.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

9 Concluding remarks

Increasingly, network providers are responsible for delivering application services to the End-users and effectively become service providers. In different circumstances, these services are associated to QoS expectations (availability, response time, etc.) that the network provider is expected to honour in behalf of the service producers. With a large number of services, the task is hard to perform unless automatic tools and techniques are available. Several mechanisms for addressing QoS have been suggested in the literature (for example see [14]) but the topic is still an open research question as services with more stringent QoS requirement are being developed and new technologies are available to devise solutions. This deliverable discusses three QoS mechanisms that the UMOBILE project has developed to support network providers in delivering different classes of QoS: less-than-best-effort, best-effort and guaranteed QoS (premium). Comparatively, less-than-best-effort and best-effort QoS are simpler to deliver. We have focus our effort on the delivering of guaranteed QoS because it is far more challenging; it demands systematic approaches like the one taken in this document. A major strength of the approach taken by the UMOBILE projects is its collaborative nature. The mechanisms that we have developed work at different levels of the software stack (from the application to the networks layer) independently from each other but can be integrated to collaborate in the fulfillment of common QoS goals.

The salient feature of the Service Migration platform, which operates at the application level, is that it takes advantage of recent progress in light virtualization technologies. It relies on opportunistic caching and replication of service instances to meet QoS expectations. It is aimed at staless services that can be quickly deployed, redeployed from scratch and discharged when they are no longer needed. The Service Migration Platform accounts for local and remote replication: a new replica of a service is collocated with existing ones in the same host or instantiated in a different host, all depending on the current status of the resources of the host.

The Service migration platform can be operated either as a proactive service migration mechanism (also known as pre-fetching) or reactive service migration mechanism.

The proactive service migration approach aims at caching images of services in convenient locations before the images are actually needed. From this perspective, it operates independently from the actual instantiation of the service and its delivery. It relies on information about the imminent need of the service in the near future. Though this delivery does not elaborate about how this information is made available to the Service Provider we can mention in passing that this information can be collected from statistical records about service usage.

The reactive service migration approach can be used at service delivery time to react to signals about resource exhaustion. The algorithms that decide on service deployment strictly rely only on information available to the service provider, that it, they do not need to disturb the end-users or instrument end-users' devices to retrieve information from them. Neither do they rely on information provided by dummy users deployed to take

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

performance measures from the end-users' side. The monitored data used by the Decision Engine is about the status of the resources (specifically, the Pis that implement the Hotspots and containers) as opposed to the activities of the end-users. The deployment algorithms use a priori collected information about critical performance thresholds of the services such as their exhaustion points. We have collected this information from laboratory experimentations. For example, we deployed web services and exposed them to heavy loads of requests to find out how many concurrent requests they can handle smoothly. Likewise, we instantiated replicas of services in HS nodes (Raspberry Pi-3 computers) to measure how many instances a Pi-3 computer can run without showing signs of exhaustion. Though this experiment-based approach is time consuming, we believe that it is far more realistic than analytical approaches as our deployment decisions are based on facts. At this stage of the development of the service migration platform the focus is on the current status of the resources of the Hotspot nodes to decide on local or remote replication. Thus deployment of instances takes into account Hot Spot resources, we do not take into account yet other factors such as network resources other than being or not being available as in the example of the integration of the Service Migration Platform with the DTN-framework. For example, the deployment algorithms do not try to optimize the location of the instances, say to reduce traffic. Neither do they consider other optimization techniques such as service prioritization, replica swapping, discharging, and so on. It is worth clarifying that the results from the laboratory experiments that we have performed are valid for the specific hardware and software that we use. However, we believe that the methodology is general enough and can be followed in other settings. We hope that other researchers and industry engineers responsible for addressing QoS issues will benefit from our experience.

The salient feature of the DTN framework is its inherent ability to support communications in adverse networking conditions, be it severe disconnections and long delays, or highly congested links. Furthermore, the reliability features of the DTN framework promote efficiency, since the UMOBILE platform can use large data packets instead of multiple smaller ones. This can be beneficial in cases when data must be forwarded via data mules, such as UAVs, or mobile nodes. Finally, service providers can support a less-than-best-effort service, by using the DTN forwarding mechanism, relying on its ability to forward packets in a delayed fashion, allowing more network resources to more important traffic.

The salient feature of the INRPP congestion control is that it can operate at the network level in the background to mitigate congestion problems that can eventually propagate to the application layer and impact the QoS of the applications. It is capable of differentiated different classes of services (premium, best-effort and less-than-best effort) at network level and prioritise traffic accordingly.

A central argument of the UMOBILE work is that QoS requirement can be conveniently addressed by a collaboration of mechanisms that operate at different levels of the software stack. They key idea is to manipulate parameters where they are most visible.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

To support this argument, we included an example that shows the collaboration of the Service Migration Platform with the DTN-framework and INRPP.

10 References

[1] UMOBILE D3.1 Deliverable, Umobile architecture report (1), Sotiris Diamantopoulos et. al., Technical Report, EU UMOBILE project (grant No. 645124), May 2016.

[2] Matt Masnider et. al., Microsoft Azure: Reliable Services overview, https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-introduction (visited in Jul 2017).

[3] Evangelos Kotsovinos and Tim Moreton and Ian Pratt and Russ Ross and Keir Fraser and Steven Hand, Global-scale Service Deployment in the XenoServer Platform, Proc. USENIX: First Workshop on Real, Large Distributed Systems, 2004.

[4] R. Wetzel, *CDN business models - not all cast from the same mold*, http://www.wetzelconsultingllc.com/CDNArticle.pdf, Oct 2001, Business Communications Review (visited in Jul 2017).

[5] R. Wetzel, *CDN business models - the drama continues*, http://www.wetzelconsultingllc.com/BCR.CDNarticle2002.pdf, Apr. 2002, Business Communications Review (visited in Jul 2017).

[6] B. Frank, I. Poese, Y. Lin, G. Smaragdakis, A. Feldmann, B. M. Maggs, J. Rake, S. Uhlig, and R. Weber, *Pushing CDN-ISP Collaboration to the Limit*, *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 3, Jul. 2013.

[7] Named Data Networking, NDN Project, http://named-data.net (visited in Apr 2017).

[8] Richelle Adams, Active Queue Management: A Survey, IEEE Communications Surveys & Tutorials, vol. 3, n. 15, 2013.

[9] UMOBILE D4.1 Deliverable, Flowlet Congestion Control – Initial Report, Ioannis Psaras, et. al., EU UMOBILE project (grant No. 645124), Jul. 2016.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124

[10] UMOBILE D4.2 Deliverable, Flowlet Congestion Control – Final Report, Ioannis Psaras, et. al., EU UMOBILE project (grant No. 645124), (due on Jul 2017).

[11] R. Rosen, Linux containers and the future cloud, Linux Journal, vol. 2014, no. 240, Apr. 2014.

[12] Docker Inc., Docker, https://www.docker.com, (visited in Jan 2017).

[13] of NS-3 based Named Data Networking (NDN) simulator: ndnSIM documentation, http://ndnsim.net/2.3/index.html (visited in Jul 2017).

[14] Aref Meddeb. *Internet QoS: Pieces of the Puzzle*, IEEE Communications Magazine, 48(1), Jan. 2010

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 645124