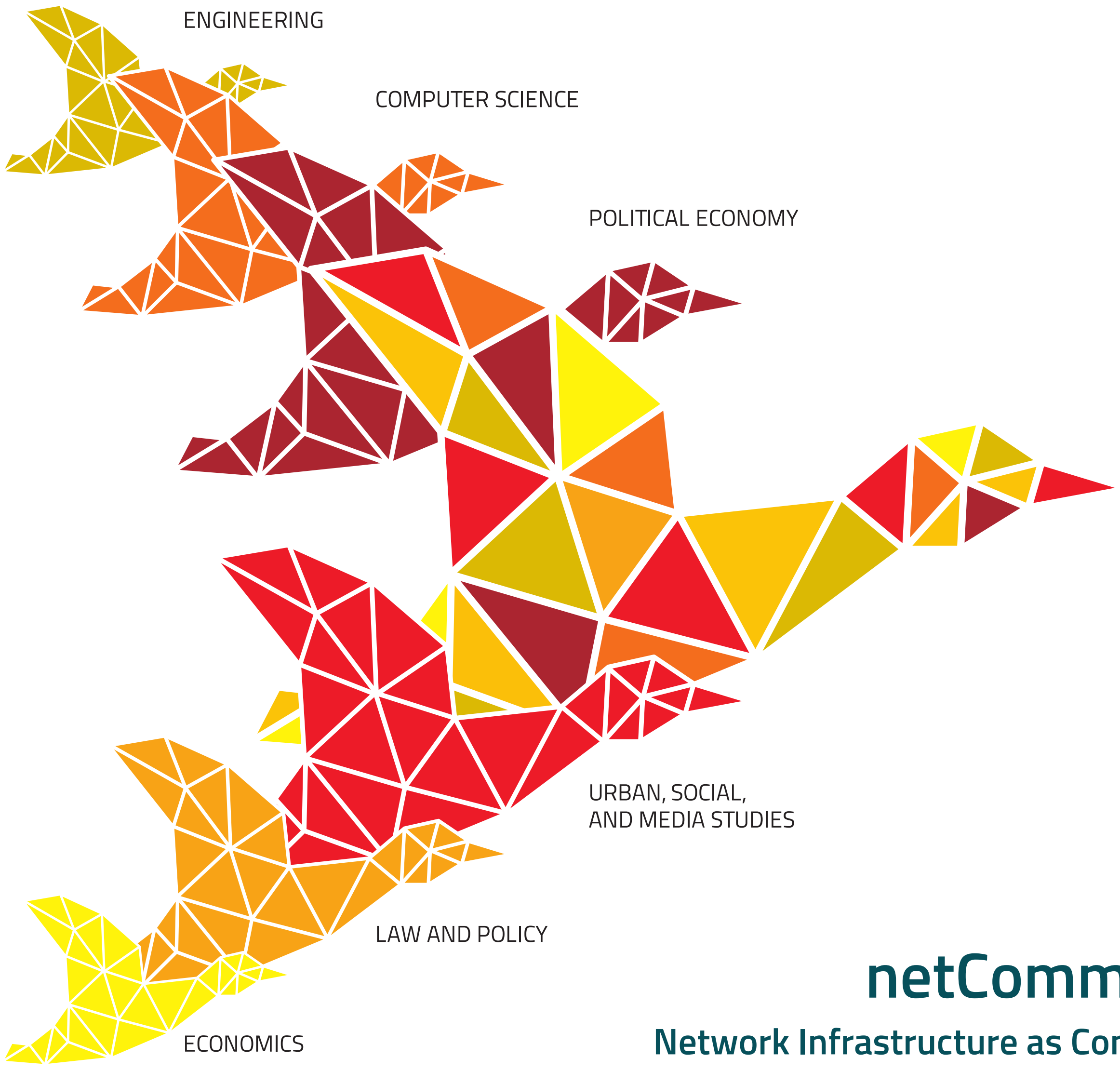


netCommons

Network Infrastructure as Commons

Release of New Open Source Software for All Applications

Deliverable Number D3.4
Version 1.0
March 2, 2018

Co-Funded by the Horizon 2020 programme of the European Union
Grant Number 688768

netCommons.eu

Project Acronym:	netCommons
Project Full Title:	Network Infrastructure as Commons.
Call:	H2020-ICT-2015
Topic:	ICT-10-2015
Type of Action:	RIA
Grant Number:	688768
Project URL:	http://netcommons.eu

Editor:	Leonardo Maccari, UniTN
Deliverable nature:	Report (R)
Dissemination level:	Public (PU)
Contractual Delivery Date:	December 31st, 2017
Actual Delivery Date	March 2, 2018
Number of pages:	78
Keywords:	community networks, applications, open source, participatory development, Cloudy, PeerStreamer, CommonTasker
Authors:	Leonardo Maccari, Luca Baldesi, Nicolò Facchi, Renato Lo Cigno UniTN Felix Freitag, Leandro Navarro, Roc Messeguer UPC Merkouris Karaliopoulos, Panagiota Micholia, Aris Pilichos, AUEB-RC
Peer review:	Leonardo Maccari, Felix Freitag, Merkouris Karaliopoulos, Renato Lo Cigno

History of Revisions

Rev.	Date	Author	Description
v0.1	12/6/2017	Leonardo Maccari	First draft
v0.2	15/6/2017	Felix Freitag	Contributions about progress in Cloudy
v0.3	15/6/2017	Leonardo Maccari	First draft with T3.3 progress
v0.4	30/6/2017	M. Karaliopoulos, P. Micholia, A. Pilichos	First draft with T3.4 progress
v0.5	30/6/2017	Renato Lo Cigno	Revision and editing for web-site publication
v0.6	22/11/2017	M. Karaliopoulos	Revision to accommodate the progress in the development of the CommonTasker app
v0.7	5/12/2017	Felix Freitag, Leandro Navarro	Adds Cloudy advances, new applications in Cloudy, Internet proxies and gateways
v0.8	20/12/2017	Leonardo Maccari	Revision of Cloudy plus PS completion
v0.9	23/12/2017	Felix Freitag	Revision of Cloudy
v0.91	16/1/2018	M. Karaliopoulos	CommonTasker completion
v1.0	28/2/2018	R. Lo Cigno	final proofreading

Executive summary

WP3 is dedicated to open-source applications for CNs: Tasks T3.2, T3.3, and T3.4 are the three tasks, where we develop open source code that can be used by CNs. The three tasks delivered the first version of the software at M12 with D3.2. In the period M12–M24 they continued the development of the software, but also advanced the research studies that are needed to support new applications and innovative solutions for CNs. This deliverable is structured in four chapters. After a brief introduction, the progress in each of the three tasks are reported:

Chapter 2 describes the work in T3.2. In T3.2, we are working on a CN-tailored linux distribution named Cloudy. This is a key theme for CNs, which are generally perceived as a gateway to the Internet, while they can be the driver for the evolution of new local applications. Cloudy makes it possible to deploy services using Docker in a way that is extremely simple for the user. Technically, it leverages the Serf gossiping protocol to be able to notify Cloudy nodes of newly deployed applications. This deliverable reports on the introduction of Docker-compose functionality in the Cloudy architecture, which enables developers to propose complex services that require more than one docker containers for higher reliability and security.

Furthermore, Chapter 2 illustrates the advances made by UPC in the research related to service and gateway placement in CNs, which will guide the future work on evolution and optimization of the Cloudy service.

Chapter 3 describes the work in T3.3. In T3.3, we focus on the development of the PeerStreamer (PS) application. PS is an open source Peer-to-Peer (P2P) live video streaming application, unique in its kind in the open source arena. PS is developed by UniTN and it is integrated with the new, docker-based Cloudy platform. This integration has required substantial work since the application had to be re-engineered to become a web-based application. PS is now a full-fledged live streaming solution that leverages the Cloudy platform to provide a distributed live video streaming service for CNs.

Chapter 3 also reports on the theoretical advances that UniTn made in exploiting the distributed nature of CNs to optimize the streaming function.

Chapter 4 describes the work in T3.4. In T3.4, we develop the CommonTasker Android application. The development of CommonTasker has been initiated as a brand new project by the AUEB team for the community of Sarantaporo.gr and progresses as a highly participatory design process, as reflected in [1] and building on the methodology described in D3.3 [2]. The mobile app is being continuously tailored to the needs of the community based on continuous interactions involving AUEB, Nethood, and local community members. CommonTasker is now tailored to the collection of data from the local farmers on their every day activities in their farms and aims to become their de facto smart assistant and online calendar. At the same time, these data can become the missing cornerstone for the provision of smart farming services to the community by external stakeholders/commercial service providers over the CN. This way, the app turns the CN into a key asset for both the community and external stakeholders, scaling up the importance of the CN for the local economy and paving the path for long-term synergies that can foster the sustainability of the network.

Contents

List of Figures

List of Tables

List of Acronyms

BASP	Bandwidth and Availability-aware Service Placement
CN	Community Network
DoA	Description of Activity
DTLS	Datagram Transport Layer Security
GUI	Graphical User Interface
HLS	HTTP Live Streaming
ICE	Interactive Connectivity Establishment
IoT	Internet of Things
ISMV	IIS Smooth Streaming Video
ISP	Internet Service Provider
IPFS	InterPlanetary File System
JSON	JavaScript Object Notation
NAT	Network Address Translation
MPEG	Motion Picture Expert Group
MP4	MPEG-4
NPAPI	Netscape Plugin Application Programming Interface
PS	PeerStreamer
ReST	Representational State Transfer
RTP	Real Time Protocol
SBC	Single-board Computer
SME	Small and medium-sized enterprise
TCP	Transmission Control Protocol
P2P	Peer-to-Peer
SDP	Session Description Protocol
RTP	Real-time Transport Protocol

1 Introduction

This report is the follow-up of D3.2 in which we documented the state of the software development carried on in netCommons at the end of M12 [3]. In that report we introduced the Cloudy, PeerStreamer and the CommonTasker applications, that are the open source applications being developed in netCommons. While the first two were started in previous EU-financed projects (the CLOMMUNITY and NAPA-WINE FP7 projects) the latter is a new application developed from scratch in netCommons. This deliverable reports on the development of these platforms in the second year of the project, in which we improved, stabilized and enhanced them. In the third year of the project we will test them with the help of some selected communities, and continue their development with the help of the methodology described in T3.1. As such, this deliverable describes the current state of the software, which is mostly complete, but needs further assessment from the communities in order to be improved and become fully usable.

As for [3], the actual deliverable is the software itself, its documentation, manuals, etc. that are referenced in the proper chapters of this document, which is indeed only a summary of the strategies adopted and the results achieved.

1.1 WP3 in the context of netCommons

In the vision of netCommons, CNs are not just a community-driven replacement of last mile access to the Internet. CNs are also the playground to experiment with new applications that can be developed, deployed and used by the local community. This is a key step in the direction of a new model of computation, opposite to the centralization of applications and data on remote, commercial, cloud-based services.

Ideally, local applications could replace cloud-based ones in order to give back to the people the control on their data, to exploit the performances of a CN as a local network (which in certain situations can be largely better than the performance of the Internet uplink), and to foster a local economy of developers and maintainers instead of relying on external services. In practice, it is extremely hard for local applications to compete with commercial, cloud-based ones, which are widely used by billions of people and are highly engineered and optimized to provide a smooth experience to the user. Local applications can leverage various factors, such as being tailored for local needs, and having a potential set of users interested not only in the application *per se* but also to its value for the community.

Therefore, it is of paramount importance to propose new applications to the communities, applications that can be the base for local modifications to make them fulfill a specific need of the community. T3.2, T3.3, and T3.4, in tight cooperation with T3.1 perform this activity. They produce open source software that can be proposed to the communities and that can be the ground for further customizations and specific adoptions by the local CN.

In this sense, the three tasks start from a different position. While Cloudy is a generic platform envisioned from the beginning as an enabler for any application that can run on Docker, PeerStreamer and CommonTasker are applications that have a specific use. PeerStreamer rides the current wave of interest for live streaming (Cisco estimation say that live video will grow 15-fold between 2016-2021[1]) while CommonTasker is developed from scratch to match the interest of a rural community interested in precision agriculture.

Also, PeerStreamer is designed as a generic service that could be integrated in different applications, while

[1]See https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html

CommonTasker is a more "social" application assuming interactions, feedback, and motivations for its users to participate in a collaborative process.

This heterogeneity gives us the chance to experiment with different approaches, as we believe that the mix of technical skills and political motivations that the participants of a CN generally own gives to researchers an excellent opportunity to test the creation of new applications.

1.2 Current State and Future Steps

All the applications mentioned so far are extensively described in the next chapters, with details on their improvement from their state at M12. All the released code uses an open source license, the decision on which specific license to use is left to the implementer, as it depends on a number of external factors, such as the dependency on other software. All the open source code can be found in the Github repository of the project[2]. This deliverable refers to the code tagged with the tag `D3.4-final` on each of the github repository. As mentioned, this deliverable is just a companion document for the release of the open source code of WP3, it is not intended to be a user manual or a specific technical reference for the released code, it overviews its functions and improvements. For further details, please see the documentation on each repository.

In the third year of the project, we will test these applications together with some CNs we are in contact with, namely Guifi, ninux and Sarantaporo.gr. In this last year we will gather feedback and eventually plan new developments on the applications based on the participatory involvement of the communities, as explained in D3.3. Reports on this activity will be included both in D3.6 (M31) and D3.5 (M36) for the parts pertaining the participatory methodology used and the software modifications respectively.

[2]See https://github.com/netcommonseu.

2 Update on the Cloudy Distribution

With reference to T3.2, the Description of Activity (DoA) states "*T3.2 will research on Community Clouds, the application of distributed cloud systems to CNs.*" The work combines the tasks of evaluating the community cloud system, in order to assess its position and needs, and the developments of components to enable new possibilities with the platform.

The starting point of year 2 is clearly D3.2 [3], where the initial support of Docker-provisioned services in Cloudy was presented. Experiences with the Cloudy operation and the research on services helped gaining insight and take decisions on developments and orientations.

The code is publicly available in https://github.com/Clommunity and https://github.com/netCommonsEU/cDistro, and the development and research activity presented in this chapter lead also to the publication of seven scientific papers listed in Sec. 2.8 and further documented in D6.2. These works can be grouped into extending the service provision capabilities of Cloudy through the development of enhanced support services, and exploring different aspects for building community clouds. The following sections describe in more detail the work and results obtained.

2.1 The proposition of edge microclouds

The Fog [4][5] and Edge Computing [6] approaches extend the classic cloud architecture, which considers cloud computing services to run in large data centers. In edge cloud computing, by placing computing devices at the network edge, an additional infrastructure layer is added to the system model. As part of a cloud service, specific functions are performed on these edge devices leveraging their closeness to the users or the data source.

Different to the above described approach, the *edge microclouds* which we propose leverage the resources of other edge devices, and therefore can inherit the characteristics of cloud computing, like elastic resource and service availability, while these microclouds are still located at the network edge.

Microclouds therefore differ from the previously described edge cloud architectures regarding the pool of resources and services that are used. While, from a traditional cloud service provider perspective, edge computing infrastructure is mainly a resource that extends the capabilities of a data center based cloud service, in microclouds, the cloud resource pool for service provision are mainly the spare resources of distributed edge devices (even if it can extend towards data center cloud services to improve the edge service performance, e.g. for improved resilience). The collaborative element between users consists in resource donations, willingness to host distributed services, and active provision of services with the community of users in the microcloud.

Fig. 2.1 illustrates the microcloud concept. In terms of hardware, we consider diverse types of computing devices which are found at the network edge, ranging from mid-capacity servers at the high end to IoT boards at the low end. These devices, by geographic location or community-tied operations are envisioned to form a distributed infrastructure, which provides some resource elasticity to edge services. In the picture four types of services are indicated: While the network service relates to the management of the networking infrastructure, the other three types of services (personal, community and enterprise) indicate services belonging to different stakeholders which use the microcloud system.

To built these collaborative edge microclouds, communication and computing devices at the network edge are leveraged. Different hardware device classes available at the network edge can become a microcloud resource. Mid-capacity servers may be found at municipalities and local Small and medium-sized enterprises (SMEs). Low-capacity PCs or Single-board Computers (SBCs) are more and more used in domestic environments.

High-end home gateways and routers to connect to the ISP today have fairly large computing capabilities. Different from data center clouds, however, such edge microclouds are distributed and heterogeneous in hardware, software, operational policies and administrative domains.

The services in these microclouds need to target the requirements of the community of the users they are to serve, in order to encourage contribution and participation by the users. Services enabled by these collaborative microclouds should aim for new applications or customization, and exploit the element, which are complementary to traditional cloud services.

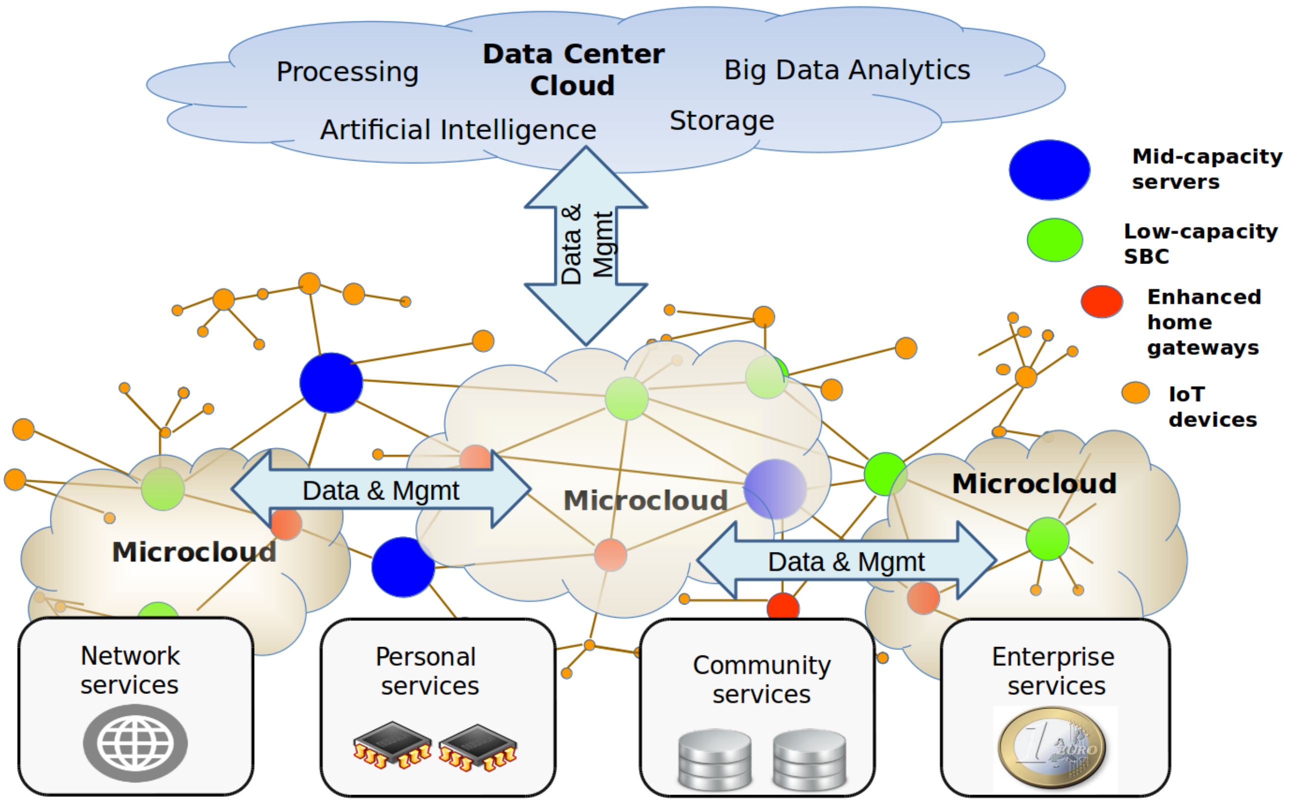


Figure 2.1: Microclouds at the network edge.

2.2 Docker compose-based service provision in microclouds

In the first year of netCommons, Cloudy was extended to deploy applications provided as single Docker containers. In order to use the microclouds for more complex applications, i.e., composed of a set of containers with dependencies among each other, the previous solution falls short in terms of providing an easily way for end users to accomplish such deployments without hassle. In order to supported such applications in the Cloudy platform, in Y2 of netCommons we have integrated *Docker-compose* in Cloudy as the tool for deploying such applications. The work described here lead to the publication of [7].

2.2.1 Software Integration

The usefulness of the services in microclouds is key for motivating users and encourage volunteer activity. While commercial players are increasing their service offer leveraging edge computing, microclouds need to differentiate by identifying services which benefit from the horizontal sharing of resources among users, a model not yet implemented by commercial providers.

Such promising services may include for instance machine learning for data analytics, which nowadays is mostly done at cloud data centers. The increase of Internet of Thingss (IoTs) data generation at the network edge, however, will create the need to perform local analytics at the edge. Microclouds may serve as infrastructures to extend the capabilities of single edge devices to conduct such tasks. Security and privacy requirements for IoT are additional use cases which can benefit from local data analytics support.

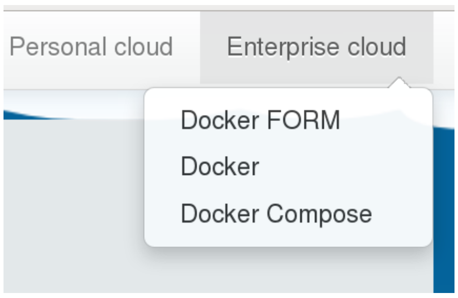
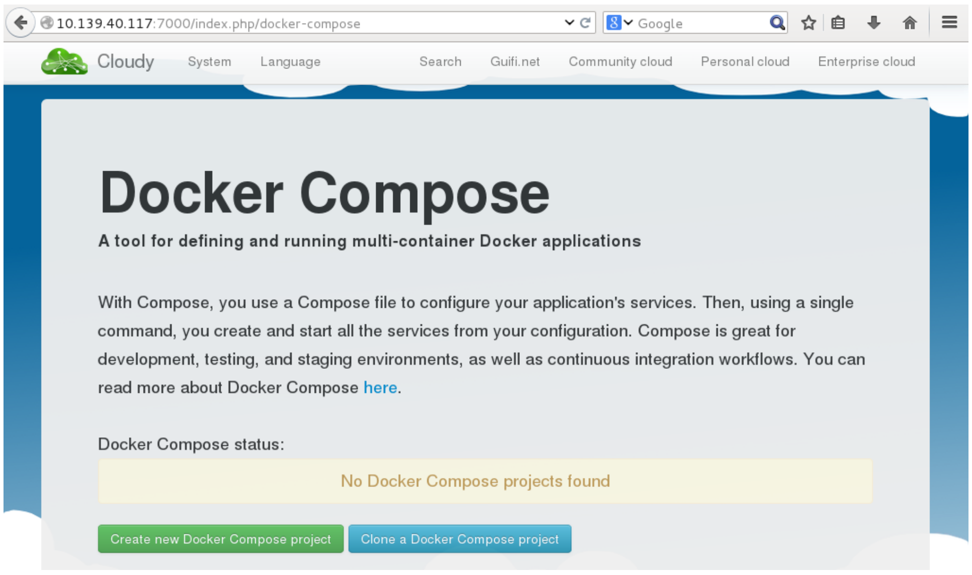


Figure 2.2: Docker-compose integration in the GUI.



Docker Compose

A tool for defining and running multi-container Docker applications

With Compose, you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration. Compose is great for development, testing, and staging environments, as well as continuous integration workflows. You can read more about Docker Compose here.

Docker Compose status:

No Docker Compose projects found

Create new Docker Compose project
Clone a Docker Compose project

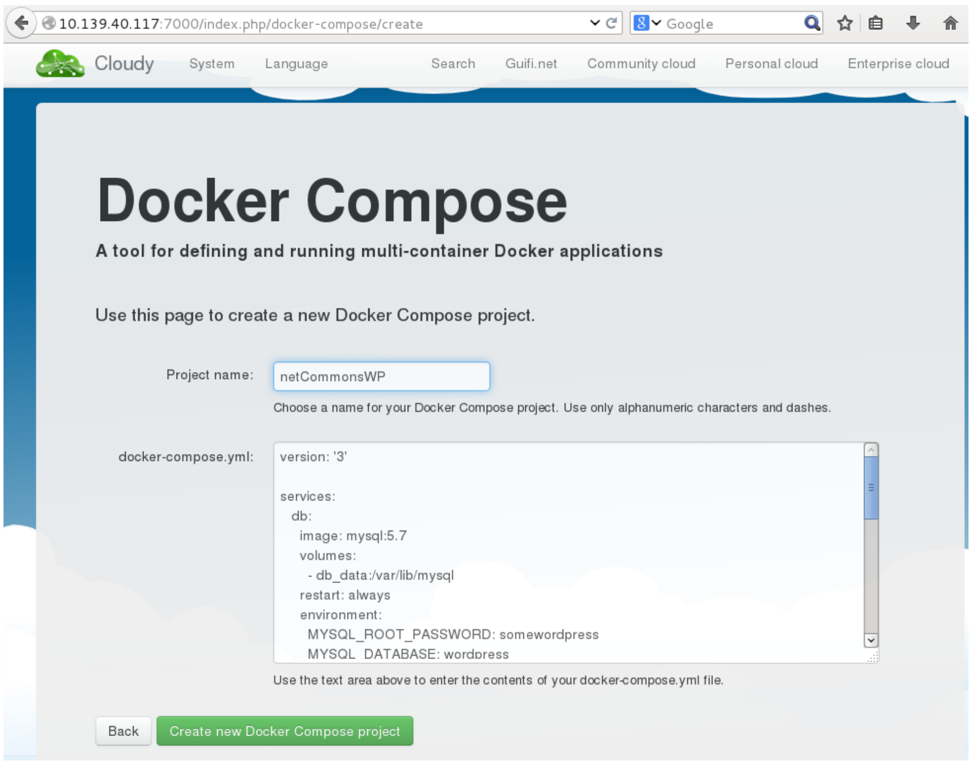
Figure 2.3: Docker-compose availability in Cloudy after clicking on the installation button.

Since the first year of netCommons, Cloudy manages Docker-based applications in the *Enterprise cloud* menu. The reason for this choice was the to consider a Cloudy node as an infrastructure for multi-tentant usage. These tenants were identified as the node owners, the community and commercial enterprises. Fig. 2.2 shows the extension of the *Enterprise cloud* menu with the integration of *Docker-compose* in the Web Graphical User Interface (GUI).

The Docker-compose software package is not installed in Cloudy by default, but the user at her choice can easily install it by means of the GUI without the need to open a terminal. After the installation, Docker-compose is available to be used. To carry out a deployment, the Docker-compose applications needs to be specified. Fig. 2.3 shows the options given to the user to introduce the Docker-compose description in the GUI.

Fig. 2.4 illustrates an example for an application described for Docker-compose. The case we present is the popular Wordpress application, which is often used for the hosting of Web sites and blogs.

Once the application is specified, it can be launched. Launching implies the download of the Docker images (if not available locally) and instantiating them as containers. Fig. 2.5 shows the management options of Docker-compose in the Web GUI, where the used Docker compose recipe can be seen.



Docker Compose

A tool for defining and running multi-container Docker applications

Use this page to create a new Docker Compose project.

Project name: netCommonsWP

Choose a name for your Docker Compose project. Use only alphanumeric characters and dashes.

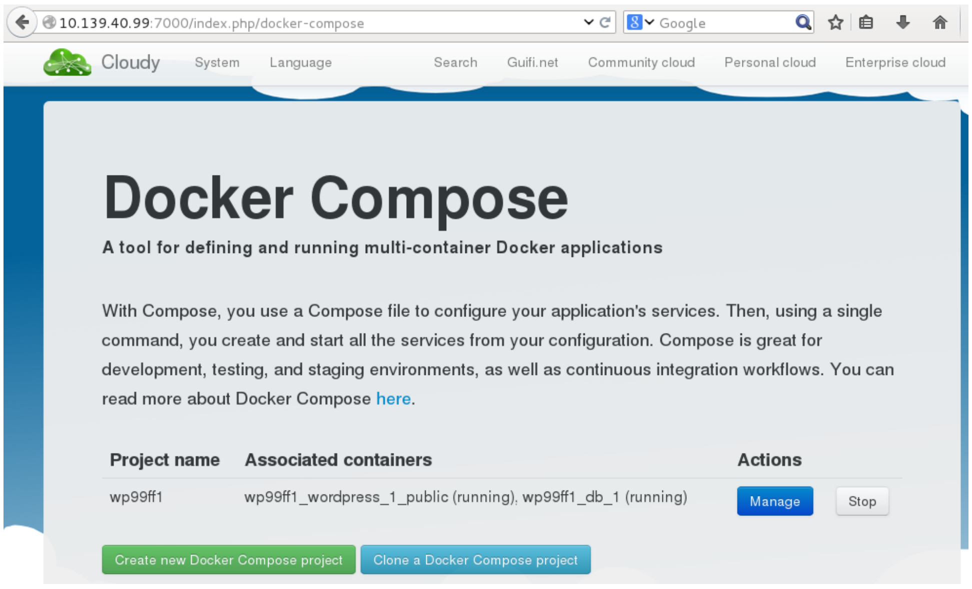
docker-compose.yml:

```
version: '3'

services:
  db:
    image: mysql:5.7
    volumes:
     - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
```

Use the text area above to enter the contents of your docker-compose.yml file.

Figure 2.4: The Wordpress application expressed as Docker-compose specification in Cloudy.


10.139.40.99:7000/index.php/docker-compose
Google
Cloudy
System
Language
Search
Guifi.net
Community cloud
Personal cloud
Enterprise cloud
Manage
Stop
Create new Docker Compose project
Clone a Docker Compose project

Docker Compose

A tool for defining and running multi-container Docker applications

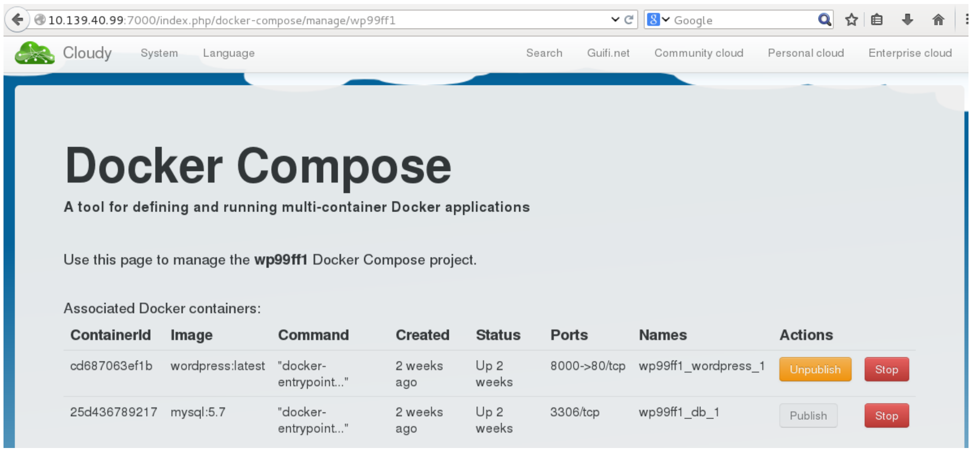
With Compose, you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration. Compose is great for development, testing, and staging environments, as well as continuous integration workflows. You can read more about Docker Compose here.

Project name	Associated containers	Actions
wp99ff1	wp99ff1_wordpress_1_public (running), wp99ff1_db_1 (running)	Manage Stop

Figure 2.5: The user can manage the Docker-compose-based application.

Cloudy users are given the option to share their deployed services with the community. Once a service or application is installed, the user has to explicitly chose to publish it. Fig. 2.6 shows for the presented case of the Wordpress application the options for the user to publish (or not) the deployed application.

Once services are published by a user, they are found by the *Search service* from any other Cloudy instance. Fig. 2.7 reports how several services at different Cloudy nodes are represented on the interface of a generic user, even if the node they reside on is not local.



Docker Compose

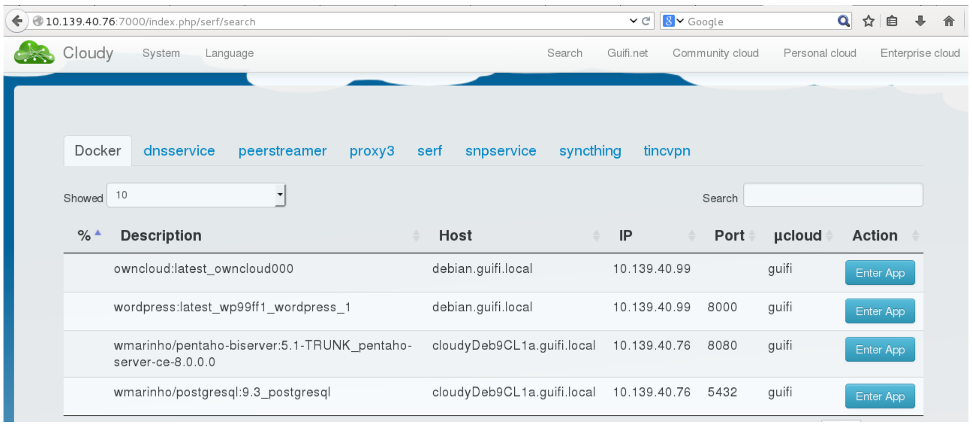
A tool for defining and running multi-container Docker applications

Use this page to manage the **wp99ff1** Docker Compose project.

Associated Docker containers:

ContainerId	Image	Command	Created	Status	Ports	Names	Actions	
cd687063ef1b	wordpress:latest	"docker-entrypoint..."	2 weeks ago	Up 2 weeks	8000->80/tcp	wp99ff1_wordpress_1	Unpublish	Stop
25d436789217	mysql:5.7	"docker-entrypoint..."	2 weeks ago	Up 2 weeks	3306/tcp	wp99ff1_db_1	Publish	Stop

Figure 2.6: The Cloudy user has an option to announce a deployed service to the community.



%	Description	Host	IP	Port	µcloud	Action
	owncloud:latest_owncloud000	debian.guifi.local	10.139.40.99		guifi	Enter App
	wordpress:latest_wp99ff1_wordpress_1	debian.guifi.local	10.139.40.99	8000	guifi	Enter App
	wmarinho/pentaho-biserver:5.1-TRUNK_pentaho-server-ce-8.0.0.0	cloudyDeb9CL1a.guifi.local	10.139.40.76	8080	guifi	Enter App
	wmarinho/postgresql:9.3_postgresql	cloudyDeb9CL1a.guifi.local	10.139.40.76	5432	guifi	Enter App

Figure 2.7: The published services in the Cloudy microcloud are found by another user.

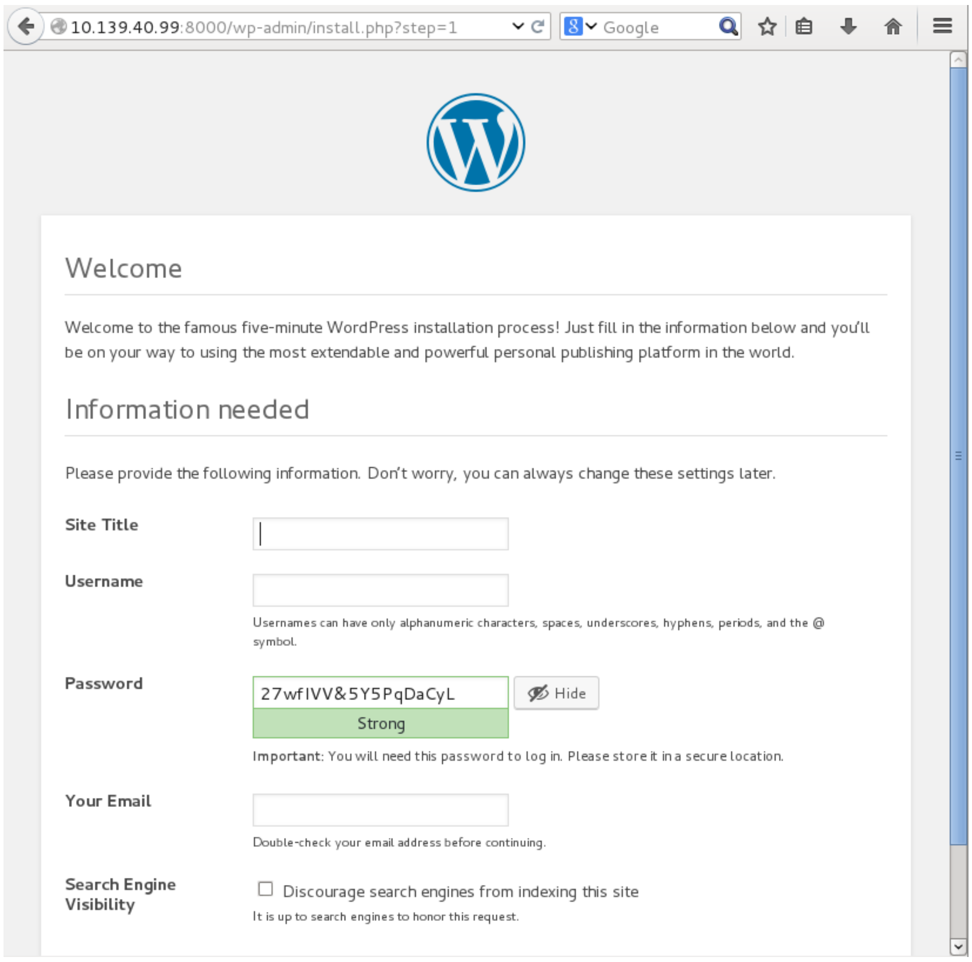
If the service is a Web application, it can directly be accessed through the search service graphical interface. Fig. 2.8 shows how another Cloudy users can access the Web GUI Wordpress application.

2.2.2 Services from the IoT Domain and practical experimentation

Huge business potentials drive important use cases in ambient intelligence and the IoT for edge computing. Most of today's edge computing services, however, are build from platforms which consist of proprietary gateways, which are either closed or fairly restricted to deploy any third-party services. This lack of openness may indeed be an opportunity for Cloudy, which, in contrast to those restricted extension possibilities, offers a participatory edge computing system running on home gateways to serve as an open environment to deploy local services. We have singled out the IoT domain for experimentation to understand better the practical possibilities and potential.

We are interested in evaluating the Cloudy use case for a large commercial IoT services provider that offers software solution for IoT, distributing it both as a commercial enterprise edition and an open-source community edition, in order for us to test some prototype components.

For this purpose, we have chosen Hitachi Vantara's Pentaho open-source business intelligence (BI) suite [8], that provides comprehensive reporting, data processing, data integration, and data mining features. The Pentaho Data Integration (PDI) client is a desktop application that users can use to process and integrate data on their own machines, and design data workflows that they can upload to the repository in the Pentaho Server. Users


10.139.40.99:8000/wp-admin/install.php?step=1

Welcome

Welcome to the famous five-minute WordPress installation process! Just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world.

Information needed

Please provide the following information. Don't worry, you can always change these settings later.

Site Title

Username

Usernames can have only alphanumeric characters, spaces, underscores, hyphens, periods, and the @ symbol.

Password: 27wfIVV&5Y5PqDaCyL — Hide — Strong

Important: You will need this password to log in. Please store it in a secure location.

Your Email

Double-check your email address before continuing.

Search Engine Visibility: ☐ Discourage search engines from indexing this site

It is up to search engines to honor this request.

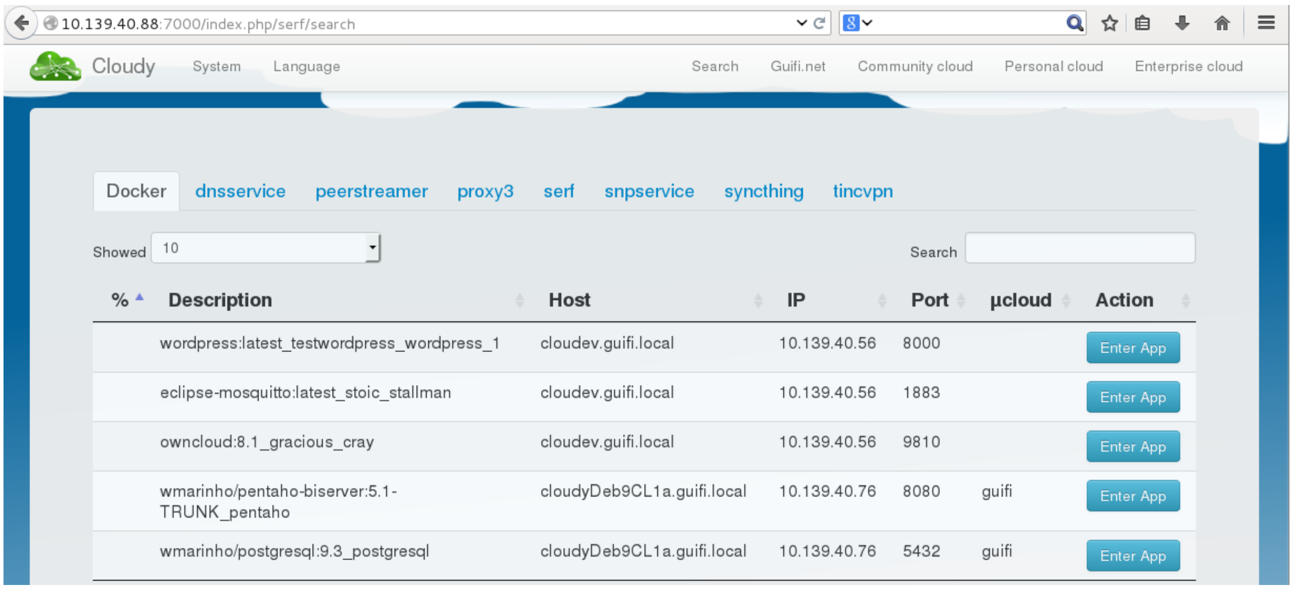
Figure 2.8: The published Wordpress application is accessed by a Cloudy user.

can schedule jobs to run data workflows at regular intervals, and perform other advanced operations. The Pentaho Platform also provides web-based tools to create reports, dashboard and charts in a browser. Pentaho also includes Weka, a popular open-source data mining software, and also integrates support for executing machine learning programs through its integration of Python and R. Pentaho provides an extensible plugin architecture, so users can integrate support for their special use cases. For these services from Pentaho platform, there is also the option to install them as Docker containers, as we explain in the following.

In the setup using Cloudy in Guifi.net, our goal is to deploy the Pentaho Server as a Docker container on the nodes in the edge community cloud, which are running Cloudy software.

In the setup for the practical experimentation, we consider the scenario where each owner of the node sets up an independent Pentaho Server instance, and the other users of the edge community microcloud will contact the owner to obtain login credentials for that instance of Pentaho Server. For data collection and storage, the user can have a virtual machine with a public cloud provider on the Internet, or set up another node in the edge community cloud as an FTP or a database server, where the nodes will upload the data received from IoT devices. In the setup, we have used PostgreSQL database that also runs as a Docker container on a Cloudy device in parallel to other applications.

After the installation of Pentaho services in Cloudy, we can observe in Fig. 2.9 from the perspective of a user connected to the Cloudy device with IP address 10.139.40.88 that she can find a Docker-based Pentaho service available in the community cloud. The Pentaho service is available from a provider with IP address 10.139.40.76, while there are other services offered by different providers too.



10.139.40.88:7000/index.php/serf/search

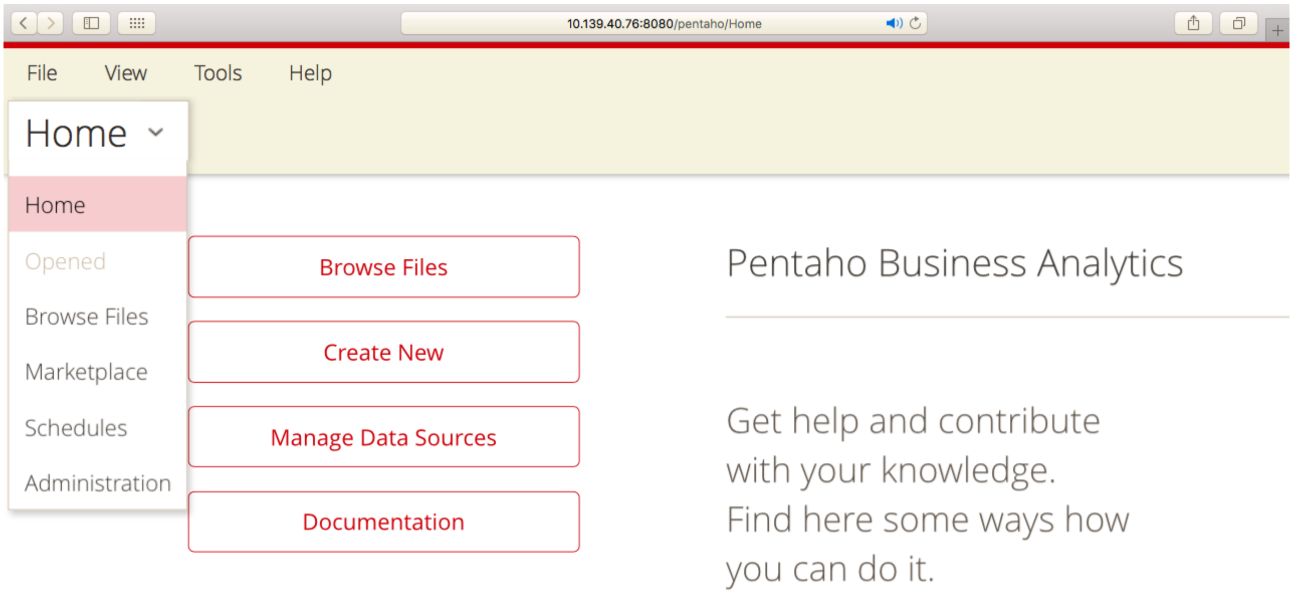
Cloudy | System | Language | Search | Guifi.net | Community cloud | Personal cloud | Enterprise cloud

Docker | dnsservice | peerstreamer | proxy3 | serf | snpservice | syncthing | tincvpn

Showed 10 Search

%	Description	Host	IP	Port	μcloud	Action
	wordpress:latest_testwordpress_wordpress_1	cloudev.guifi.local	10.139.40.56	8000		Enter App
	eclipse-mosquitto:latest_stoic_stallman	cloudev.guifi.local	10.139.40.56	1883		Enter App
	owncloud:8.1_gracious_cray	cloudev.guifi.local	10.139.40.56	9810		Enter App
	wmarinho/pentaho-biserver:5.1-TRUNK_pentaho	cloudyDeb9CL1a.guifi.local	10.139.40.76	8080	guifi	Enter App
	wmarinho/postgresql:9.3_postgresql	cloudyDeb9CL1a.guifi.local	10.139.40.76	5432	guifi	Enter App

Figure 2.9: Pentaho service installations in Cloudy.



10.139.40.76:8080/pentaho/Home

Get help and contribute with your knowledge. Find here some ways how you can do it.

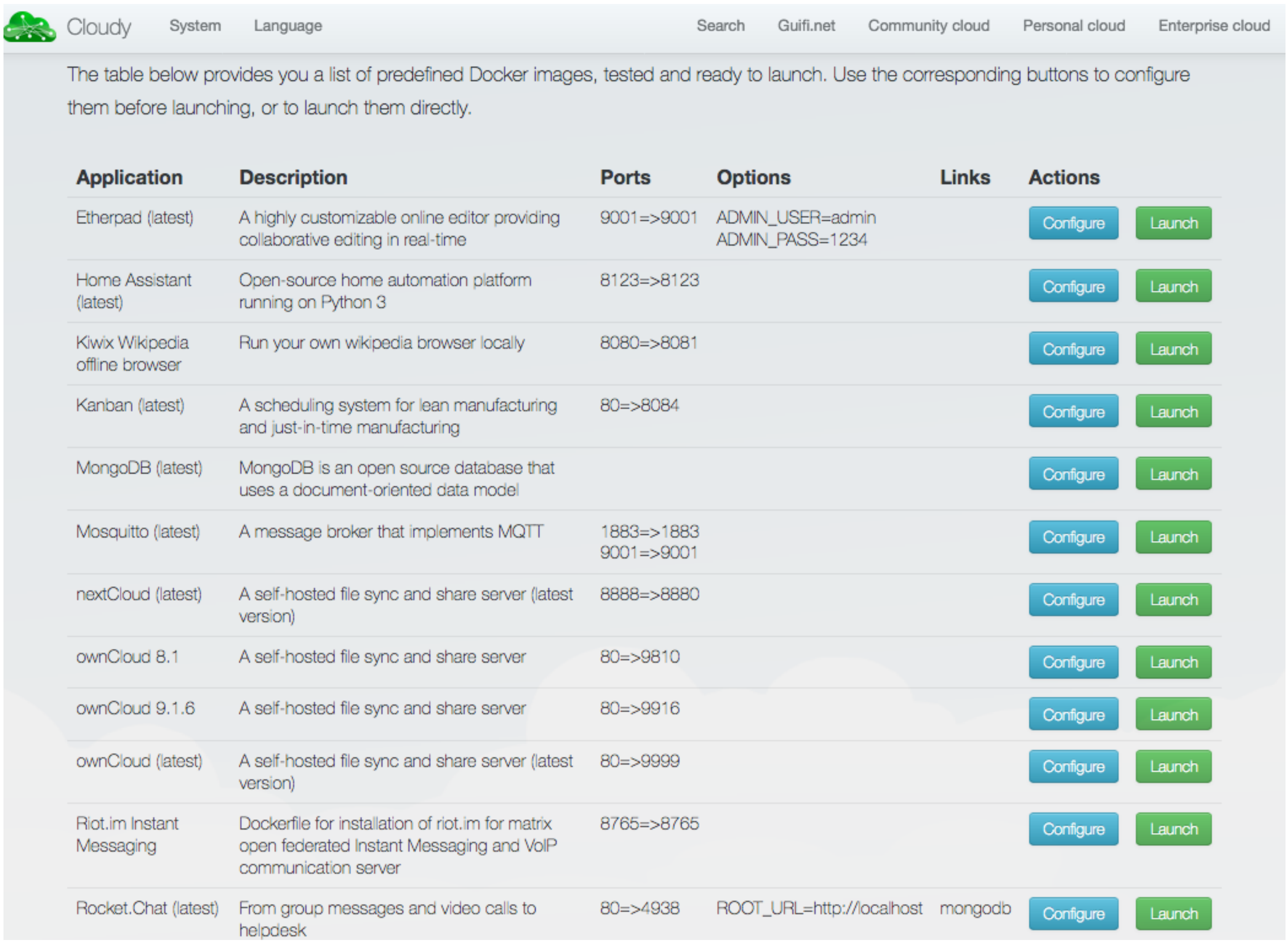
Figure 2.10: Access to the Pentaho service installed by the user.

The Pentaho service is discoverable in Cloudy because the provider has tagged it as a *shared* service. Therefore anyone in Guifi.net running Cloudy software will be able to find it.

Fig. 2.10 shows the home screen of a Pentaho service, where the user can manage stored files and scheduled jobs, after logging in by accessing it through Cloudy web interface.

2.3 New applications enabled by Docker

The addition of Docker support, the ability to download images, run containers, and compose them, brings to Cloudy a wide range of applications. The current list of predefined Docker images, tested and ready to launch are listed in Fig. 2.11.



The table below provides you a list of predefined Docker images, tested and ready to launch. Use the corresponding buttons to configure them before launching, or to launch them directly.

Application	Description	Ports	Options	Links	Actions
Etherpad (latest)	A highly customizable online editor providing collaborative editing in real-time	9001=>9001	ADMIN_USER=admin ADMIN_PASS=1234		Configure Launch
Home Assistant (latest)	Open-source home automation platform running on Python 3	8123=>8123			Configure Launch
Kiwix Wikipedia offline browser	Run your own wikipedia browser locally	8080=>8081			Configure Launch
Kanban (latest)	A scheduling system for lean manufacturing and just-in-time manufacturing	80=>8084			Configure Launch
MongoDB (latest)	MongoDB is an open source database that uses a document-oriented data model				Configure Launch
Mosquitto (latest)	A message broker that implements MQTT	1883=>1883 9001=>9001			Configure Launch
nextCloud (latest)	A self-hosted file sync and share server (latest version)	8888=>8880			Configure Launch
ownCloud 8.1	A self-hosted file sync and share server	80=>9810			Configure Launch
ownCloud 9.1.6	A self-hosted file sync and share server	80=>9916			Configure Launch
ownCloud (latest)	A self-hosted file sync and share server (latest version)	80=>9999			Configure Launch
Riot.im Instant Messaging	Dockerfile for installation of riot.im for matrix open federated Instant Messaging and VoIP communication server	8765=>8765			Configure Launch
Rocket.Chat (latest)	From group messages and video calls to helpdesk	80=>4938	ROOT_URL=http://localhost	mongodb	Configure Launch

Figure 2.11: Predefined templates for launching Docker containers.

In comparison with the version of Cloudy delivered in D3.2 [3], due to the extension for Docker-compose, we needed to re-design the internal representation for the service descriptions. The list of predefined services has increased, which includes now for instance the HomeAssistant, Mosquitto, nextCloud, while for other applications the description of the updated Docker images was integrated.

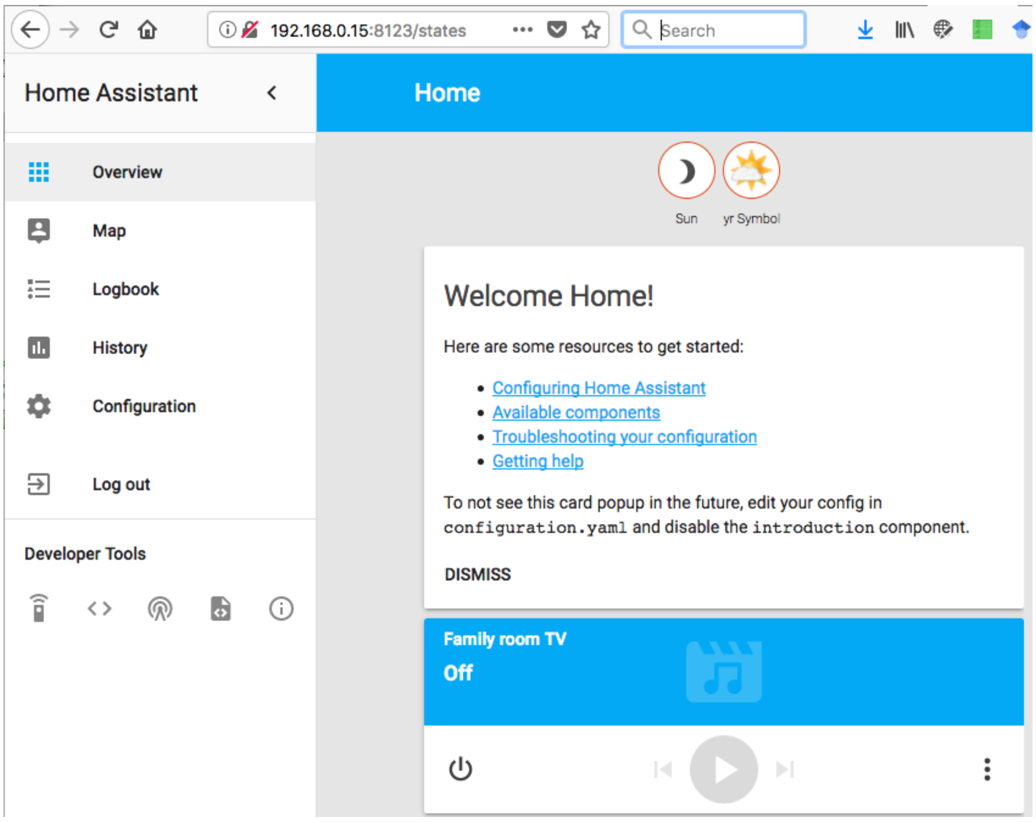
The following subsections describe some of these applications that illustrate the diversity of added value brought by the Docker integration in Cloudy.

2.3.1 Home Assistant

Home Assistant is an open-source home automation platform running on Python 3. It allows to track and control all devices at home and automate control. That includes media, security, environmental sensors, lighting, and any other device with a wired or wireless control interface. The introductory web is shown in Fig. 2.12, a web page to send audio messages to home devices in Fig. 2.13, and the home page with a control element for a media device (Chromecast) in Fig. 2.14.

2.3.2 OwnCloud and NextCloud

Nextcloud is a suite of client-server software for creating and using file hosting services. It is functionally similar to Dropbox, although Nextcloud is free and open-source, allowing anyone to install and operate it on a private server. In fact Cloudy supports two variants: ownCloud and NextCloud, as shown in Fig. 2.15. The



Welcome Home!

Here are some resources to get started:

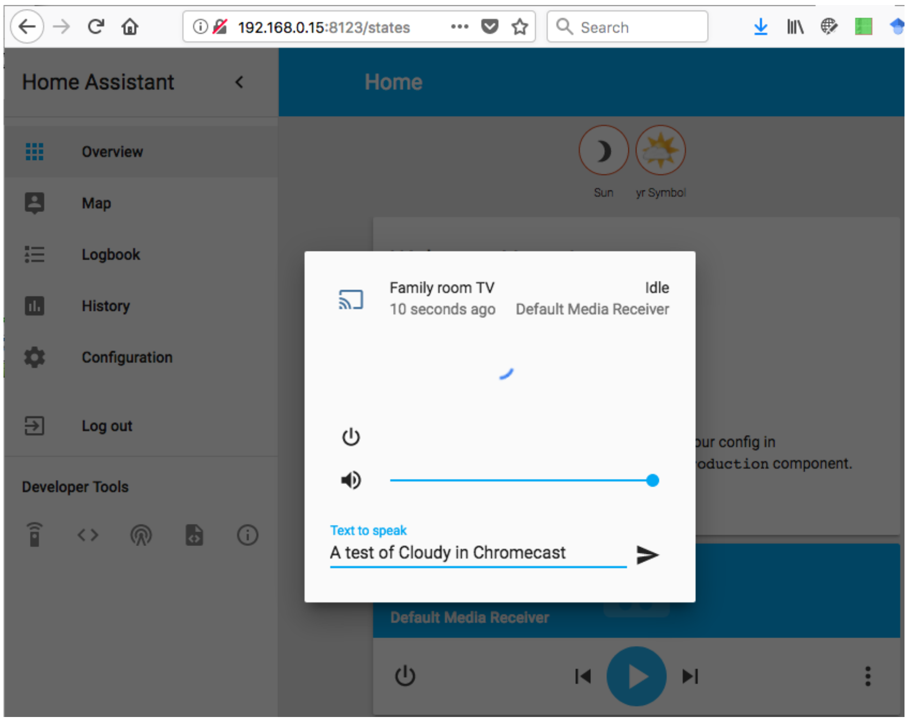
- Configuring Home Assistant
- Available components
- Troubleshooting your configuration
- Getting help

To not see this card popup in the future, edit your config in `configuration.yaml` and disable the `introduction` component.

DISMISS

Family room TV
Off

Figure 2.12: Home Assistant: Welcome screen.



Family room TV — Idle
10 seconds ago — Default Media Receiver

Text to speak
A test of Cloudy in Chromecast

Figure 2.13: Home Assistant: application to send voice commands to home devices.

advantage of both is that the user can combine this service running on a Cloudy device with a client application, as shown in Fig. 2.16, running on a mobile or desktop device to access the file repository in Fig. 2.17, that can be

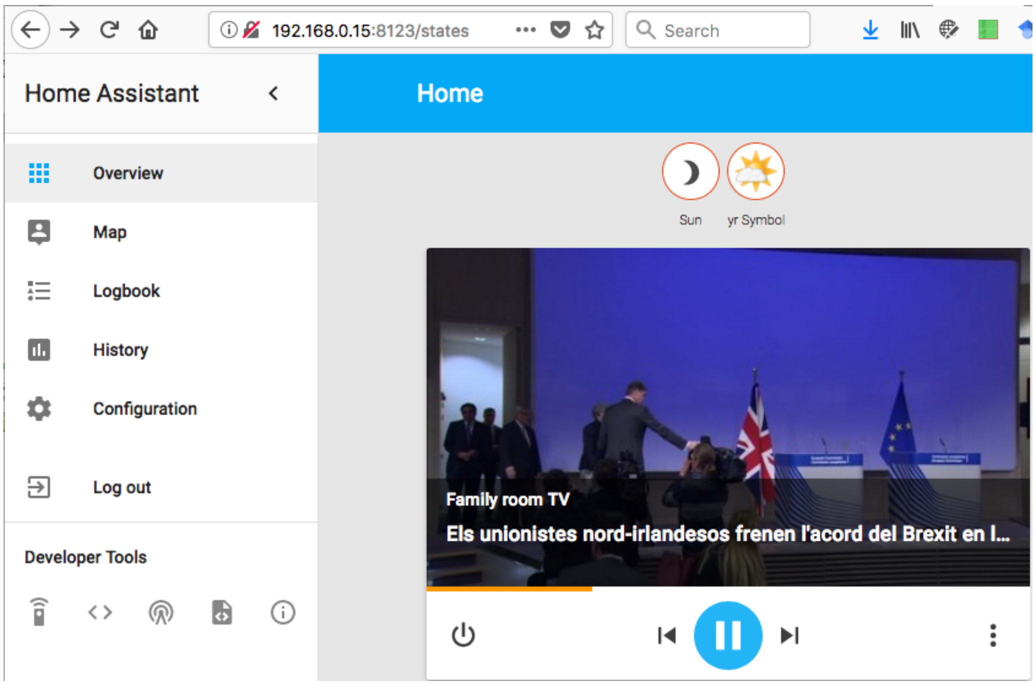


Figure 2.14: Home page with a control element for a media device (Chromecast).

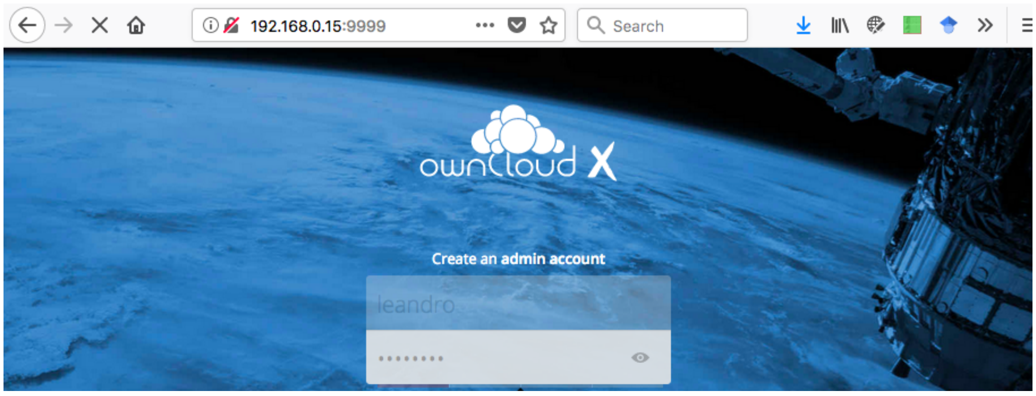
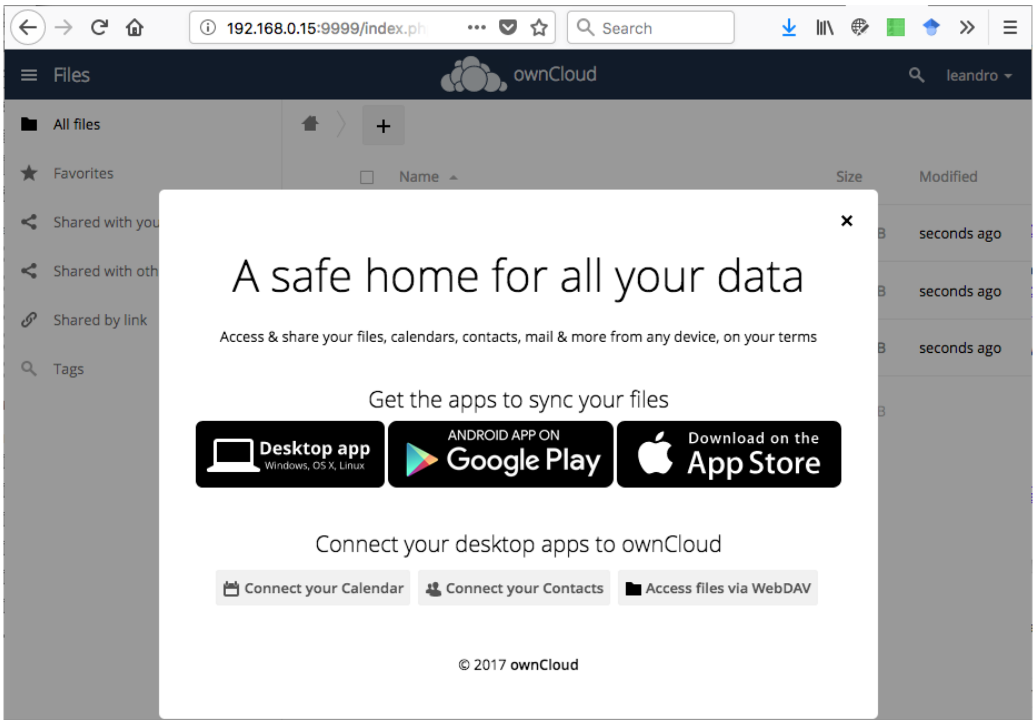


Figure 2.15: OwnCloud: login page, a Docker container running in a Cloudy device.

shared with multiple and external users. Both have a marketplace (see Fig. 2.18 and installable add-ons which include complex applications like videoconferencing in Fig. 2.19, a group calendar in Fig. 2.20 document editing (the LibreOffice suite) and visualization (PDF viewer) in Fig. 2.21.

2.3.3 Etherpad

Etherpad is an open source, web-based collaborative real-time editor, allowing authors to simultaneously edit a text document, and see all of the participants' edits in real-time, with the ability to display each author's text in their own color. A sample document being edited concurrently by two users running on a Cloudy instance is listed in Fig. 2.22.



A safe home for all your data

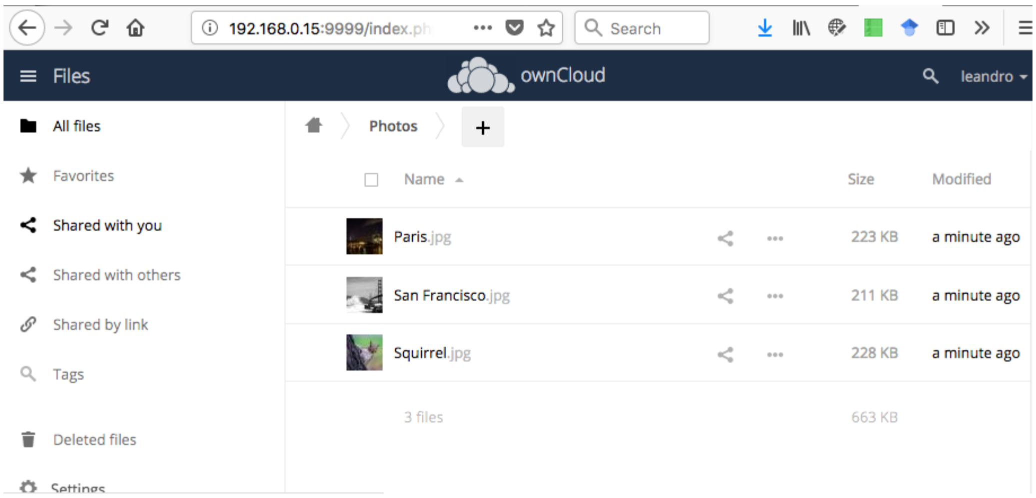
Access & share your files, calendars, contacts, mail & more from any device, on your terms

Get the apps to sync your files

Connect your desktop apps to ownCloud

© 2017 ownCloud

Figure 2.16: OwnCloud: reminder of the client applications to this service.



Name	Size	Modified
Paris.jpg	223 KB	a minute ago
San Francisco.jpg	211 KB	a minute ago
Squirrel.jpg	228 KB	a minute ago
3 files	663 KB	

Figure 2.17: OwnCloud: the file browser in the Cloudy device.

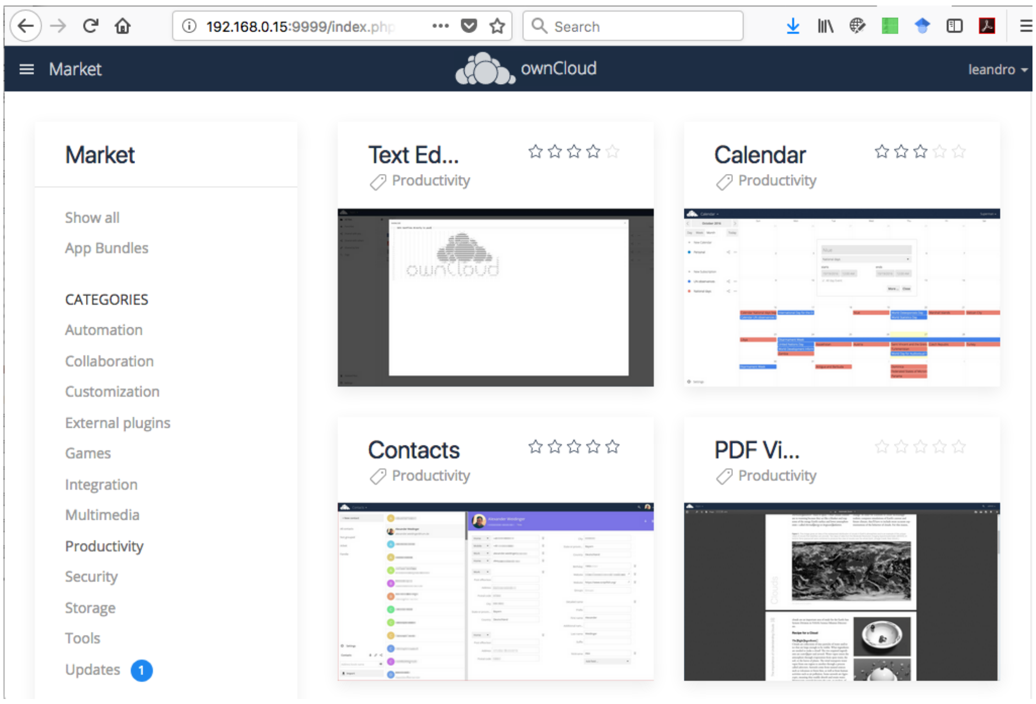


Figure 2.18: OwnCloud: the marketplace of installable applications.

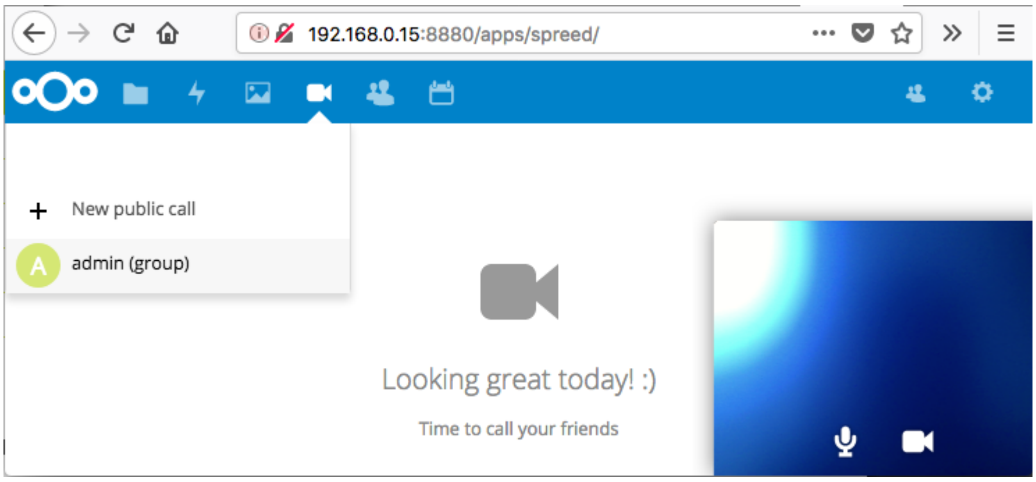


Figure 2.19: NextCloud: videoconferencing application.

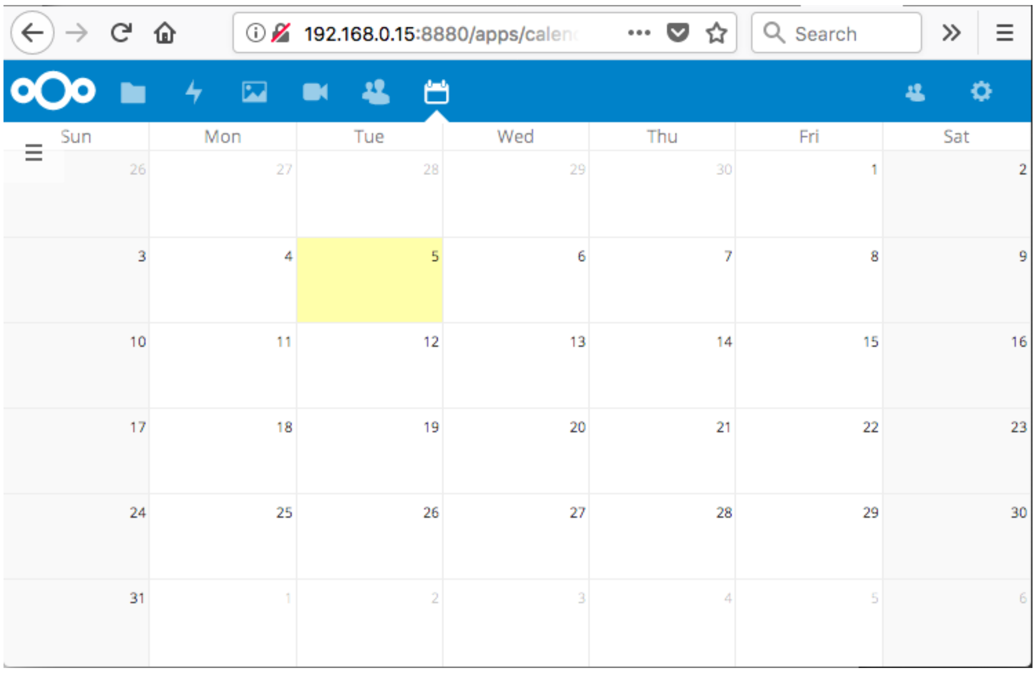
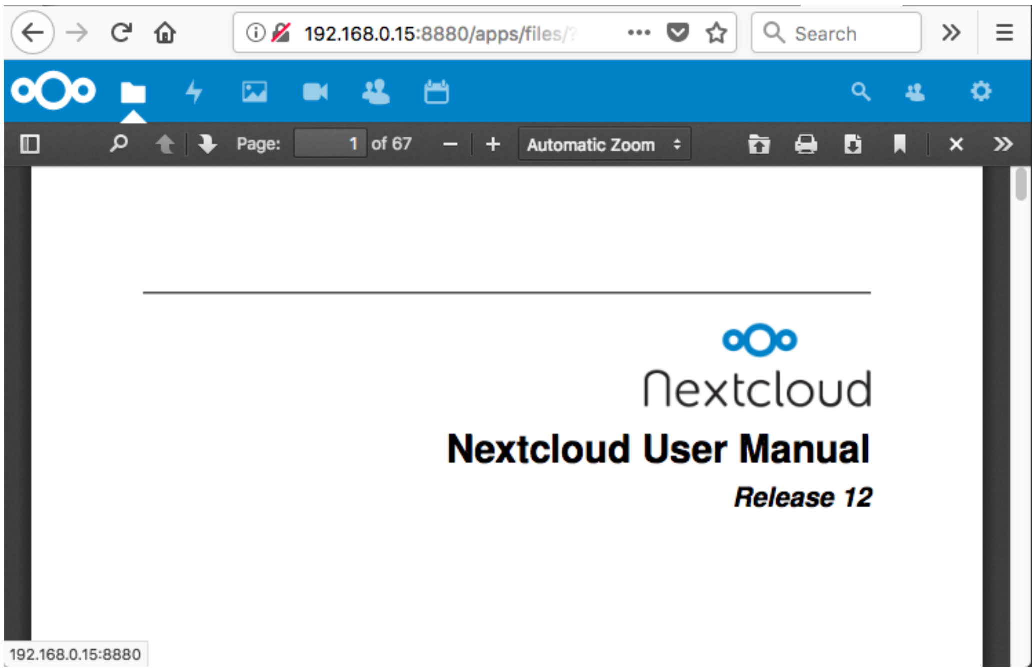


Figure 2.20: NextCloud: Group calendar application.

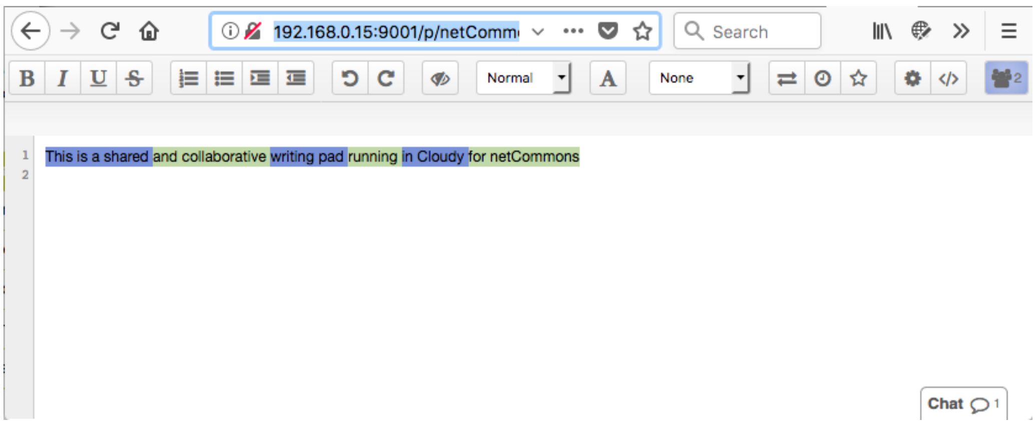


Nextcloud

Nextcloud User Manual

Release 12

Figure 2.21: NextCloud: PDF viewer.



This is a shared and collaborative writing pad running in Cloudy for netCommons

Figure 2.22: Etherpad running on a Cloudy instance with two remote editors

2.4 Service monitoring

Information on service availability in Cloudy provides useful data to design and enable new features. Therefore, it was considered important to understand better the options for a monitoring system in Cloudy. For this purpose we developed and evaluated a monitoring system leveraging the gossip platform used in Cloudy. The work described here has lead to the publication of [9].

2.4.1 Motivation and goals

Current solutions for monitoring services under classic cloud systems support and designed for the use in data centers, which disregards the unique properties that an edge cloud environment has, such as the high latency between nodes, changes in network (nodes churn rate), and most importantly the use of low-capacity devices. Furthermore, an issue found in the edge cloud computing (such as community network clouds) is the lack of a suitable mechanism for logging, monitoring of resources and services, and dissemination of information.

The main goal of this work is to investigate an efficient way for monitoring services in wireless community network clouds, as a case study of edge cloud computing, using gossip-enabled networks to achieve efficient data dissemination and sharing. The platform considers a decentralized way to handle the dissemination of data, by using a gossip overlay to communicate with the nodes. Also, the platform gathers distributed data, by using the gossip-enabled network through which it disseminates data.

2.4.2 Monitoring platform for edge clouds

Gossip protocols rely on disseminating information by utilizing a small subset of neighboring nodes to pass data towards the whole network, instead of flooding the network or using a single server. Thus, each neighbor is required to disseminate the messages only to its direct neighbors, forming a directed graph over the current networks to achieve quick and efficient dissemination.

The community network environment creates its own challenges, differing from classic cloud environments, which need to be addressed, such as low-capacity devices used, network changes (node churn rate), low bandwidth and high latencies between nodes and user related privacy concerns. In our work, we address these challenges by using eventual consistency and gossiping methods on the shared data, while also making sure that, by means of Serf, each node can join or leave the network without affecting the overall information within the network.

The type of information monitored can include the resources, services and social aspects of the community. For our work, one of the main reasons to gather knowledge on the community cloud system is to have a tool to understand social behavior on the network and the usage of services and resources. Also, the monitoring data can be extended with additional types of information, such as service configuration, resource configuration and usage.

The monitoring platform is split in three stages: 1) Logging of raw data from services, resources and user interaction shared by the member nodes; 2) processing of the raw data, into a format that is user readable and can be shared among the community, such as a time-line of service usage; 3) showing the results to the users in the community, through the user interface, which is an additional service in Cloudy. These stages represent the major objectives on which we focus our attention, tailoring them towards an efficient monitoring service on the edge community cloud.

2.4.3 Experimental analysis

We conduct an evaluation by emulation of the monitoring platform, as the means to understand the characteristics and properties relevant to a real deployment. The simulation of the gathered network gives us an insight into best practices for an efficient way of dealing with dissemination of data across a wireless mesh network. The emulation of nodes provides details about the scalability of the monitoring platform for edge cloud computing. In this way, we can understand the properties and issues that arise when dealing with higher number of nodes, higher latencies and the use of low-capacity devices.

The characteristics of the monitoring platform come into evidence when we analyze the results of data dissemination and convergence, scalability of the platform and the tuning of Serf properties. Data convergence gives us the amount of data that a node receives across time, from the total disseminated data. This means that a certain amount of time passes until a node gathers the total data (or convergence time). The time elapsed between disseminating and convergence of data is then important to understand how consistent the system is when using a gossip overlay.

For the evaluation, we emulate nodes and simulate the network, which allows us to obtain the average of time for dissemination and convergence data rates, while also being able to tune certain aspects (such as gossip interval, gossip fan-out) of the overlay created through Serf. We used the Mininet simulator [10], merged with Mininet-Wifi [11] and Mininet ContainerNet[1]. Mininet-Wifi adds to Mininet the ability of simulating wireless links. The capability of simulating wireless links is then more attuned with the environment created with community network clouds. The ContainerNet project enables each of the emulated nodes to run as Docker containers, guaranteeing execution isolation for each of the emulated nodes. Furthermore, we are able to run different executions of the same applications, such as Serf, without interference among each emulated node.

In the experiments we used as network topology a random geo-positioning of the nodes, where each of the nodes positions itself within a maximum range of 100 meters from another node. The characteristics of the topology are drawn from the CNs, where each person connects to their nearest neighbors to join the network. Therefore, each experiment run uses a randomly created topology, in an attempt to not be influenced by a given network topology.

Moreover, the positioning of the nodes influence the overall latencies in the network. The nodes average latencies can be as high as 800 milliseconds.

The evaluation is performed with several runs (around 10) and their results are averaged. The average on these runs are enough to point out the characteristics that determine the efficiency of the monitoring platform. Each experiment has 40 virtual nodes (Docker containers), interconnected through a virtual mesh network and randomly positioned in the network.

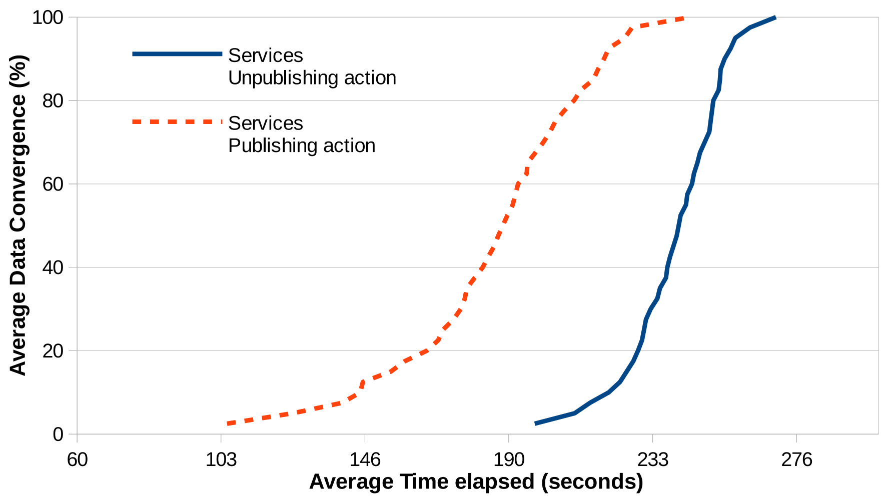


Figure 2.23: Averaged data convergence in the time elapsed for the actions of publishing and unpublishing services.

[1] https://github.com/mpeuster/containernet

From the experimental results we can observe the rate of data convergence across the simulated network and infer on the data dissemination. Fig. 2.23 shows the percentage for average monitoring data convergence for all nodes, after the services were published (dashed line) and unpublished (continuous line), averaged from all the experiments done, where the 100% is reached when the data on all nodes is consistent. The actions are not immediately propagated to the network, therefore the dissemination occurs some time after starting. Also, we can see that the convergence of data in both actions happens within 1 minutes in high latency conditions. We can observe that the convergence is done faster for the unpublish action than for publish. This is because the nodes already know about others in the network, and thus it only requires to update the current information. The figure also demonstrates for each action the data convergence on all nodes, which on average is 130 and 80 seconds in each action, respectively.

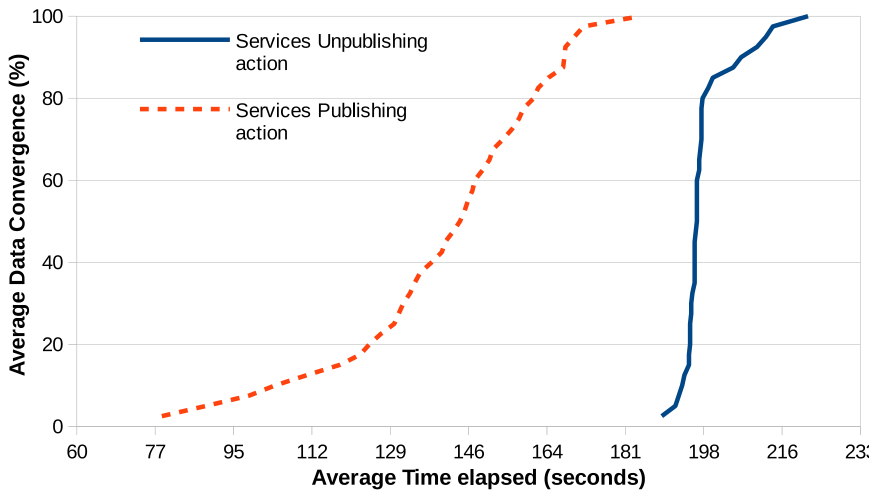


Figure 2.24: Average data convergence in time elapsed for the actions of publish and unpublish services, using the tuned gossip properties for wireless devices.

Fig. 2.24 depicts the same actions (Publish and Unpublish of services) as previously, however the gossip properties (Gossip interval and Gossip fan-out) were adjusted for wireless environments. The figure shows that the optimization done has an effect on the data dissemination and convergence on the nodes. Furthermore, the figure also demonstrates that the convergence of data for the nodes, on average, is 87 and 38 seconds in each action respectively. Therefore, each of the actions shown, on average, are faster (around 25%) for the wireless community networks scenario than with the previous gossip properties.

2.4.4 Monitoring system prototype

A prototype for the monitoring platform was developed using the available low-capacity devices and connected to the community network (Guifi.net). The prototype showed the feasibility for monitoring services and resources in the edge cloud computing scenario. Therefore, services were started and terminated at certain intervals of time in the available nodes, in order to gather service usage information. The information for each service is sent to Serf when a service is published. The information is updated in Serf when the service terminates. Thus, all nodes can gather the logs of services within the data coming from Serf members (nodes interconnected in the gossip overlay).

From the prototype monitoring platform, we could observe that using a gossip overlay to disseminate information is as a suitable option for a non-critical analysis of the overall network. The shared information can be gathered at any of the nodes that are members of the gossip overlay, within certain conditions such as the time delay for the eventual consistency of the data, and the amount of data that is updated to each node.

With regards to the latter observation, we also observed that the size of the payload of the Serf messages is a limitation which may affect further extensions of the monitoring system.

2.4.5 Final insight

In our experiments with node emulation and network simulation, we showed that the dissemination of data over a gossip-enabled network is done fast enough and is suitable for non-critical shared information. We also observed that under high latency situations and with low-capacity devices the use of a gossip-enabled network is a best practice to overcome the harsh conditions of edge networks, while removing the need to flood the network with information, or to know the structure of the whole infrastructure.

The directions for future research should consider studying solutions to overcome the payload limitation of the Serf platform. One possibility is an extension of the support services by InterPlanetary File System (IPFS)[2], which does not rely on any centralized component, and appears to be an option matching very well the paradigm of CNs. Another issue is to consider the limits of information sharing with regards to the privacy related aspects that can arise from contributing data on resources and services of user devices.

2.5 Service placement

Just as monitoring services provides the necessary insight to improve them and maintain their functionality, the correct placement of services in microclouds can improve performance. By putting services closer to consumers, microclouds pursue not only a improving certain aspects of service provision, but also a low entry barrier for the deployment of mainstream Internet services within the CN. Unfortunately, the provisioning of the services is not so simple. Due to the large and irregular topology, high software and hardware diversity of CNs, a "careful" placement of microclouds and services over the network is required to optimally leverage the available resources. This section discusses the work that also lead to the contribution presented in [12].

2.5.1 Goals and model

A microcloud is a platform to deliver services to a local community of citizens within the vast CN. Services can be of any type, ranging from personal storage to video streaming.

The model of microclouds is *entirely different from fog computing*, which extends centralized cloud computing by introducing an intermediate layer between devices and datacenters, leaving the central entity in full control of the system. Microclouds take the opposite track: by placing services close to consumers, a microcloud goal is that no (or minimal) action takes place in the Internet. The idea is to tap into the shorter, faster connectivity between users to deliver a better service and alleviate overload in the backbone links.

The microcloud approach, however, poses new challenges, such as that of the optimal placement of microclouds within the CN to overcome suboptimal performance. In addition, a placement algorithm that is agnostic to the state of the underlying network may lead to important inefficiencies. Although conceptually straightforward, it is challenging to calculate an optimal decision due to the dynamic nature of CNs and usage patterns.

In this part of our work the following questions are investigated:

[2]Citing directly from its descriptive white paper IPFS https://ipfs.io/ is

> "*. . . a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of files. In some ways, IPFS is similar to the Web, but IPFS could be seen as a single BitTorrent swarm, exchanging objects within one Git repository. In other words, IPFS provides a high throughput content-addressed block storage model, with content-addressed hyper links. This forms a generalized Merkle DAG, a data structure upon which one can build versioned file systems, blockchains, and even a Permanent Web. IPFS combines a distributed hashtable, an incentivized block exchange, and a self-certifying namespace. IPFS has no single point of failure, and nodes do not need to trust each other.*"

1. Given that sufficient state information is in place, is network-aware placement enough to deliver satisfactory performance to CN users?
2. Can the redundant placement of services further improve performance?

These questions are closely related to the optimal topology control for video streaming studied in [13, 14, 15]

2.5.2 Service placement proposal

Considering the concepts of service and network graphs, we can formulate the problem statement more precisely as: *"Given a service and network graph, how to place a service on a network as to maximize user QoS and QoE, while satisfying a required level of availability for each node (N) and considering a maximum of k service copies ?*

Let B_{ij} be the bandwidth of the path to go from node i to node j. We want a partition of k clusters (i.e., services) : $C \leftarrow C_1, C_2, C_3, ..., C_k$ of the set of nodes in the mesh network. The cluster head i of cluster C_i is the location of the node where the service will be deployed. The partition maximizing the bandwidth from the cluster head to the other nodes in the cluster is given by the objective function:

$$\arg\max_C \sum_{i=1}^{k} \sum_{j \in Ci} B_{ij} \tag{2.1}$$

with respect to the following constraints:

1. The total bandwidth used per link cannot exceed the total link capacity:

$$\forall e \in \mathbb{E} : \sum_{s1,s2 \in \mathbb{S}} X_{s1,s2}(e) \times \beta_{s1,s2} \leq B_e \tag{2.2}$$

2. Availability-awareness: the node availability should be higher than the predefined threshold λ:

$$\forall n \in \mathbb{N} : \sum_{n \in \mathbb{N}} R_n \geq \lambda \tag{2.3}$$

Table 2.1: Input and decision variables

Symbol	Description
N	set of physical nodes in the network
E	set of edges (physical links) in the network
S	set of services
D	set of service copies
k	max number of service copies
B_e	bandwidth capacity of link e
$\beta_{s1,s2}$	bandwidth requirement between services s1 and s2
R_n	Availability of node n
λ	Availability threshold
$X_{s1,s2}$	use of physical link e by at least one service for the placement of virtual link between s1 and s2, 1 iff placed

3. Admission control: At most, k copies can be placed for each service:

$$|D| = k \tag{2.4}$$

Solving the problem stated in Equation 2.1 in brute force for any number of N and k is NP-hard and very costly. The naive brute force method can be estimated by calculating the Stirling number of the second kind [16] which counts the number of ways to partition a set of n elements into k nonempty subsets, i.e., $\frac{1}{k!}\sum_{j=0}^{k}(-1)^{j-k}\binom{n}{k}j^n$ $\Rightarrow \mathcal{O}(n^k k^n)$. Thus, due to the obvious combinatorial explosion, we propose a low-cost and fast heuristic called *BASP*. The Bandwidth and Availability-aware Service Placement (BASP) allocates services taking into account the bandwidth of the network and the node availability. The symbols used are listed in Tab. 2.1.

The BASP algorithm is given as follows (see Algorithm 1):

2.5.3 Evaluation and results

We take a network snapshot (capture) made of 71 physical nodes of the QMP network[3] regarding the bandwidth of the links and node availability. The node and bandwidth data obtained has been used to build the topology graph of the QMP. The QMP topology graph is constructed by considering only operational nodes, marked in ”working” status, and having one or more links pointing to another node. Additionally, we have discarded some disconnected clusters. The links are bidirectional and unidirectional, thus we use a directed graph. The nodes of QMP consists of Intel Atom N2600 CPU, 4GB of RAM and 120 GB of disk space. Our experiment is comprised of 5 runs and the presented results are averaged over all the runs. Each run consists of 15 repetitions.

Fig. 2.25 depicts the average bandwidth to the cluster heads obtained with the *Random*, *Naive K-Means* and the *BASP* algorithm. Here, random placement refers to the situation when service deployment is not centrally planned but initiated individually by the CN members. The figure reveals that for any number k of services, *BASP* outperforms both *Naive K-Means* and *Random* placement.

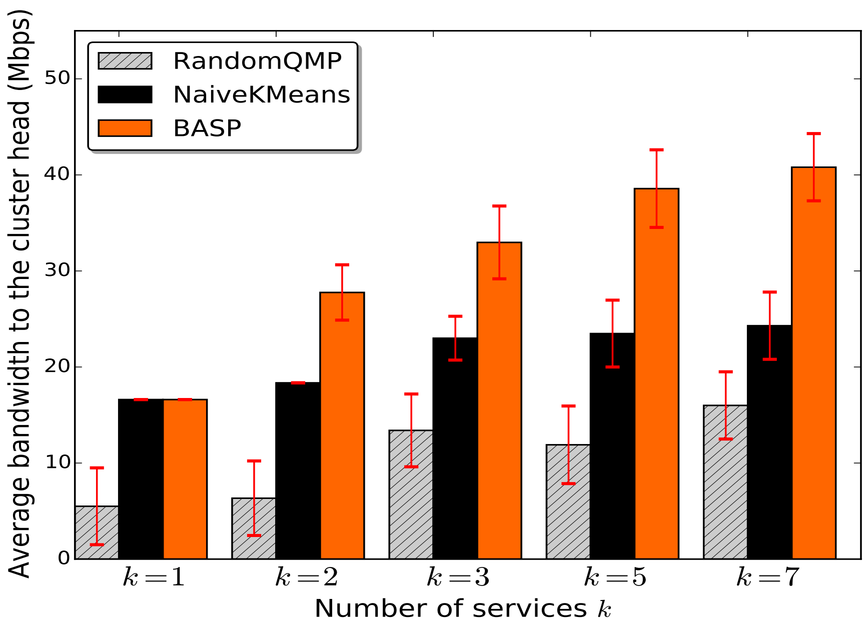


Figure 2.25: Average bandwidth to the cluster heads.

For $k = 2$, the average bandwidth to the cluster heads is increased from 18.3 Mbps (*Naive K-Means*) to 27.7 Mbps (*BASP*), which represents a 50% improvement. The largest increase of 67% is achieved when $k = 7$. On

[3]http://tomir.ac.upc.edu/qmpsu/index.php?cap=56d07684, part of guifi.net.

Algorithm 1 B A S P

```
Require: $G(N, E)$                                    ▷ Network graph
    $C \leftarrow C_1, C_2, C_3, ..., C_k$             ▷ $k$ partition of clusters
    $B_i$                                              ▷ bandwidth of node $i$
    $R_n, \lambda$                ▷ availability of node $n$, $\lambda$ availability threshold

 1: procedure PERFORMKMEANS($G, k$)
 2:     if $R_n \geq \lambda$ then
 3:         return $C$
 4:     end if
 5: end procedure
 6: procedure FINDCLUSTERHEADS($C$)
 7:     $clusterHeads \leftarrow list()$
 8:     for all $k \in C$ do
 9:         for all $i \in C_k$ do
10:             $B_i \leftarrow 0$
11:             for all $j \in setdiff(C, i)$ do
12:                 $B_i \leftarrow B_i + estimate.route.bandw(G, i, j)$
13:             end for
14:             $clusterHeads \leftarrow \max B_i$
15:         end for
16:     end for
17:     return $clusterHeads$
18: end procedure
19: procedure RECOMPUTECLUSTERS($clusterHeads, G$)
20:     $C\prime \leftarrow list()$
21:     for all $i \in clusterHeads$ do
22:         $cluster_i \leftarrow list()$
23:         for all $j \in setdiff(G, i)$ do
24:             $B_j \leftarrow estimate.route.bandw(G, j, i)$
25:             if $B_j$ is best from other nodes $i$ then
26:                 $cluster_i \leftarrow j$
27:             end if
28:             $C\prime \leftarrow cluster_i$
29:         end for
30:     end for
31:     return $C\prime$
32: end procedure
```

average, when having up to 7 services in the network, the gain of *BASP* over *Naive K-Means* is of 45%. Based on the observations from Figure 2.25, the gap between the two algorithms grows as k increases. We observe that k will increase as the network grows. And hence, *BASP* will presumably render better results for larger networks than the rest of strategies.

Regarding the comparison between *BASP* and *Random* placement, we find that *Random* placement leads to an inefficient use of network's resources, and consequently to suboptimal performance. As depicted in Fig. 2.25, the average gain of *BASP* over naive *Random* placement is 211%.

In order to understand the gains of our network-aware service placement algorithm in a real production CN, we deploy our algorithm in real hardware connected to the nodes of the QMP network, located in the city of

Barcelona. We concentrate on benchmarking two of the most popular network-intensive applications: *Live-video streaming service*, and *Web 2.0 Service* performed by the most popular websites [17].

We use 20 real nodes connected to the wireless nodes of QMP. These nodes are co-located in either users homes (as home gateways, set-top-boxes, etc.) or within other infrastructures distributed around the city of Barcelona. They run the Cloudy system. As the controller node, we leverage the experimental infrastructure of Community-Lab[4]. Community-Lab provides a central coordination entity that has knowledge about the network topology in real time and allows researchers to deploy experimental services and perform experiments in a production CN. The nodes of QMP that are running the live video streaming service are part of Community-Lab. In our experiments, we connect a live streaming camera (maximum bitrate of 512 kbps, 30 frame-per-second) to a local PeerStreamer instance that acts as a source node.

Figure 2.26 shows the average chunk loss for an increasing number of sources k. The data reveals that for any number of source nodes k, BASP outperforms the currently adopted random placement in QMP network. For $k = 1$, BASP decreases the average chunk loss from 12% to 10%. This case corresponds to the scenario where there is one single source node streaming to the 20 peers in the QMP network. Based on the observations from Figure 2.26, the gap between the two algorithms is growing as k increases. For instance, when $k = 3$, we get a 3% points of improvement w.r.t. chunk loss, and a significant 37% reduction in the loss packet rate.

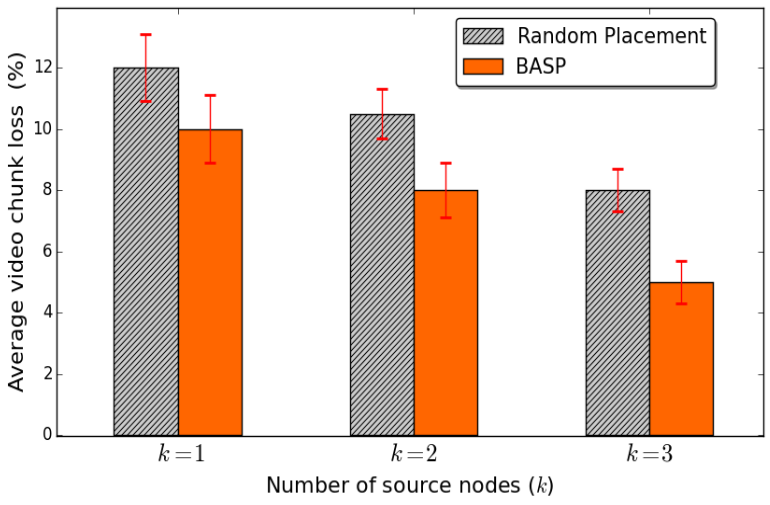


Figure 2.26: Average video chunk loss in QMP.

Fig. 2.27 and Fig. 2.28 report the response time observed by three clients for the update live feed operation, when placing the web server with *Random* and *BASP*, respectively.

When placing the web server with the *Random* approach, Figure 2.27 reveals that, as far as we increase the number of threads (i.e., concurrent operations) per client, the response time increases drastically in three clients. For up to 120 operations per client (i.e., 20 threads), all clients perceive a similar response times (300-350 ms). Response time increases more than one order of magnitude in Client 2 and Client 3, and an order of magnitude in Client 1 when performing 160 operations (i.e., 80 threads)

2.5.4 Final insight

It is clear that even a low-complexity service placement heuristic such as the proposed BASP can greatly improve the bandwidth allocation when deploying CN microclouds. We presented algorithmic details, analysed its complexity, and carefully evaluated its performance with realistic settings, so that deployment in CNs can be safely undertaken in the future. Our experimental results show that BASP consistently outperforms the

[4]https://community-lab.net/

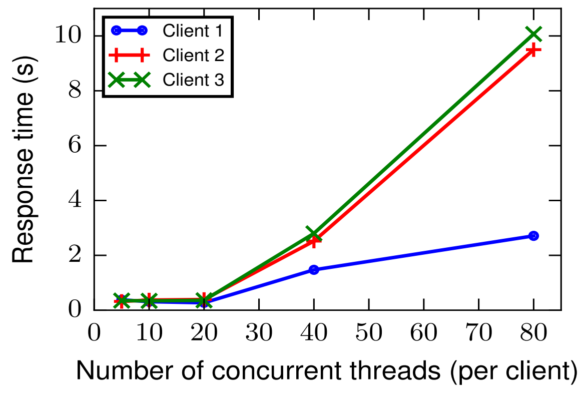


Figure 2.27: UpdateActivity-Random

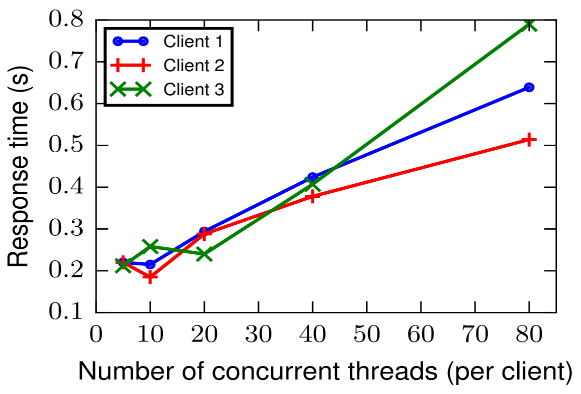


Figure 2.28: UpdateActivity-BASP

currently adopted random placement in `Guifi.net` by 211%, i.e., more than a factor of two! Moreover, as the number of services increases, the gain tends to increase accordingly. Furthermore, we deployed our service placement algorithm in a real network segment of QMP network, a production CN, and quantified the performance and effects of our algorithm. We conducted our study on the case of a live video streaming service and Web 2.0 Service integrated through Cloudy distribution. Our experimental results show that when using the BASP algorithm, the video chunk loss in the peer side is decreased up to 3% points, i.e., worth a 37% reduction in the loss packet rate. When using the BASP with the Web 2.0 service, the client response times decreased up to an order of magnitude, which is a significant improvement.

2.6 Decentralized access to the Internet: proxies and Internet gateways

While a community network may offer different applications, Internet/Web access is the most popular application in most Community Networks (CNs). Web proxy nodes (application layer gateways) or IP tunnel servers (network layer gateways), connected both to a CN and an ISP are often used by community network users as Internet gateways. Most of these gateways are free as they are giving away the spare Internet capacity that many Internet connections provide. However there are also comparable schemes where a common Internet access is shared by a community of users and the cost is proportionately shared among them. One example of that is the eXO association with about 70 users use IP tunnels to share Internet capacity bought in the guifi.net Internet gateway. Many of the web proxies run on simple servers at the homes of the community members that donate their spare Internet capacity for free. In the Cloudy distribution, this proxy service was integrated at the beginning of the Cloudy development and is part of a set of services for network management offered by Cloudy.

We have been looking at ways to improve different aspects of this service. One of the limitations of the current proxy service in Guifi.net is that proxies are selected manually by the users. This may have the effect that many users choose and burden "popular" proxies. As a consequence, potential (congested) hotspots can reduce the quality of the user experience. In addition, the non-optimal usage of the access links saturates the available bandwidth of these links and the shared bandwidth to the ISP, while the community member who voluntarily shares her connectivity may suffer performance degradation in her Internet access service. This field has an important potential for Cloudy since understanding how to build an automated proxy assignment could lead to the design of a component, which supports this functionality in the Cloudy's proxy service. This year we have focused on research in three aspects: usability, scalability, and fairness, which has resulted in several research publications.

There is a rich body of work focused on reducing the cost and increasing Internet coverage under several scenarios. For instance, the Lowest Cost Denominator Networking (LCD-Net) [18] explores resource pooling Internet technologies to support benevolence on the Internet. Some of these ideas are illustrated by WiFi access sharing schemes, community-led (PAWS [19]) or commercially-run (FON [20]), where home broadband subscribers donate their controlled (but for free) broadband Internet spare capacity to fellow citizens. This is done by sharing a fixed portion of throughput [21]. In contrast, our work considers not just local access to a shared WiFi hotspot, but also remote access to Internet gateways over a CN that can use any network technology, such as, wired or wireless meshes. Our research also focuses on spare capacity, with little or no visible impact on the primary user. This means secondary users can get from all to nothing, depending on the demand from primary use. This activity has lead to several publications, that are important for WP3, albeit they are not directly related to the software development in T3.2. The following paragraphs summarize these contributions and point to the relevant publications.

Usability: Analysis of a crowdsourced Web proxy service We considered here the environment of Internet access sharing through Web proxy gateways in local or regional CNs. In an effort to understand the effectiveness and performance of this shared Internet access, we performed a measurement study of a crowdsourced Internet proxy service in the guifi.net CN, that provides free Internet access to a large community with a high users to proxies ratio. Our study focused on a representative subset of the whole network with about 900 nodes, 4 proxies and roughly 470 users of the Web proxy service. We analyzed the service from three viewpoints: Web content of users' traffic, performance of proxies and influence on the access network. We saw that Web proxies in a CN can be used to provide basic Internet access. Nevertheless, we observed the necessity of a regulation mechanism that would enable the fair and efficient usage of the available resources, considering the entire sets of users and proxies instead of a local level.
The main results related with this contribution are presented in [22].

Scalability: Web Proxy selection mechanism To facilitate the fair usage of the Web proxy service we investigated and built a selection mechanism for client-proxies. We developed a client-side distributed system that optimizes the client-proxy mapping, agnostic to the underlying infrastructure and protocols, requiring neither any modification of proxies nor to the underlying network [23]. Clients choose proxies taking into account network congestion, proxy load and proxy performance, without requiring a minimum number of other participating clients in order to select an adequate proxy. Our proposal was evaluated experimentally with clients and proxies deployed in guifi.net. We show that our selection mechanism avoids proxies with heavy load and slow internal network paths, while achieving a network overhead linear to the number of clients and proxies. Moreover, we demonstrated that informed proxy selection and admission control in proxies could alleviate up to a certain level imbalances and unreliability, and also improve the overall service with small additional overhead [24]. Nevertheless, the Internet sharing process can negatively affect the service experience of the users that share their connections, thus jeopardizing the continuity of this community service.
The main results of this research are presented in [23, 24].

Fairness: Secondary usage of spare Internet capacity Building upon the studied model of proxy usage, we argue that it is important to differentiate between primary users, who share their Internet connection, and the secondary users in order to preserve the contribution of the former group. To guarantee that there is no additional cost incurred for the users sharing their connections we proposed a mechanism where a middlebox separates the traffic of the primary users from that of the secondary users. We evaluated the efficiency and behavior of several traffic separation mechanisms, in order to determine how to maximize network utilization and usage of the spare network capacity, while minimizing the possible impact on the primary traffic. Finally, we presented a set of recommendations to achieve the best performance isolation for the primary user, without significant impact to the perceived experience of the secondary users.
The main results related with this research are presented in [25].

The lessons learned, and the ongoing research on IP tunnels as alternative gateways to Web proxies, will support future improvements of this service in Cloudy.

2.7 Takeaway

By integrating Docker-compose into Cloudy, we have expanded the scope of applications which can be deployed in Cloudy. As a consequence, a much larger set of applications, such as many Web-based applications consisting of front-end and back-end provided as separate Docker images, can now be deployed in an user-friendly way. Now Cloudy can interconnect (compose) several containers to run more complex services. The increase of the reach of applications for which Cloudy can be used is important for the usability and applicability of Cloudy in diverse scenarios. Specifically, there is potential for the IoT domain, where local data management and data privacy comes into play. We have started to explore the deployment of IoT services from the community-editions of popular commercial service providers. Additional Docker containers supported, like HomeAssistant, bring also support for a wide range of smart home devices, like temperature or security sensors, actuators, or digital media player devices such as Chromecast.

Several directions to enhance Cloudy with additional features were researched. One aspect we investigated was service monitoring, which is important since it strongly relates to the distinctive characteristic of Cloudy given by the publication of services within the Cloudy microcloud. We observed the suitability of the chosen Serf platform with regards to fast service publication, but also observed the limitation of the payload size, which is an obstacle for the growth in the number of services. For this purpose, we have developed support to combine Serf with IPFS to offload Serf from data that can be served by IPFS. The result seems promising since IPFS, like Serf, operates in a decentralized way, which is considered suitable for the organic growth of microclouds.

The aspect of service placement was investigated. Its potential lies in the capability to operate services in a more optimized way in resource-constrained microclouds. The findings from the service placement work improve the position of Cloudy with regards to its capability to achieve a determined quality of service and quality of experience, which can become important for critical applications.

Our work on investigating the usage and management of the proxy servers within Guifi.net targets to improve the performance of a major service of Guifi.net, which is Internet service provision. The gained insights help to understand more options for the management of this service, which is key for many CN users.

In year three of netCommons, the achieved usability of Cloudy should sought to be combined with important applications domains, like the IoT, and target enablers, such as deep learning techniques. We can observe a strongly increasing research community in edge computing and the appearance of many solutions which leverage similar technology as Cloudy, but our platform has a distinctive feature given by its participatory service provision, for which we should see to make a differentiated value proposition. While application such as deep learning still run in data centers, we can observe strong research efforts to move components closer to the network edge, and Cloudy may find important tools and use cases for local analytics and data governance if this tendency of the research community to enable local machine learning continues.

2.8 List of publications related to T3.2

1. "Internet Access for All: Assessing a Crowdsourced Web Proxy Service in a Community Network", E. Dimogerontakis, R. Meseguer and L. Navarro., in *Proceedings - Passive and Active Measurement Conference (PAM 2017)*, Sydney, Australia, 2017.
2. "Gossip-based Service Monitoring Platform for Wireless Edge Cloud Computing", N. Apolónia, Freitag, F., and Navarro, L., in *Proc. of the 14th IEEE International Conference on Networking, Sensing and Control (ICNSC)*, Calabria, Italy, 2017.
3. "Client-Side Routing-Agnostic Gateway Selection for heterogeneous Wireless Mesh Networks", ME. Dimogerontakis, Neto, J., Meseguer, R., Navarro, L., and Veiga, L., *Proc. of the IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Lisbon, Portugal, 2017.
4. "Practical Service Placement Approach for Microservices Architecture", M. Selimi, Cerdà-Alabern, L.,

Sánchez-Artigas, M., Freitag, F., and Veiga, L., in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2017)*, Madrid, Spain, 2017.

5. ”On Edge Microclouds To Provide Local Container-based Services”, R. Baig, R. Pueyo, F. Freitag, L. Navarro, in *Global Information Infrastructure and Networking Symposium (GIIS'2017)*, Saint-Pierre, France, 2017.
6. “Community Sharing of Spare Network Capacity”, E. Dimogerontakis, Meseguer, R., Navarro, L., Ochoa, S. F., and Veiga, L., in *IEEE International Conference on Networking, Sensing and Control (ICNSC)*, Calabria, Italy, 2017.
7. “Design Trade-offs of Crowdsourced Web Access in Community Networks”, E. Dimogerontakis, Meseguer, R., Navarro, L., Ochoa, S. F., and Veiga, L., in *IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, Wellington, New Zealand, 2017.

3 Developments on PeerStreamer

This chapter describes the current effort in the development of the PeerStreamer application, and its deep integration with the Cloudy platform, plus the recent advances in the research themes related to the use of PeerStreamer over CNs.

The expected outcome for the T3.3 Task at M24 is the development of the new version of PeerStreamer with a specific support for CNs, and the enhancement of PeerStreamer to become a web-conferencing system. During the evolution of the project we decided to focus on the integration of PeerStreamer into the new (Docker-based) Cloudy platform, in order to give to the CNs a unique tool that embeds several useful applications in a web-based, easy-to-use platform. While this effort was not initially planned we decided to include it in order to maximise the impact of the project on CNs.

In general, having people install and use different platforms is discouraging and hinders adoption. If, instead, people can have access to a multitude of services through a single platform and an easy interface, we increase our chances to have early adopters, feedback, consequent improvement and wider community engagement. In a nutshell, we can have a better product and wider adoption. For this reason we decided to make the effort of re-designing PeerStreamer in order to be fully integrated with Cloudy. That required mainly two new features:

- Integrating PeerStreamer into the Cloudy with support for the SERF protocol;
- Transforming PeerStreamer I/O components into a web-based platform.

The first task was successfully carried out, as is detailed in Sec. 3.1. Likewise, we implemented the web-based version of PeerStreamer, which will be tested with the communities in period M24-M36 (as stated in the DoA). Due to some difficulties given by the phasing out of the Netscape Plugin Application Programming Interface (NPAPI) libraries (see Sec. 3.2) and the effort needed for Cloudy integration, the enhancement of PeerStreamer as a conferencing tool has been slightly delayed. Currently, we have a stable design and an initial implementation that needs more internal testing, while modifications to the web interface are still undergoing. This effort will be performed in the period M24–M36 and will be documented in D3.5 at M36.

The goal of T3.3 is to explore how live P2P video streaming techniques can form the basis of key applications like local event broadcasting, e-learning platforms and group video conferencing in CNs. The Cloudy platform, as described in D3.2 [3] and further developed as described in Chapter 2, is a generic GNU/Linux distribution that easily supports the creation of internal community applications. It was extended in netCommons to support the Docker platform, which makes it even easier to extend the set of available applications. Nevertheless, the initial experiences with Cloudy have shown that it is used mainly by “power-users” to monitor the state of the network and of their nodes, while its use as a generic community set-top-box is still not widespread.

We believe that PeerStreamer can contribute to the diffusion of Cloudy, and, at the same time, it can benefit from the the Cloudy platforms in terms of simplified installation and management. Thus, after the first year, where we completed a stripped-down version of PeerStreamer that can be installed directly on network nodes as documented in D3.2, we now followed up with work in three areas:

- The better integration of PeerStreamer into Cloudy;
- The extension of PeerStreamer as a multi-stream application;
- The design of PeerStreamer as a P2P video conferencing system.

All the realized source code can be found in the `PeerStreamer-ng`, `psng-pyserf` and `PeerStreamer-Docker` repositories in the project Github account https://github.com/netCommonsEU/.

3.1 Integration of PeerStreamer into Cloudy

Cloudy is a complex system that interconnects three main components:

- The Docker container system;
- the Serf protocol[1];
- the CN.

Docker is a well known container solution that allows to run processes in separated environments. T he services on a Cloudy distribution can be expanded adding containers with new applications that run locally in the user's Cloudy node. Serf is a gossip-based membership protocol, that makes it possible to distribute in a decentralized way information among groups of members. Information can relate to the current composition of the group (addition and removal of members) and custom events that one member generates and wants to distribute to the other nodes. Cloudy uses Serf to distribute the knowledge about a new service in the network, after a new container has been downloaded and run by a node. Finally, Cloudy relies on a running CN to work and assumes that IP addresses are all routable and there are no Network Address Translation (NAT) devices in between.

PeerStreamer (PS, for brevity) is a live P2P streaming platform, already described in D3.2 [3]. It had been already integrated in Cloudy before it switched to Docker. PS uses its own gossiping system to distribute the video chunks among the swarm of peers that participate in the streaming session, and thus, apart from being able to run and stop the service, the integration with Cloudy was only partial. After the work to integrate PS in the wireless router [3], in this deliverable we describe our effort to integrate PS in Cloudy, thus, addressing another use-case.

The main sub-tasks of this activity were:

- Making PeerStreamer available as a library of files: PeerStreamer became a set of independent components that ease the integration of the application to other platforms. This is a revolution in the PS architecture that was designed to run as a separate application, and forms the first step for the work needed to include multi-stream support in PS, which is useful also for the development of PS as a conference system (see Sec. 3.3). For this reason we renamed this system as next-generation PeerStreamer, PS-ng.
- Dockerizing PeerStreamer. We realized a custom Docker image for PS, based on the Ubuntu 16.04 image that can be downloaded from the Docker Hub[2]. The image contains PS, which is run in a container and appears among the available services in Cloudy.
- Integrating PS with the Serf protocol: We realized a C language interface to make Serf communicate with PS, so that new live video streams can be published on Serf. So, not only can Cloudy be used to run PS, but Serf can be used by the instances of PS to notify each other about the presence of new videos.
- Realizing a Web interface for PS, which can be accessed by the user with a browser, in order to make the user experience as smooth as possible.

3.1.1 Introducing PS-ng

Fig. 3.1 reports the configuration we imagine for PS-ng when integrated into Cloudy. The home network of a user is provided by his wireless router that is connected to the CN and to the Cloudy device. For our testing purposes, we used the Minix NEO Z64 which can be purchased for less than 130€ and was initially used by UPC to develop Cloudy. The device as of today has been superseded by a new model (Neo Z83-4) for a similar price. As Cloudy has been tested on the cheaper widespread Raspberry PI boards, we will test PS on that device in Y3. The Cloudy device can be connected to the router with a wireless or wired link, it should have an IP

[1] See https://www.serf.io/

[2] See https://cloud.docker.com/swarm/nfunitn/repository/docker/nfunitn/peerstreamer/general.

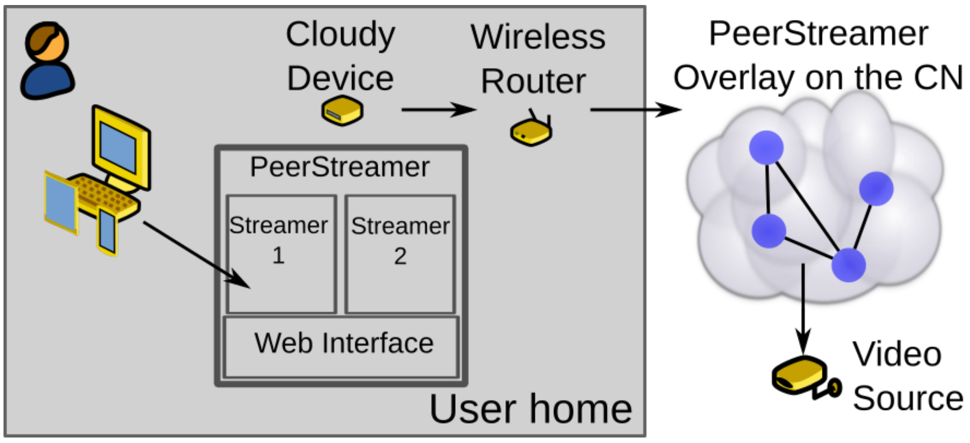
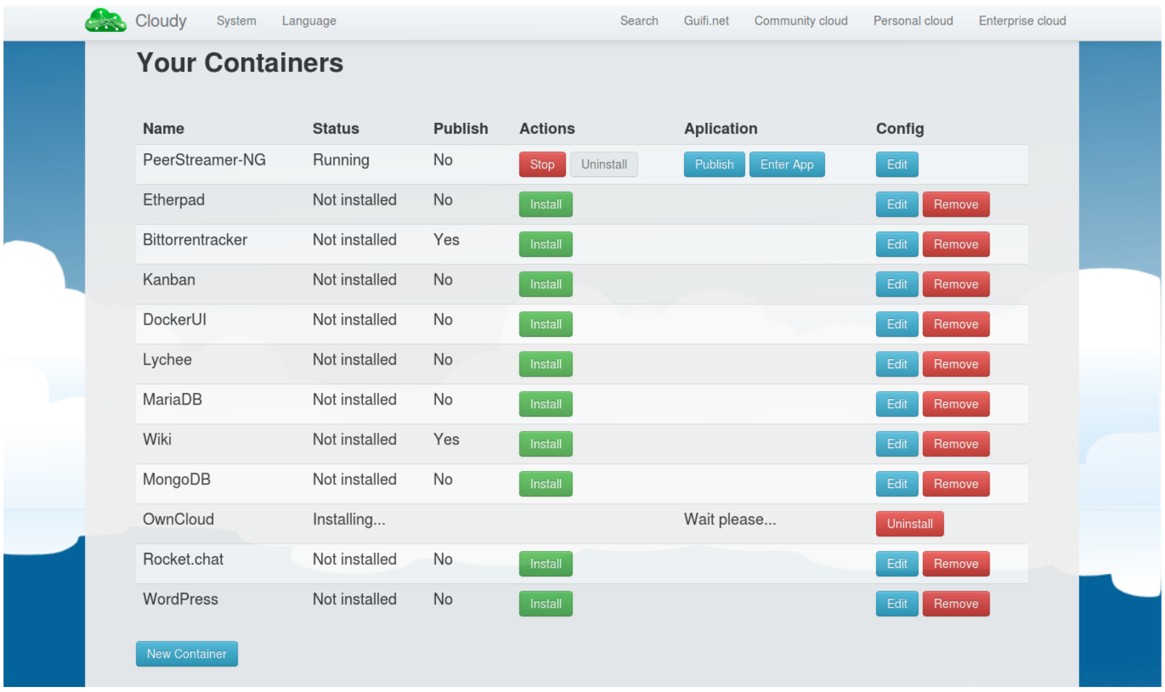


Figure 3.1: The components of the live streaming system inside a home network.



Your Containers

Name	Status	Publish	Actions	Aplication	Config
PeerStreamer-NG	Running	No	Stop Uninstall	Publish Enter App	Edit
Etherpad	Not installed	No	Install		Edit Remove
Bittorrentracker	Not installed	Yes	Install		Edit Remove
Kanban	Not installed	No	Install		Edit Remove
DockerUI	Not installed	No	Install		Edit Remove
Lychee	Not installed	No	Install		Edit Remove
MariaDB	Not installed	No	Install		Edit Remove
Wiki	Not installed	Yes	Install		Edit Remove
MongoDB	Not installed	No	Install		Edit Remove
OwnCloud	Installing...			Wait please...	Uninstall
Rocket.chat	Not installed	No	Install		Edit Remove
WordPress	Not installed	No	Install		Edit Remove

New Container

Figure 3.2: The list of available services in the Cloudy interface.

address that is directly reachable from the IPs outside the user's home (no NAT in between, which is a key feature of CNs).

In some network node (in general remote) there is a video streaming source. This source generates a live video that can be received from any node in the network, the configuration of the source can be done following the available documentation[3]. PS-ng does not depend on the video source, in our experiments we used a simple USB webcam connected to a PC running PS-ng (details on how to generate this video are given in Sec. 3.3). Obviously, the video quality depends on the kind of camera and on the encoding of the video, but this is transparent to PS video distribution.

When a Cloudy node is turned on, it will connect to the Serf swarm and receive the list of the other running Cloudy instances. This is the default behavior of Cloudy as described in [3].

The users will connect with a browser to the Cloudy device, and, using the configuration panel of Cloudy, they

[3]See https://ans.disi.unitn.it/redmine/projects/peerstreamer-ng/wiki/Streaming_a_live_camera

can download the PS docker image. PS will appear among the list of available services, as Fig. 3.2 shows.

Once PS-ng is launched inside Cloudy, it will again use Serf to list the current list of available videos (possibly more than one). The users will connect through their web browser to the PS web interface directly from the Cloudy interface, and will interact with the PS web page running on Cloudy. They will be able to see the list of currently running videos and choose one to play in the browser or in the dedicated application (such as the widely used Video LAN Client open source program), as shown in Fig. 3.3.

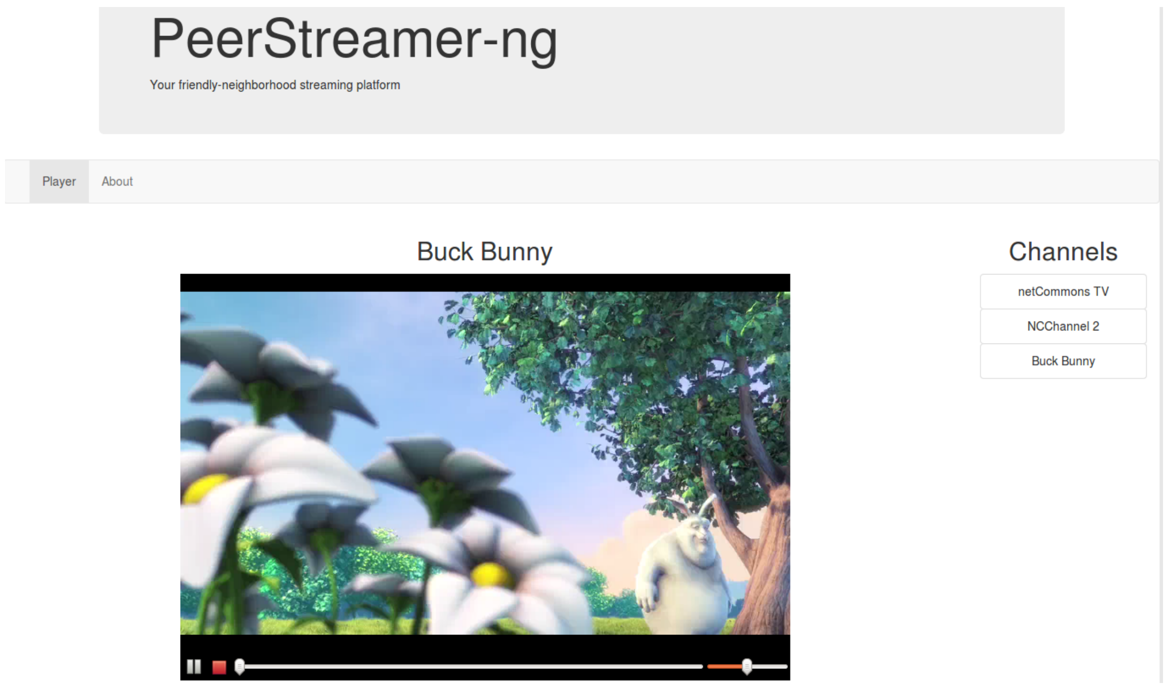


Figure 3.3: The web interface of PS-ng using Video LAN Client.

Even if this feature is not strictly required, more than one users can access the local Cloudy device, receive the list of channels and see a different video at the same time. This use case allows more people in the same house to watch the same video (or a different one) at the same time.

3.1.2 Internals of PS-ng

Fig. 3.4 reports the architecture of PS-ng, integrated into Cloudy, broken down into its functions. PS-ng has been designed with modularity in mind and it leverages the flexibility of its carefully chosen components. The core of the P2P streaming algorithms is based on the GRAPES library [26], following the architeture presented in [27, 28, 29].

Besides the Serf communication with the Serf network, the PS Cloudy containers utilize three different communication protocols:

- HTTP, for the user interface and more generally for controlling the platform;
- PS signalling and chunk protocols for P2P streaming;
- RTP, as last-mile widely supported streaming format.

We chose HTTP for controlling as it is a commonly used solution for providing user-friendly, highly portable GUI and it does not require any installation burden for the users. PS-ng implements a Representational State Transfer (ReST) interface upon the Mongoose HTTP library[4], which has been designed to be lightweight and highly portable, serving the GUI as HTML/Javascript document and exposing the controlling interface:

[4]https://www.cesanta.com/

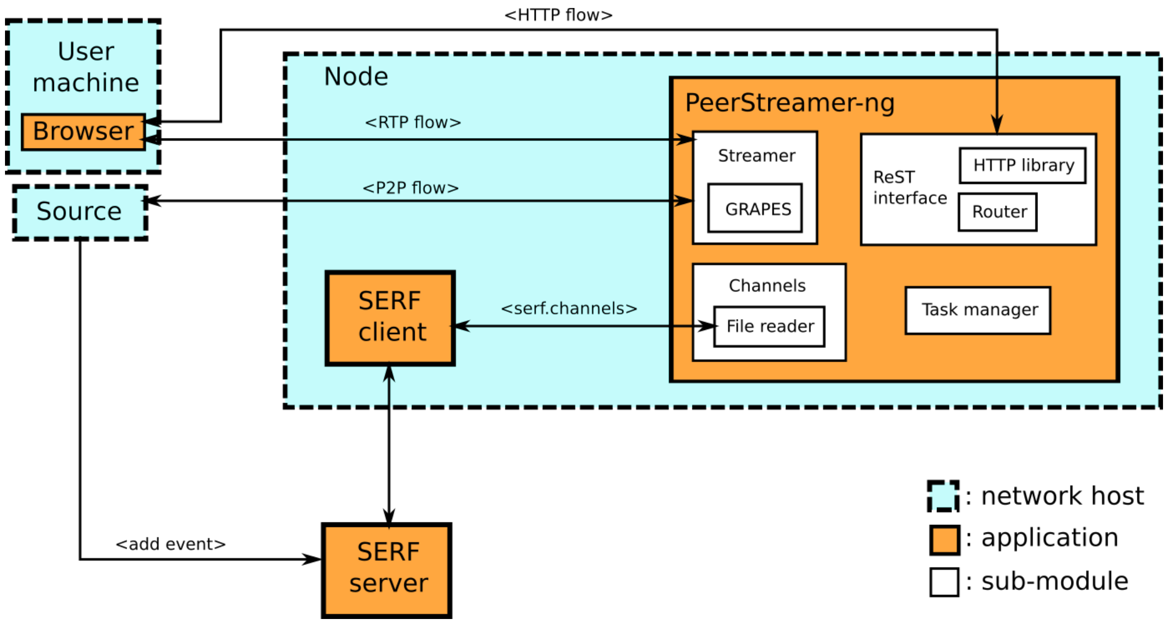


Figure 3.4: The architecture of PS-ng and the integration with Cloudy.

- GET /channels : JavaScript Object Notation (JSON) list of available channels
- POST /channels/<stream_id>?ipaddr=<source_ip>&port=<source_port>: create the streaming resource <stream_id> and launch the streaming instance; it returns a JSON describing the resource attributes with which the plugin player is initialized.
- UPDATE /channels/<stream_id>: heartbeat request, to be called frequently on <stream_id>

The *GET /channels* request is routed toward the ”Channels components”, which is in charge of keeping an up-to-date channel list from the SERF network. The *POST /channels/<stream_id>?ipaddr=<source_ip>&port=<source_port>* request is routed towards the streamer manager which, after checking the existence of a channel with the provided parameters, launches a new PS streaming instance using the PS library. Finally, we use a heartbeat-mechanism to keep each streaming instance alive instead of providing a ReST streaming DELETE as we consider users not reliable and we want to prevent streaming from being active indefinitely.

All the components building up PS-ng take advantage of the task manager, which orchestrates and manages the periodic application tasks of:

- Handling HTTP requests,
- Refreshing the channel list from the SERF client,
- Handling the PS P2P communication.

3.1.3 Integration with Serf

Cloudy uses the Serf protocol to distribute information about the available services in the Cloudy network, but it does not support access to the Serf primitives from inside each container. In practice, a Cloudy service can notify other Cloudy instances of its presence (informally, broadcasting a message that says that “a new PeerStreamer instance has been activated”), but there is no way for a PeerStreamer instance to actively control the content of the Serf message. In our case we need a way to specify which video is produced by the source, in order for other nodes to populate their channel list. Therefore, we need to implement a Serf library that interacts with Serf and pushes the necessary information to the gossiping service. Please see the specific documentation in the `psng-pyserf` repository at https://github.com/netCommonsEU/ to use this library.

3.2 The PS-ng web interface

PS-ng, as described in Fig. 3.4 now embeds a custom web-server which can be accessed by browsers. From the web page users can see the live video directly in their browsers.

The original design of PS-ng involved the use of Real-time Transport Protocol (RTP) to carry the video stream. This approach was well justified by the following reasons:

- RTP is a standard protocol, well documented and adopted by many applications,
- RTP is natively supported by Android,
- All the most popular browsers could use the VLC Media Player plug-in[5] to directly support RTP streams in the browser. Such plug-in requires the NPAPI libraries from the browser.

Unfortunately, during the development we understood that the browsers were phasing out the support for NPAPI libraries. Firefox in particular, which is our reference platform, removed the support since the version 52, released in March 2017[6]. Actually, the support is still present, so people can enable the VLC plug-in, and probably it will still be there for a long time since other very popular plug-ins (such as the Flash player) still need it. Nevertheless the procedure for enabling the VLC plug-in has become cumbersome, which discourages user adoption. When facing this situation we had to decide if we could accept the compromise of a working but user-unfriendly application, or it was better for the outcome of the project to spend more effort in re-implementing the web interface with a different technology. We decided to follow the second path since we favoured the impact on communities, even if this slightly delayed the developments on the video conference tool.

Currently, the most popular way of streaming a video file is using a VIDEO HTML5 tag, while there is not a single solution for live streaming. There are proprietary solutions (Flash, Silverlight, HTTP Live Streaming (HLS)) that are widely used but they require browser extensions similar to the VLC one, and are progressively being phased out in favour of standard ones. New standards are coming out, but none of them can today be considered a de-facto standard. Among them, based on the maturity of the specifications and the available open source implementations, we picked WebRTC[7]. Note that the WebRTC solution is very complex. There are few open source implementations of WebRTC outside of the browser space, everyone of them is conceived for a specific use-case and brings several dependencies. We had already analysed it in the first year of the project [3], but decided at the time for a straightforward adoption of RTP. On the other hand, WebRTC is supported by a number of browsers: Firefox, Chrome, Opera, Android and it was announced for Safari, whereas it is not supported by Microsoft Internet Explorer. It can support two-way communications, and thus serve as the basis for the conferencing system.

Our final choice was to support all the options:

- We decided to finalize an RTP version of the code that requires VLC, which can be used with the in-browser plugin or opening the URL with VLC (if the user has installed VLC).
- We supported the HTML VIDEO tag with Motion Picture Expert Group (MPEG) streaming, integrating the FFmpeg open source libraries[8], which are universally supported by all browsers, but not optimized for live video
- We realized a WebRTC version around the Janus open source platform[9].

Note that the use of RTP is required also in the WebRTC protocol, but the last link (from the web server to the browser) is completely different. Thus, we could partly re-use the code we realized in the VLC version in the

[5]See https://www.videolan.org/

[6]See https://www.mozilla.org/en-US/firefox/52.0/releasenotes/

[7]WebRTC is an open project supported by several major Internet players lead by Google https://webrtc.org/. The licence is very liberal and grants perpetual irrevocable use and modification rights. WebRTC communication protocols and modes are also standardized into RFC7874 https://tools.ietf.org/html/rfc7874 ensuring stability and openness.

[8]See https://ffmpeg.org/ for an overview, the general support of FFmpeg, and its licensing.

[9]See https://janus.conf.meetecho.com/ for an overview of the software r and the academic spin-off that developed it.

WebRTC design.

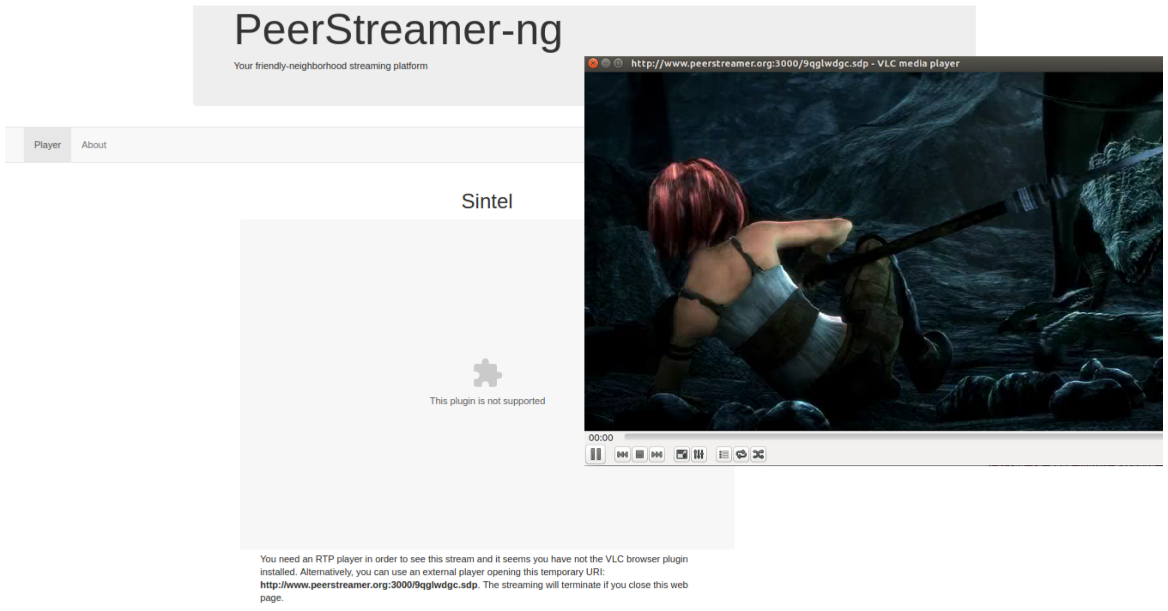
While we had the best results with the third option, we wanted to finalize the others in order to give more options to the users, and in the experimentation phase with the communities we will decide what is the option the communities will be more willing to support in the future.

In the next sections we will detail the solutions adopted, focusing just on the receiving side. The streamer will be better detailed in Sec. 3.3.

3.2.1 PS-ng with VLC plugin

The PS-ng integration with VLC is the one already shown in Fig. 3.3, and relies on the generation of an RTP stream from PS-ng web server to the browser directly. In general, this approach can face problems when the browser is behind a NAT. In our case this is imagined to happen in a local area network (i.e. the home Wi-Fi network) so it can be easily done without incurring in problems due to NAT or middleboxes.

In case the browser does not support the VLC plug-in the web page offers an URL that the user can click on in order to open the video stream with any external applications, such as VLC itself. This is shown in Fig. 3.5.


PeerStreamer-ng
Your friendly-neighborhood streaming platform
http://www.peerstreamer.org:3000/9qglwdgc.sdp - VLC media player
Player
About
Sintel
This plugin is not supported
00:00

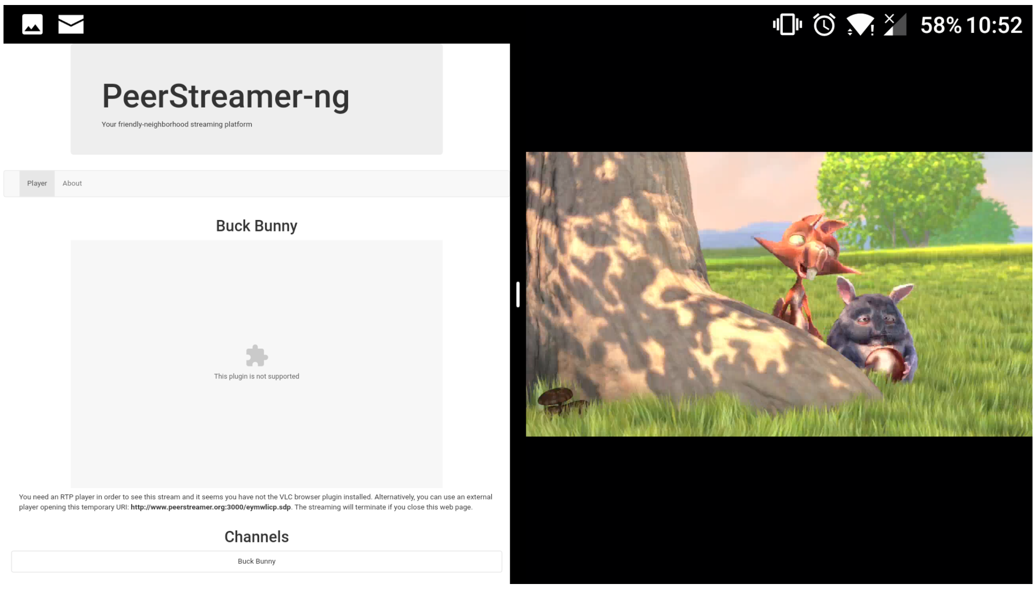
You need an RTP player in order to see this stream and it seems you have not the VLC browser plugin installed. Alternatively, you can use an external player opening this temporary URI: **http://www.peerstreamer.org:3000/9qglwdgc.sdp**. The streaming will terminate if you close this web page.

Figure 3.5: The web interface of PS-ng showing the URL to the video descriptor, and the VLC program playing the video (note the URL bar on top of the window frame).

The use of standard protocols (such as the RTP protocol) and open formats for the video and audio encoding makes it possible, in principle, to watch the PS-ng video also on mobile devices.

The current limitation resides on the fact that we need a separate application for playing RTP videos, as in the configuration depicted in Fig. 3.5. In our current browser-based design we require the browser to be active during the stream, in order to detect when the user is watching the video and stop streaming when they leave. When the browser is killed (or stopped) the Javascript contained in the web-page stops loading and the video is not streamed anymore. This is not an issue in a standard PC, but, since Android phones send the browser to sleep when a new app is launched, we have an additional difficulty in the current set-up to use PS-ng on mobile devices.

Fig. 3.6 depicts a video stream played on an Android device, note that there are two applications running, a browser and the VLC app in parallel. This configuration is not really usable but shows that the building blocks of PS-ng are compatible with the Android mobile environment even in the VLC version. The usability problem


58% 10:52
PeerStreamer-ng
Your friendly-neighborhood streaming platform
Player
About
Buck Bunny
This plugin is not supported
Channels
Buck Bunny

You need an RTP player in order to see this stream and it seems you have not the VLC browser plugin installed. Alternatively, you can use an external player opening this temporary URI: **http://www.peerstreamer.org:3000/eymwlicp.sdp**. The streaming will terminate if you close this web page.

Figure 3.6: An android phone playing a video with PS-ng. Note the necessary split-screen with the browser and the VLC mobile app running at the same time.

could be avoided with a custom mobile app that loads the web page and plays the stream, but also with the adoption of WebRTC, which is supported by mobile browsers.

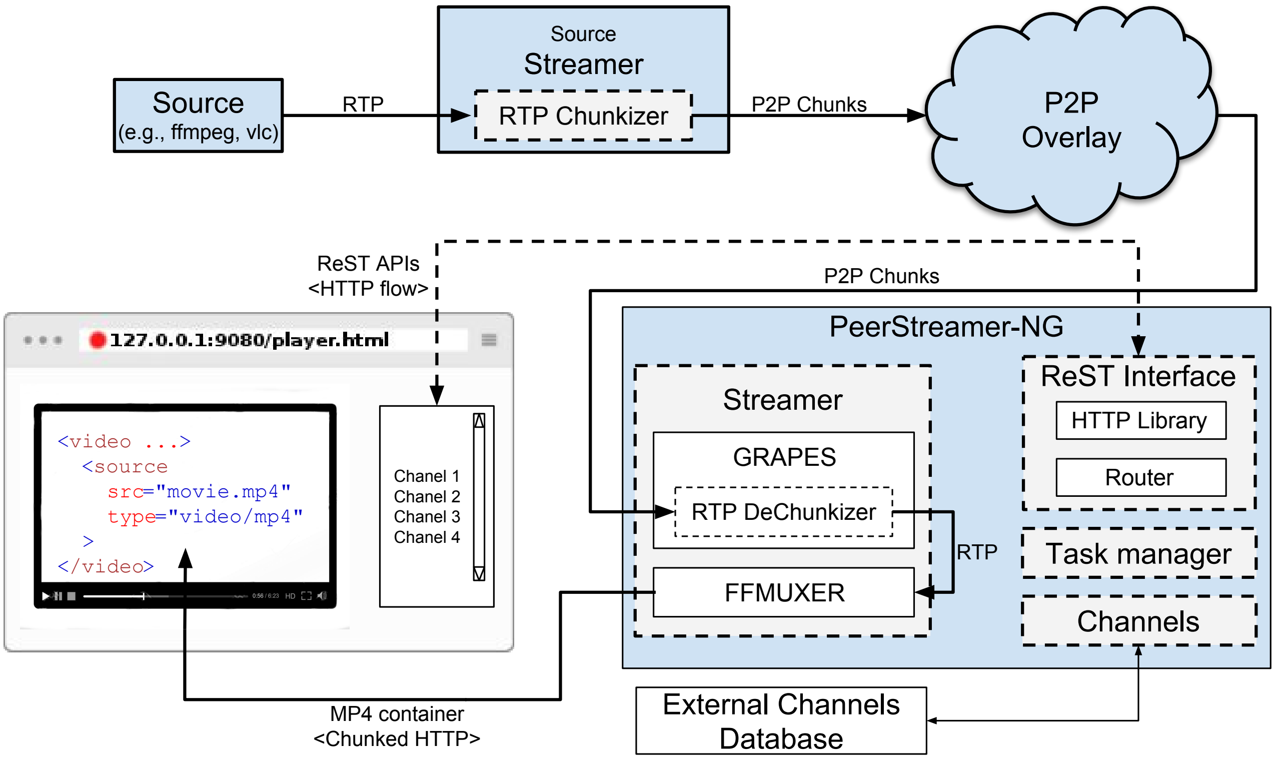
3.2.2 HTML5 Solution

The solution based on the HTML5 standard uses the open source FFmpeg library with the FFMUXER function, which multiplexes an RTP stream into an MPEG-4 (MP4) container to be served directly to the browser.

As Fig. 3.7 shows, the source produces a generic video with the RTP format, this choice is kept for compatibility with the previous solution and does not impact the architecture. The RTP stream is "chunkized" by the PeerStreamer at the source, which means it is split into content-independent transmission units, passed to the P2P network made of the PeerStreamer instances, and finally reaches the PeerStreamer on the user device. So far, the architecture is the same one used for the VLC solution.

Here it is "de-chunkized" and the original RTP stream is reconstructed and passed to the FFMUXER, which takes the RTP stream, synchronizes audio and video (which are actually two different streams), inserts it into an MP4 container which is finally transmitted to the client browser using HTTP chunks. In a bit more details what happens is that when the HTML5 VIDEO tag sends an HTTP request to PeerStreamer-NG, the Streamer instance responsible for the requested video saves a reference to the socket opened for handling the request and initializes the FFMUXER module. During the initialization process, the FFMUXER module reads the previously buffered RTP messages for retrieving the video and audio streams information (e.g., video and audio encoding, real bit-rate, frames per second, duration, etc.) which are then used to create an internal object used for generating the MP4 container that will be sent to the browser. In particular, the video and the audio streams are multiplexed into an IIS Smooth Streaming Video (ISMV) file type, that is an MP4 container with fragmented and multiplexed payload specially packaged for streaming. Finally, the FFMUXER is fed with the RTP messages extracted from the PeerStreamer chunks for creating the MP4 output container that is transmitted (as HTTP chunks) to the browser using the socket previously open by the client and saved by the Streamer instance. Note that, for efficiency reasons, the FFMUXER does not save any output video container on the file system of the host running PeerStreamer-NG. Instead, everything is handled in memory using the functionality provided by the FFMPEG dynamic buffers.

This is the major difference from the previous solution, in which the PS-ng web page using Javascript, detected



```
<video ...>
  <source
    src="movie.mp4"
    type="video/mp4"
  >
</video>
```

Figure 3.7: The architecture of PS-ng integrated with the FFMPEG library

the presence of the VLC plugin and directly started streaming the RTP messages to the plug-in. In this case instead, FFMUXER handles the RTP messages directly inside PeerStreamer-NG and generates the MP4 container that is served as HTTP chunks to the HTML5 VIDEO tag which is dynamically generated into the web page served to the client browser.

The main advantage of this solution is that every browser supports the playback of an MP4 video inside a VIDEO frame, and so, it is the most portable of all. The main drawback is instead that this video transport scheme is not imagined for live streaming, the browser expects to receive a video from a file (as it happens for video-on-demand platforms). Thus, it does expect the video to be interrupted (in case of temporary network outage), but it expects all the video to be completely transported, one frame after the other as a unicast HTTP connection runs on top of the Transmission Control Protocol (TCP), which is reliable and delivers information in order. Since instead in PeerStreamer architecture this is only the last leg of a P2P distribution, frames may arrive out-of-order and can also be lost, thus interacting negatively with the HTML5 VIDEO tag. Indeed, in the live video scenario instead there are chances that some video chunks may be lost, and so that the resulting stream could have holes and interruptions. In a live video stream this is normal and accepted, the video player should just skip the missing frame and move to the next one. In a unicast file-based video stream this cannot happen as TCP will stop and recover lost information. The outcome is that the browser video players, albeit tolerating it to a certain extent, behave in different and potentially erratic ways.

The solution to this problem is to offer a video to the browser that, even if it does not contain all the frames, at least has a streaming continuity so that at every instant the player has something to play, and does not enter a starvation condition. In a nutshell, we need to buffer the video. This is done in two places, before the FFMUXER the RTP messages are buffered by the PeerStreamer instance, so the FFMPEG library receives long enough RTP streams to recognize and synchronize the video and audio streams, and after the FFMUXER, the HTTP chunks are buffered before being served to the browser. With this approach, the video playback is stable, but a delay is necessarily introduced. We were able to produce a stable video playback in our laboratory deployment with a delay estimated between 5 and 10 seconds.

While this needs to be fine tuned in order to potentially optimize the dimension of the buffers, we observe that this delay is not compatible with a live streaming (let alone with an interactive application) and thus, we introduced the WebRTC solution documented in Sec. 3.2.3.

3.2.3 WebRTC Solution

WebRTC is a newly defined specification that

> *"provides browsers and mobile applications with Real-Time Communications (RTC) capabilities via simple APIs."*

As mentioned above, WebRTC is backed by Google, Mozilla and Opera, among others; it provides a specification and a code-base, and it is adopted by live streaming services such as Google Meet and Jitsi[10].

We already considered the adoption of PS-ng in the OpenWRT version of PeerStreamer [3], and, for that specific application it resulted too complex to implement, especially because initially the target platform at the time was a wireless router with little computational resources and disk space. With the Cloudy integration we expect to be able to run PS-ng in a more powerful device and we explored ways to use the WebRTC code base offered by some open source projects. Contrarily to the HTML5 solution in fact, WebRTC carries a number of complex features that can not be simply re-implemented in PS-ng. For instance, a mandatory support for tunneling protocols (such as STUN or Interactive Connectivity Establishment (ICE)), Datagram Transport Layer Security (DTLS) and a complex state machine and API. Since we want to interface with a standard browser, we can not strip down the requested functionalities to the minimum, as the browser may request them anyway.

To cope with this complexity we decided to use the Janus platform, that is an open source general purpose WebRTC gateway. Janus is a general purpose gateway, described in Janus website as:

> *"it implements the means to set up a WebRTC media communication with a browser, exchanging JSON messages with it, and relaying RTP/RTCP and messages between browsers and the server-side application logic they are attached to. Any specific feature/application needs to be implemented in server side plugins, that browsers can then contact via the gateway to take advantage of the functionality they provide."*

In practice Janus mediates between a source of RTP video (as in our case, PS-ng) and a browser that needs to access it. It exposes APIs and interfaces so that the commands that the browser sends through a Javascript page will be passed from Janus to a specific plug-in implementing the needed functions (such as stopping/pausing the video).

Fig. 3.8 describes the architecture of the WebRTC-based solution. The source is not changed and thus it is not represented, the only difference with the previous scheme, is that WebRTC does mandates a specific video encoding, so the video source will have to encode the video in the open VP8 standard. PS-ng, whose internals are hidden in the figure for simplicity, will communicate with Janus using a Janus plug-in. This plug-in is called P2STR and it uses the Janus plug-in APIs. Janus makes available to the plug-ins a HTTP transport layer, which the plug-in can leverage to provide a ReST interface. This ReST interface accepts commands from PS-ng, in order to control the plug-in, and from the browser in order to establish WebRTC communications. The web page that is displayed to the user contains Javascript code that interacts with the plug-in, using REST. Therefore, the code developed to interact with Janus is divided in two parts:

- a C language plug-in, running server side, using Janus extension (`p2str.c`);
- a Javascript code, running on client side (`p2str.js`).

[10] Jitsi is one of the few independent and free conferencing tools available https://meet.jit.si.

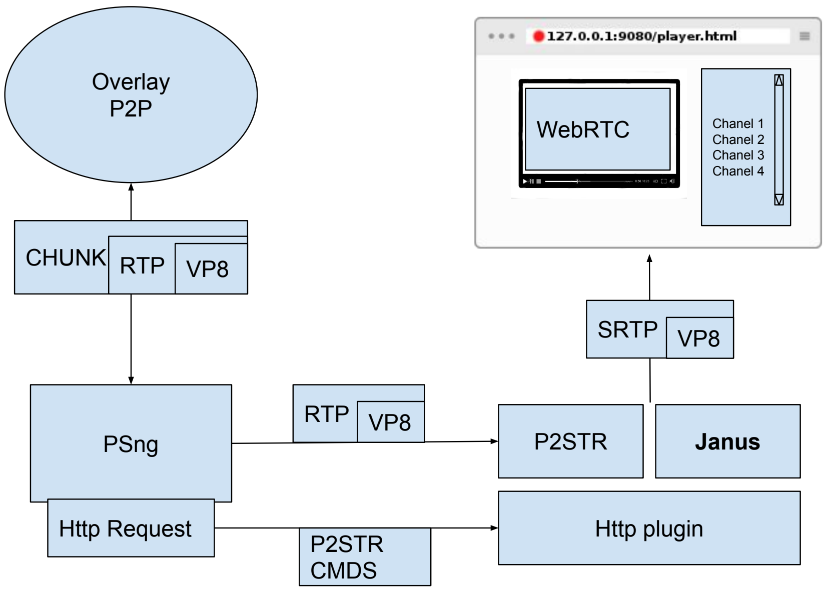


Figure 3.8: The architecture of PS-ng integrated with the Janus library

The role of Janus is to route requests back and forth, to receive the RTP stream that PS-ng generates and to embed it into a WebRTC stream to the browser. While Sec. 3.2.3.1 describes the interactions in detail, in a nutshell, what happens when a video is displayed is the following:

- The user loads a web-page with a list of channels available, exactly as in the previous cases;
- PS-ng receives REST calls from the browser to start a video, so it:
  - sends a REST call to p2str.c to create a new RTP2WebRTC tunnel,
  - starts the RTP stream to Janus/P2STR;
- Janus routes it back to the browser via WebRTC.

The main advantage of this configuration is that WebRTC is imagined for live interaction, thus, the delay we measured in our lab set-up is lower than a second. Another advantage is that Janus is already prepared to acquire the video from the web browser (i.e., a webcam) and push it back to PS-ng. This is key in the video conference scenario, but also to ease the realization of a video source. Moreover, WebRTC can be used also in mobile devices that may not support VLC. The drawback of this solution is the burden of complexity that Janus introduces, a separate daemon running, a set of complex interactions between the two projects (Janus and Ps-ng).

3.2.3.1 Details of the ReST Interface

This section details the interfaces and the workflow needed to set-up a video stream.

`p2str.js` implements the following interface:

1. createJanus();
2. startStream(streamID);

The first one instantiates a WebRTC peer capable of interacting with Janus, the JS end-point of the WebRTC session. The startStream function is responsible of requesting Janus for a particular streaming instance and visualize the video in the HTML related component.

`p2str.c` implements the following interface:

1. `{"request": "create", "type": "rtp"}`: to create a streaming mountpoint for a given input RTP stream;
2. `{"request": "destroy", "id": <streamID>}`: to destroy a given streaming mountpoint;
3. `{"request": "watch", "id": <streamID>}`: to join an RTP stream through a valid WebRTC connection;
4. `{"request": "start", "id": <streamID>}`: to start streaming an incoming RTP stream through a valid WebRTC connection;
5. `{"request": "stop", "id": <streamID>}`: to stop streaming an incoming RTP stream through a valid WebRTC connection;

The first two calls, for the management of the mountpoint, are implemented in PS-ng which is responsible of setting up and down the RTP streams. The latter three p2str calls are instead demanded to the web client interface which, through the browser API and Janus Javascript library, can create valid WebRTC connections.

The typical workflow is the following one:

1. the web client requests a stream by calling `POST /channels/<stream_id>` with payload `ipaddr=<sourceip>&port=<sourceport>`
2. PeerStreamer-ng then:
   a) instantiates a P2P streamer for the P2P source: `<sourceip>,<sourceport>`;
   b) set the output of that streamer toward a convenient set of RTP ports: `<port_audio>,<port_video>`;
   c) calls the P2STR request
      `http://localhost:8088/janus/&lt;mgmt_session&gt;/&lt;mgmt_handle>`
      with the following data structure:

```
{"transaction": "random", "janus": "message",
 "body": {"request": "create", "type": "rtp","audio": true,
          "audioport": <port_audio>, "audiopt": 111,
          "audiortpmap": "opus/48000/2","video": true,
          "videoport": <port_video>, "videopt": 100,
          "videortpmap": "VP8/90000"}
}
```

      and it collects the JSON answer containing the streamID;
   d) returns the newly created channel description inclusive of the streamID to the web client.
3. the web client can finally call startStream(streamID) that:
   a) calls the 'watch' method of P2STR Janus extension;
   b) create a video component and redirect the incoming media to it.

3.3 Support for Bi-Directional streams in PeerStreamer

So far, we assumed the presence of a video source that streams the live video to the P2P network. This can be easily realized with several open source programs, like the already mentioned VLC. VLC can acquire the video stream directly from a webcam (or a file) and produce an RTP stream encoded with the right parameters needed by all the solutions mentioned so far. This can be done with a script, and should be possible with VLC graphical interface. VLC will not only generate the video but also a descriptor file (based on the Session Description Protocol (SDP) syntax, as described later on), which is distributed in the Cloudy network using Serf. This configuration assumes that the person running the video source will execute a few commands to start the video[11]. While this is feasible in general for a P2P-TV application, in which there is one person running a source and a multitude of other people only receiving it, it is not practical for a two-way communication between people in a conference, and must be managed in some other way.

Thus, essentially three steps are needed to move from a P2P-TV towards a P2P videoconference system:

1. The automation of the video acquisition,
2. The creation and management of the video descriptor file, and
3. The changes in the user interface.

We analyse these three steps in detail in the next sections.

3.3.1 Acquiring the Video with WebRTC

The advantage of relying on Janus is that, being a media gateway, it accepts the video streams in both directions. We can modify the architecture of PS-ng and the P2STR plug-in in order not only to receive the video, but also to acquire it from a camera and forward it to Janus. Janus will then convert it to a simple RTP stream, that we can then distribute with PeerStreamer.

In order to realize this task, `p2str.js` implements the following interface:

1. createJanus();
2. startSource(streamID);

The first one instantiates a WebRTC peer capable of interacting with the running Janus instance of PeerStreamer-ng, as in the previous case. The startSource function is responsible of sending Janus content for a particular streamID (identifying a unique position).

`p2str.c` Janus extension implements the following interface:

1. `{"request": "create", "type": "webrtc2rtp"}`: to create a valid bridge instance between WebRTC and RTP ports (it returns the streamID);
2. `{"request": "destroy", "id": <streamID>}`: to destroy the bridge.

Both calls are intended to be called by PeerStreamer-ng, which is also the one managing the RTP ports to be open. The workflow is then following one:

1. the web client requests to stream by calling `POST /mysources/<channel_name>`
2. PeerStreamer-ng:
    a) instantiates a P2P streamer source listening to the selected RTP ports;
    b) calls the P2STR request to create a WebRTC2RTP bridge;
    c) returns the newly created channel description inclusive of the streamID to the web client.
3. the web client can finally call startSource(streamID) which
    a) captures the webcam video;
    b) send the video to the Janus WebRTC instance.

This part of the project is still under development. While we have some running code, it still needs to be tested, we will stabilize it in the period M24-M36 and report on it in D3.5.

3.3.2 Distributing SDP Files

A video player, to be able to play a video needs to know some of the parameters needed to decode the video itself. These parameters, when opening a file, are generally embedded in the headers of the video container,

[11]Scripts used in our tests can be found in the PS documentation: https://ans.disi.unitn.it/redmine/projects/peerstreamer-ng/wiki/Streaming_a_live_camera

but when the video is live streamed, there must be a way for the player to acquire these details with an ‘out of band’ channel.

One standard way of describing multimedia contents is the SDP format, a standardized format for describing video sessions [30] that several video players already support. SDP is human readable, and a typical descriptor is made of a subset of the fields reported in Listing 3.1.

```
Session description
v= (protocol version)
o= (originator and session identifier)
s= (session name)
i=* (session information)
u=* (URI of description)
e=* (email address)
p=* (phone number)
c=* (connection information - not required if included in all media)
b=* (zero or more bandwidth information lines)

One or more time descriptions ('t=' and 'r=' lines; see below)
z=* (time zone adjustments)
k=* (encryption key)
a=* (zero or more session attribute lines)

Zero or more media descriptions
Time description
t= (time the session is active)
r=* (zero or more repeat times)

Media description, if present
m= (media name and transport address)
i=* (media title)
c=* (connection information - optional if included at session level)
b=* (zero or more bandwidth information lines)
k=* (encryption key)
a=* (zero or more media attribute lines)
```

Listing 3.1: The available fields in the SDP format.

In the Cloudy integration we use Serf to distribute SDP files. For each video source, the group of peers that exchange the video chunks is different (the group of peers is called an *overlay* in the P2P terminology), and each peer collects the SDP file from Serf.

In the case of video conference, before deciding how to distribute SDPs there is one architectural choice to be made: all the videos can be transferred on the same overlay, or one overlay per video can be used. Which one is the best solution is still an open research challenge: on the one hand, overlays (as we observe in Sec. 3.4) tend to be optimized for one stream, so that in a video conference we expect to have better performances if every source uses a different overlay. On the other hand, maintaining an overlay requires some traffic overhead due to control packets, and thus, having one overlay only could be more efficient in terms of network usage.

For the time being, we decided to opt for one overlay per stream, so that we can use Serf to distribute the SDP in a similar way to the single source case. Nevertheless, we are designing also the introduction of SDP files in the signalling mechanisms of PeerStreamer in order to support more than one stream per overlay. The SDP description was added to the topology description messages in PS. Once a peer enters a new overlay, it will poll other peers with topology messages. These messages are used to ask and receive a set of IP addresses of available peers in the overlay, in order to build a partial view of the PS overlay topology, sufficient to participate to the video chunk distribution. We added to the topology messages an optional list of the available session identifiers, with a pointer to a known peer that is currently distributing it. Each session identifier represents one video stream and consequently one SDP descriptor. Once the new peer has collected a list of identifiers, it will collect the SDP directly from the corresponding peers.

With this mechanism, the workflow of a potential P2P video conference is the following one:

1. The user enables PS in the Cloudy interface;
2. The PS web interface collects the information about the available video streaming sessions. Among them, there will be typical one-to-many streams (like P2P-TV) but also the currently ongoing group conferences;
3. The user will choose one of the ongoing conferences, then PS will start collecting information on the peers that are present in the corresponding overlay;
4. Together with the information on the peer PS collects also information about the video sources (probably one per peer). This information is not shown in the channel list, but is managed internally by PS;
5. PS will start showing the available video sources in the PS web interface.

This solution is still under development, but it has a lower priority than the Serf-based one, which is easier to achieve in the short term.

3.3.3 The modified Web Interface

The last step needed to turn PS-ng into a conferencing application is to modify the web interface in order to accommodate all the video sources. For the initial roll-out of the platform we plan to support 2-5 sources, and possibly more receivers, and so, the interface should not require massive modifications.

After that number of users, the scalability of WebRTC itself is to be explored. As a reference, Google Hangout, that runs on a proprietary centralized platform limits the number of participants to 25 (so as Skype does[12]), but contemporarily shows only up to 10 videos[13]. This is mostly due to the limitations in terms of CPU and memory usage of the browser, as Google has virtually limitless capacity on the back-end server. In our case the back-end is the Cloudy device, so we can't expect our conference call to support such a high number of users, even if the solution is distributed.

The modification to the Web interface will be documented in D3.5 at M36.

3.4 New Research results on P2P Video streaming on CN

Two new publications have been produced in the field of P2P video streaming optimization for CNs:

- Leonardo Maccari, Nicolò Facchi, Luca Baldesi, Renato Lo Cigno: "Optimized P2P Streaming for Wireless distributed Networks", Journal of Pervasive and Mobile Computing, Volume 42, pages 335-350, Dec. 2017.
- Luca Baldesi, Leonardo Maccari, Renato Lo Cigno: "On the Use of Eigenvector Centrality for Cooperative Streaming", IEEE Communications Letters, Volume 21, Issue:9 , pages 1953–1956, Sept. 2017.

The two proposals are complementary and are under implementation in an experimental branch of the PS-ng software. In this section we briefly summarize their content, that can be found in the above publications [15, 31].

[12]See https://blogs.skype.com/news/2016/02/18/announcing-group-video-calling-for-mobile-phones-and-tablets/
[13]See https://gsuiteupdates.googleblog.com/2016/03/connect-with-more-people-using-google.html

3.4.1 Optimized P2P Streaming for Wireless distributed Networks

The key factor hampering P2P development, specially for video distribution, has been the difficulty to realize P2P overlays optimized from the point of view of the Internet Service Providers (ISPs), mostly due to lack of information on the physical network. The reasons of this difficulty are threefold. First, the TCP/IP protocol suite naturally separate the application protocols from the network protocols, making it very difficult to perform any kind of cross-layer optimization. Second, ISPs are very reluctant to discover details (routes, costs, etc.) of their networks. Third, the business and economic sustainability of the P2P paradigm was yet not explored and understood, and in any case it clashes with the interests of well established ISP, which have no interest in supporting it.

In mesh networks, the underlay is normally known, since the routing protocols export it to each node (as long as a link-state routing protocol is used), which removes one of the technical barriers that blocked the deployment of P2P video streaming on the Internet.

In [15], we proposed a cross-layer optimization scheme to perform *live* video streaming (i.e., with a strict deadline on the arrival delay) in mesh networks. The optimization minimizes the impact of the streaming overlay on the underlay network exploiting information on the topology and the routing of the underlay. The optimization is based on the concept of *centrality*, taking into account the centrality of peers in the underlay graph, the optimized overlay topology greatly improves the efficiency of the video distribution and maintains high performance.

More specifically, the key paper contributions are as follows:

- We formalize an optimization problem that, given the underlay of the P2P network and the peers in the overlay, finds the overlay topology whose cost on the underlay is minimum. We define the cost of the overlay as a combined metric composed by the load and the fairness imposed by the overlay on the underlay;
- We prove that the optimal strategy is NP-complete by reducing it to a quadratic knapsack problem;
- We propose two techniques for relaxing the optimal strategy, which relies on the concept of betweenness centrality of the nodes of the underlay. The relaxed strategies are applicable to any wireless mesh network, as long as they use a link-state routing protocol;
- We further improve the performance of the relaxation strategies by introducing a *neighborhood pruning* heuristic, which exploits characteristics often found in real network scenarios;
- We evaluate the proposed relaxation strategies and pruning heuristics through extensive simulations that compute the best overlay, according to each proposed technique, on synthetic network topologies. Simulation results show that the proposed relaxations and heuristics are reasonably close to the optimal solution and largely outperform random overlay building strategies.

The paper builds on a previous work published in Y1 of netCommons, in which we first formalized the optimization problem and then introduced the relaxation strategy [14]. This paper extends these results with further optimizations to make the proposal even more efficient in overlays with a realistic topology, i.e. comprising many leaf-nodes in the network graph.

3.4.2 On the Use of Eigenvector Centrality for Cooperative Streaming

As we said in Sec. 3.4.1 the typical assumption done in P2P video streaming is that the underlying physical network is unknown. In such conditions, the best choice of the overlay graph is a purely random graph, for several reasons we do not treat here. Once the assumption falls, and the overlay is built in a way that matches the underlay, a new problem emerges, that is, how to propagate the video chunks in an overlay that is not anymore "random" but has a definite structure, and thus, a purely random propagation may be suboptimal.

Consider a generic network represented by a graph $G(\mathcal{V}, \mathcal{E})$; $v_i \in \mathcal{V}$ is the generic node, and $e_{ij} \in \mathcal{E}$ means there is a connection between v_i and v_j. The streaming source is a network node that generates a packetized video with a rate of one packet every τ s and sends one copy of each packet to a set of nodes chosen from its neighborhood. Each node v_i stores the received packets in a temporary buffer, and every τ_i s it picks another node v_j among its neighbors and a packet stored in the buffer and forward the latter to the former with a unicast transmission. This way packets percolate from the source to all the nodes in the network. We assume that neighbor nodes share the composition of their buffer one another, so that when a node receives a new packet it knows for which of its neighbors this packet is *useful* (the neighbor does not own it). This assumption is realistic because the buffer maps can be easily piggybacked on content.

The problem we tackled in [31] is twofold: i) define the values of τ_i per each node; ii) define a strategy for the choice of the target neighbor. The constraints we posed are that at steady state every node receives the entire content, and at the same time the overall resources used to achieve this goal are minimal.

In [31] we showed that this can be obtained exploiting the *eigenvector centrality* of the normalized adjacency matrix describing G and that there is a performance improvement in terms of reduced packet loss and delivery delay compared to other solutions currently in use. Finally we highlighted that the eigenvector centrality can be efficiently estimated with distributed algorithms, so that the implementation of our solution in real systems is feasible and can scale up to systems made of hundreds of nodes.

4 Update on CommonTasker

CommonTasker has been conceived from the beginning as an Android mobile app that will serve the needs of the Sarantaporo.gr community. The original idea about the application was that it could feature three discrete components, enabling users to crowdsource tasks, exchange information (e.g., question and answers capabilities), and share equipment in line with the sharing economy paradigm, respectively. However, from the very beginning, the work on it has followed the participatory design approach and has relied on interactions of various forms with the local community in shaping its scope and features.

These interactions escalated during a workshop in the Sarantaporo area in November 2016. The workshop participants included community members but also actors from the broader Greek community of commons and the private sector documented in D3.1 [1]. The fruitful discussions in that workshop and the reaction of the community to different topics presented by the workshop participants, motivated the focusing of the application towards the data crowdsourcing functionality. The idea that attracted most the interest of local farmers was to turn the application into an online assistant, which would help them record and organize their everyday activities in their farms but also share them, where appropriate, with other community members and external actors.

Particularly inspiring to this end was a presentation by a Greek SME called GAIA Epicheirein (hereafter referred to as GAIA) on smart farming. Smart farming is an umbrella term for practices that leverage the modern ICT technology for optimizing farming tasks. It can generate a number of benefits for farmers such as reducing the production cost and ameliorating the quality and quantity of products, saving environmental resources and minimizing the risk for the farmers. Almost two years ago, GAIA launched the provision of experimental smart farming services with the aim to provide them commercially country-wide by 2019. Their platform, called GAIA Sense, is built in the cloud and combines four different types of data: satellite images openly available by the European Space Agency (ESA) for research purposes; data on environmental conditions (temperature, humidity, air pressure) and ground sanity (acidity, penetration of water, ingredients etc) automatically collected by custom IoT telemetric devices called GAIATrons; and detailed data that needs to be provided manually by the farmers through a web application about their daily activities in the farm (e.g., time and duration of irrigation, time of field fertilization, chemical composition and brand of fertilizers used).

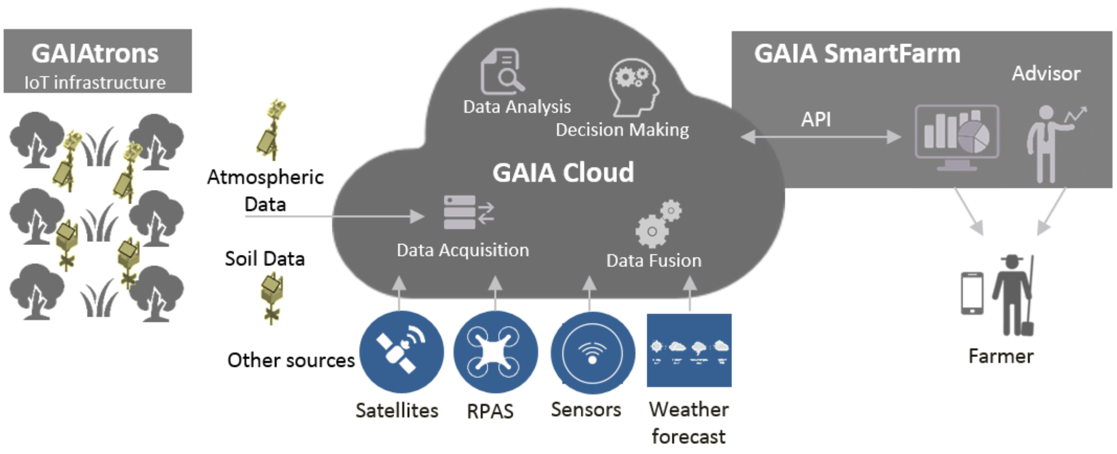


Figure 4.1: GAIA SENSE. GAIA Infrastructure for supporting smart farming services.

These data are then processed in the cloud through smart farming analytics to generate knowledge that is fed back to farmers as visualizations, best practice advice and recommendations by experienced agronomists, as sketched out in Fig. 4.1.

4.1 The reshaped CommonTasker and a longer-term case study for its use

After the November 2016 workshop and through several interactions with community members that followed and are documented in D3.3 [2], the CommonTasker's scope has evolved towards a multipurpose tool that will become the de facto online assistant of local farmers in their everyday farming activities. This implies capabilities such as recording data on-the-go about all carried out farming tasks and practices about them, scheduling activities, extracting and exporting information out of these records as files, and sharing data and tips with colleagues and friends. We detail the design requirements that arose for CommonTasker and how these have been addressed during the development phase in Sec. 4.2. As a result of this iteration on its scope, CommonTasker is turned to an app that serves professional needs of the local community, and eventually, the local economy as such. This way, it multiplies the value of the Sarantaporo.gr CN and its impact on the community it serves.

Notably, the interactions that followed the November 2016 workshop hinted at an even more ambitious and impactful role of CommonTasker for the Sarantaporo.gr CN in the longer term. The collection of data from farmers has emerged as a bottleneck for the smart farming services GAIA intends to offer. Contrary to the other types of data that become available either independently or automatically, this data requires the active human involvement. However, the majority of the agricultural population appear reluctant to use Web forms to log such data, although most of them keep their manual logs in various ways.

Briefly, CommonTasker could become the vehicle for a synergy of the Sarantaporo.gr CN with GAIA along the following lines:

- The services of GAIA would be offered over the CN in the area of Sarantaporo rather than over 2.5G networks as the case is in other areas in Greece. This would leverage further the value of the CN for the local community and serve as a strong incentive for its participation in the CN;
- The CommonTasker app would serve the collection of data from the population in the Sarantaporo area over the CN. The application would be promoted as a local app and would account for the data privacy concerns and preferences of the local community. Hence, it would have far better chances to overcome what is currently the bottleneck for the implementation of GAIA's smart farming services;
- GAIA would also contribute in kind to the Sarantaporo.gr CN maintenance and evolution (i.e., through sponsoring nodes).

Such a synergy could emerge as a major contribution to the Sarantaporo.gr CN's pursuit of economic sustainability and serve as a guide for its longer-term strategy. Nevertheless, the actual implementation of such a synergy is a vision that goes far beyond the scope of Task 3.4. The economic implications, e.g., business model, of such a synergy are further explored theoretically in the context of Task 2.2 of the project.

4.2 Application requirements

The reorientation of the CommonTasker app, as described in Chapter 4, implies that the app should combine calendar/log capabilities and gamification functionality, the former for enabling the collection of data and the latter for motivating it. For the same purpose, that is making the application more attractive and promoting its frequent use, the app should support basic social networking features, such as the possibility to define "friends", share photos, chat with them, add comments/likes to content they upload etc. Equally important is the dimension of data management and privacy, which could be a sensitive issue for the local community (see Sec. 4.2.5). The application should give farmers some control over which data they share and when. This capability will be crucial if Sarantaporo.gr would undertake the role of proxy/intermediate for managing the availability of data from community members to third-party service providers such as GAIA.

In what follows, we codify the requirements for the CommonTasker app, as these have been emerging through the participatory design process outlined in D3.1 [1] and formalized further in D3.3 [2]. We categorize them along the main functionalities the app features. It should be noted that this list of requirements is a dynamic one. It gets updated over time in line with feedback obtained from all the stakeholders involved in the Sarantaporo.gr CN. What is listed below reflects its status by M24 of the project, i.e., in an advanced stage of app development. Hence, it can be considered quite stable, although additions/modifications to it should not be excluded throughout its development and pilot deployment phases.

4.2.1 Data collection and logging requirements (RDC)

The application should:

- (RDC1) provide an easy way for a user to log information about everyday farming activities such as irrigation, top dressing, spraying, tillage;
- (RDC2) let the user detail custom information about each activity: for example, the duration of irrigation or the amount of fertilizer used;
- (RDC3) support the collection and uploading of photos which are collected from the field on-the-go;
- (RDC4) support the offline collection of data, when network connectivity is problematic, and their uploading when network connectivity resumes;
- (RDC5) help the user easily search and browse through past entries as well as organize them according to their date and scope;
- (RDC6) provide users with ways to export all or properly filtered subsets of posts as a printable document such as a .pdf file;
- (RDC7) let users schedule and get reminders about (periodically repeating) tasks such as maintenance tasks related to their farming equipment.

4.2.2 Information-feeding requirements (RI)

Interviewed members of the community insisted that the app should also ease the availability of important information pertaining to farming such as weather updates and relevant alerts from institutions that issue warnings/recommendations about threats pertaining to certain cultivations. Hence, CommonTasker should:

- (RI1) provide up-to-date prognostic information about the weather in the area;
- (RI2) capture and broadcast to the farmers alerts and warnings about threats to certain cultivations. These are highly seasonal and are issued by institutions such as the Ministry of Agriculture and public Regional Centers of Plant Protection and Qualitative Control.

4.2.3 Gamification requirements (RG)

The gamification component should leverage well-known practices for encouraging a contest between individual users but also groups of users. Therefore, it should:

- (RG1) keep a log of information posts by individual users and compensate them with points for their contributions;
- (RG2) award them with different points, depending on the frequency, volume, and completeness of the provided information;
- (RG3) rank users according to their contributions and visualize these ranks in smart eye-capturing ways;

- (RG4) support filtering capabilities in ranking users, i.e., present the rank of a particular user among apps users belonging to a specific age-group or coming from her village only;
- (RG5) support aggregation capabilities in that the scores of users can be aggregated on age group or village basis, and the ranks are over age groups villages, respectively;
- (RG6) couple the score keeping for the app users with offers from either the Sarantaporo.gr (and, in the longer term, by third-party service providers such as GAIA), to motivate the sustained participation of users in the app.

4.2.4 Social networking requirements (RS

The social networking layer aims to make the application more attractive and align it with current trends in application design and user expectations. The CommonTasker social component should:

- (RS1) let users exchange photographs, not necessarily related to their farming activities;
- (RS2) give users the opportunity to define friends in the app, follow other users, and exchange likes/comments for content they upload;
- (RS3) support chat/instant messaging between users, in ways that are familiar to them through mainstream social apps.

4.2.5 Data management and privacy control requirements (RP)

One of the requirements from the app, in line with the values and principles embedded in the operation of CNs, is to let users exercise control over their data. In the longer term, there could be three levels of data storage in the system: a local one at the user premises, storing their own data; a server at the CN premises, storing data from all CN users; and, an external cloud, either by a dedicated cloud service provider or by a service provider (e.g., GAIA), which could aggregate data from more users, beyond those from the Sarantaporo area. To this end, the application should

- (RP1) let users specify which posts are for private use and which ones can be shared;
- (RP2) have an alert sign on posting pages reminding them the setting they have chosen for the specific post;
- (RP3) implement reciprocity in the sharing of data, i.e., a user would be able to see and benefit from postings from other users as far as she shares her data as well.

4.2.6 User profiling and authentication requirements (RPR)

The user profiles created by the application should

- (RPR1) include information about the number and location of farms and what is cultivated in each;
- (RPR2) provide them with the interfaces to determine their privacy settings;
- (RPR3) request from users the minimal personal information needed for the app, i.e., they could log the age group of the user rather than their precise age;
- (RPR4) let them authenticate through multiple identifiers, including their email, mobile phone number, and accounts in other social media.

4.2.7 User interface & graphic design (RU)

Finally, it is worth recalling that the target group for the application includes people with very elementary or near zero familiarity with the use of mobile apps. Except for the training that has to be made to these people, it is important to maintain the user interface as simple as possible. At the same time, it would help to customize it in ways that the users feel more comfortable with, i.e., using local pictures as background and icons with clear semantics. In summary, the app should:

- (RU1) create a simple, straight-forward user experience, building as much as possible on concepts, visualizations, icons, interfaces users are familiar with;
- (RU2) provide a user with a graphical interface hinting at the use of a calendar;
- (RU3) minimize the need for textual inputs from user by using drop-down menus, picker buttons and default choices as aggressively as possible;
- (RU4) adopt guidelines with respect to fonts and font sizes that account for the needs of elderly people.

4.3 Application architecture and design

The functionality of the app is split between two main parts:

- The mobile client running on a smart mobile device (Mobile UI);
- The application back-end, currently implemented over the Firebase web platform.

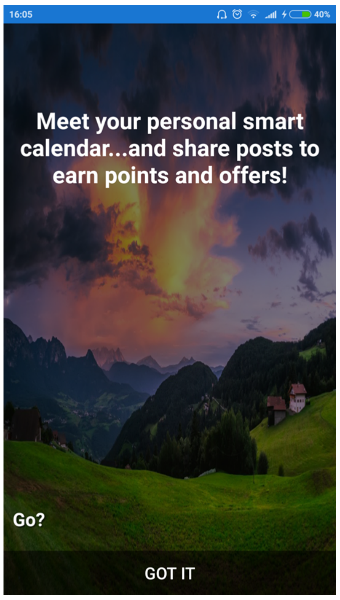
These two parts are further detailed in the following subsections.

4.3.1 The mobile UI

It is implemented over the Android OS, with the use of Android studio.

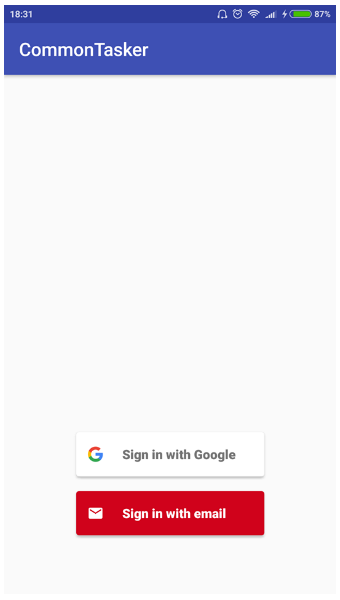
It has nine main components detailed in the following nine divisions of this subsection.

(a) Animated splash page of the mobile app: entry frame

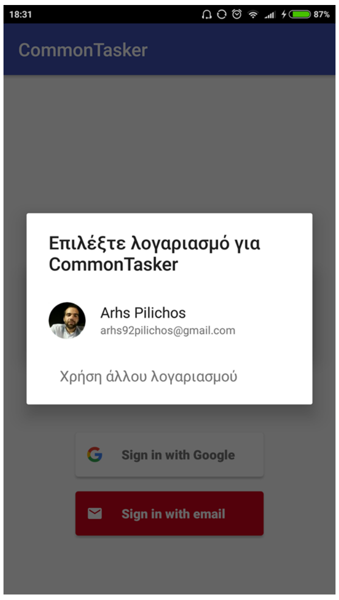

16:05
40%
Meet your personal smart calendar...and share posts to earn points and offers!
Go?
GOT IT

(b) Animated splash page of the mobile app: second frame

Figure 4.2: Splash page views



(a) Login page of the mobile app



(b) Login page with Google credentials authentication

Figure 4.3: Login page views

4.3.1.1 The splash page

Its main objective is to communicate the scope of the application in a eye-capturing way. It consists of two frames presented sequentially to the user. The first one combines a background picture of the area with a welcome message. The second one points to farming with a message summarizing the application functionality and an exit button that leads to the app homepage, as shown in Fig. 4.2.

4.3.1.2 The Login Page

This page is going to focus on the Authentication service provided by Firebase using Google Sign-In and Firebase Authentication with Realtime Database. We can click on the Google Sign-in button to trigger the Google pop up. If signing in is successful, the developer console will log in and confirm our user in the apps the process of identifying an individual takes a primary unique token, as shown in Fig. 4.3.

4.3.1.3 The user home page

After the splash page, the mobile app user lands on the homepage. We may identify three different parts in it, as shown in Fig. 4.4.

On the top part of the screen, there is a small extract of the scoreboard reporting how the user ranks with respect to data posting and sharing and how many points (s)he has collected so far.

Below this scoreboard extract, there is an action button with a calendar icon. This is the entry point to the calendar module, an interface through which the user can add posts for all the tasks she carries out in her farm(s).

Finally, the bottom part of the screen is occupied by a menu bar letting access to four items: the user profile, the scoreboard structure ("rank" button), an add-in component for making brief notes and setting reminders ('stickies' button), and the social layer of the app ('community' button).

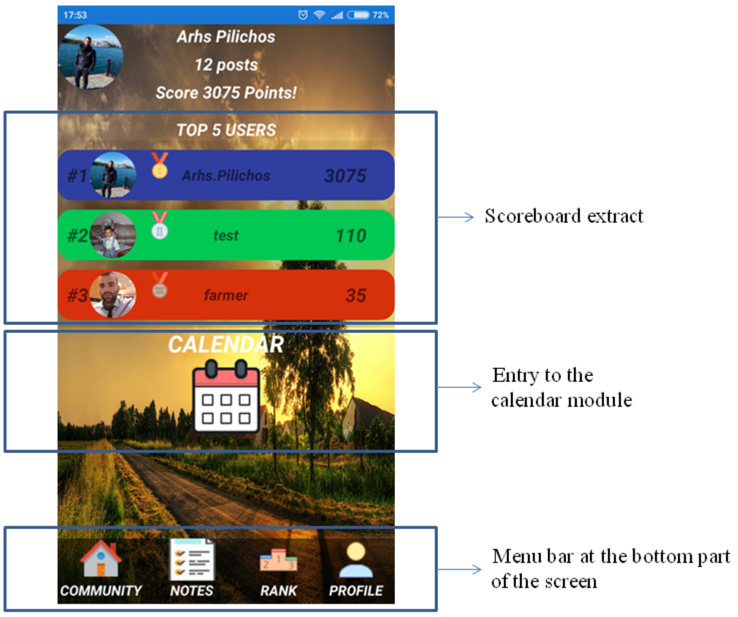
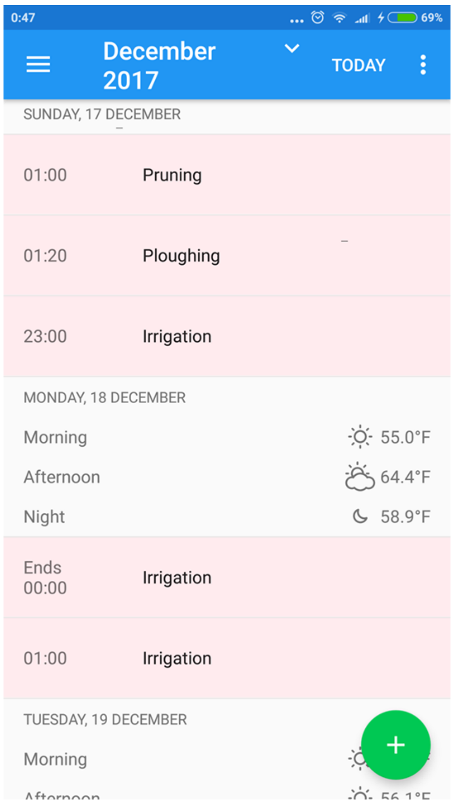


Figure 4.4: The user homepage and its three parts: the calendar icon-button, the extract from the scoreboard interface and the menu bar with four action buttons at the bottom.



SUNDAY, 17 DECEMBER

01:00	Pruning
01:20	Ploughing
23:00	Irrigation

MONDAY, 18 DECEMBER

Morning	55.0°F
Afternoon	64.4°F
Night	58.9°F

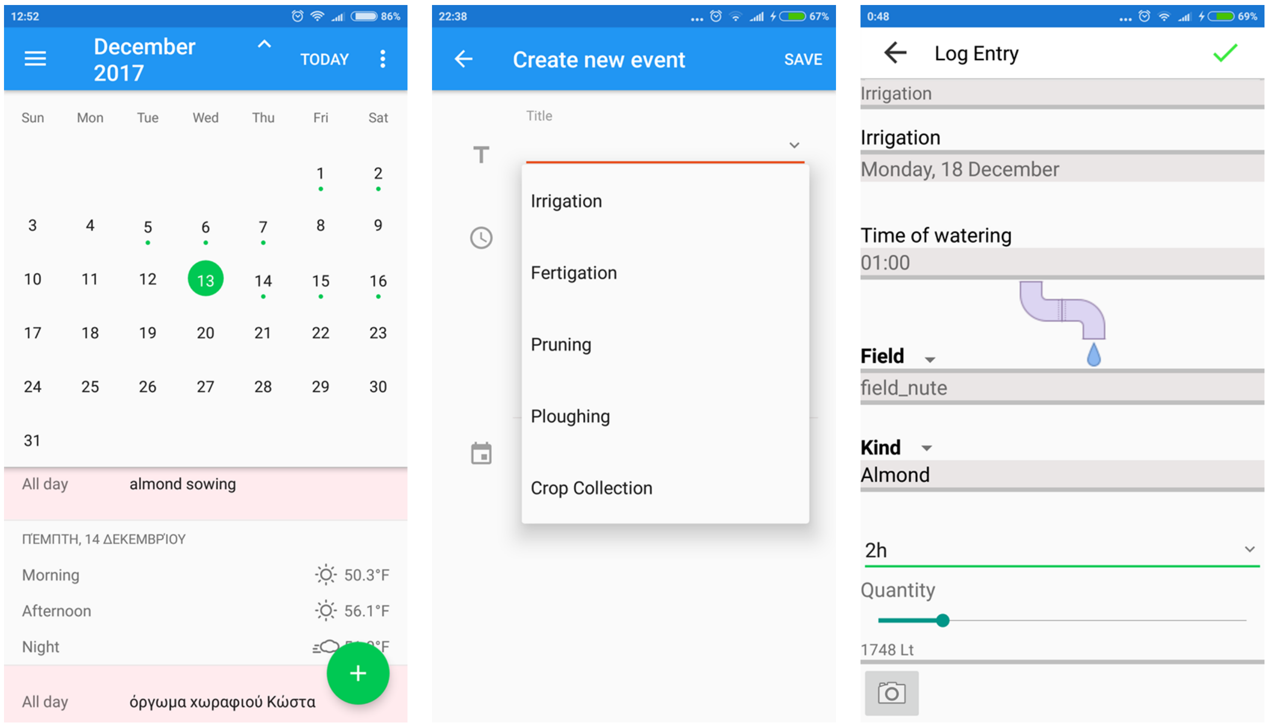
Ends 00:00	Irrigation
01:00	Irrigation

TUESDAY, 19 DECEMBER

Morning

Afternoon 56.1°F

Figure 4.5: Entry calendar screen.



Irrigation

Irrigation
Monday, 18 December

Time of watering
01:00

Field
field_nute

Kind
Almond

2h

Quantity
1748 Lt

(a) Creating an entry

(b) BetterSpinner choices to the entry

(c) Form for adding an almond-irrigation entry to the calendar

Figure 4.6: Adding entries to the calendar interface.

4.3.1.4 The calendar module

This represents the core functionality of the application. It presents the user with a calendar interface, where she can add entries for different activities undertaken in the farm (Fig. 4.5). The user may scroll the calendar left and right to the proper month and choose a day to make an entry for a task.

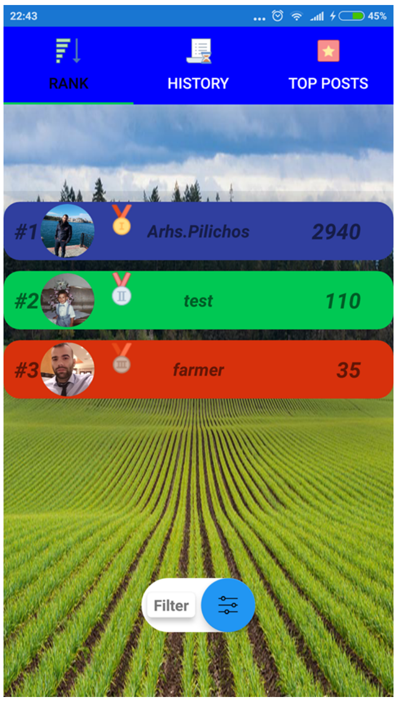
As soon as the green ADD button is pressed, the user is presented with a menu of all the farms she has declared in her profile. She chooses the one her entry is intended for and, through the profile information, the app knows what kind of crop (e.g., almond, chestnut, apple etc) the entry relates to. Depending on the crop, the user is presented with a drop-down menu of the possible relevant activities for the specific crop; for instance, irrigation, spraying, tree pruning. The choice of each of these activities opens an activity-specific form that is customized for the activity context (Fig. 4.6). For example, if the activity is irrigation, the form has user-friendly interfaces for logging the duration of the activity, the time, the volume of water used, the method (e.g., surface, sprinker, drip/trickle).

4.3.1.5 The scoreboard module

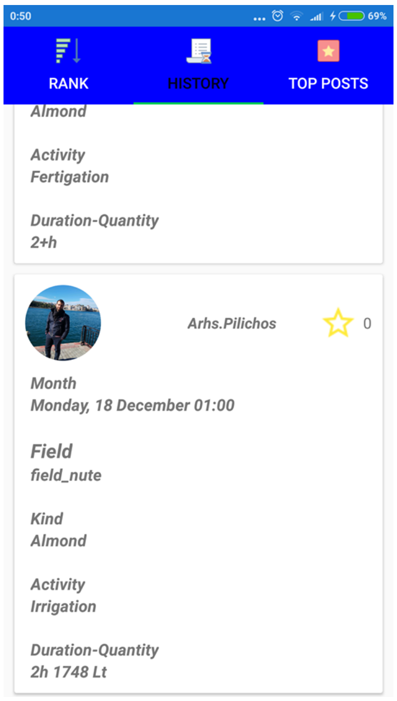
This component is a major part of the gamification functionality in the app. It computes and reports the points that users collect for sharing posts and knowledge with other application users. By pushing the “rank” button in the homepage, users are navigated to the main scoreboard page that logs their current rank among the other app users, as shown in Fig. 4.7 (a).

The top of the main scoreboard page includes a menu bar with three buttons. The ’history’ button offers view to the list of prior user posts, as one can see in Fig. 4.7 (b).

The scoreboard page also provides a filter button. Through this, the user may choose to filter the full ranking



(a) Main scoreboard page



Almond

Activity
Fertigation

Duration-Quantity
2+h

Month
Monday, 18 December 01:00

Field
field_nute

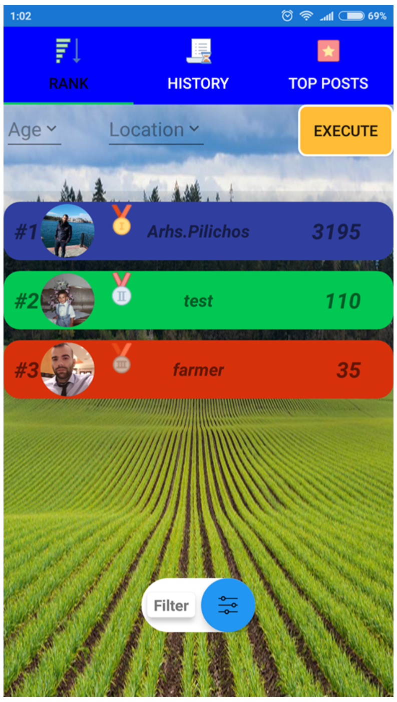
Kind
Almond

Activity
Irrigation

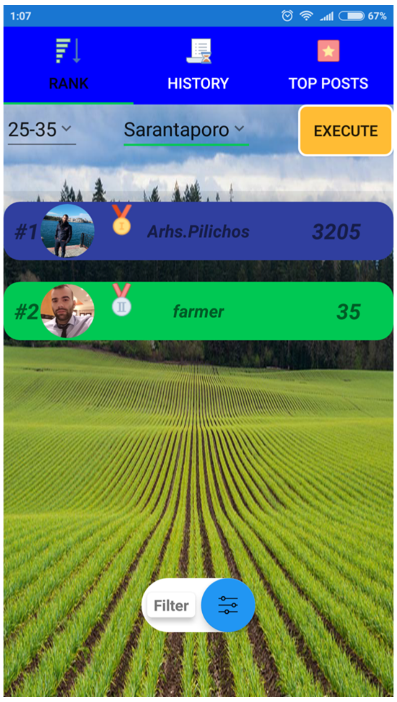
Duration-Quantity
2h 1748 Lt

(b) History of user shared posts

Figure 4.7: Scoreboard layout and user post history

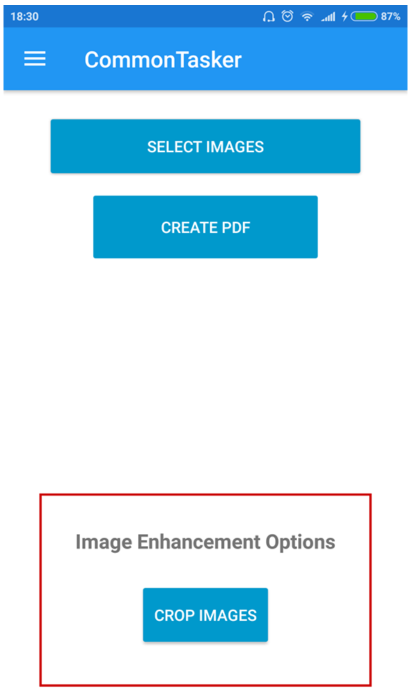


(a) Filtering options

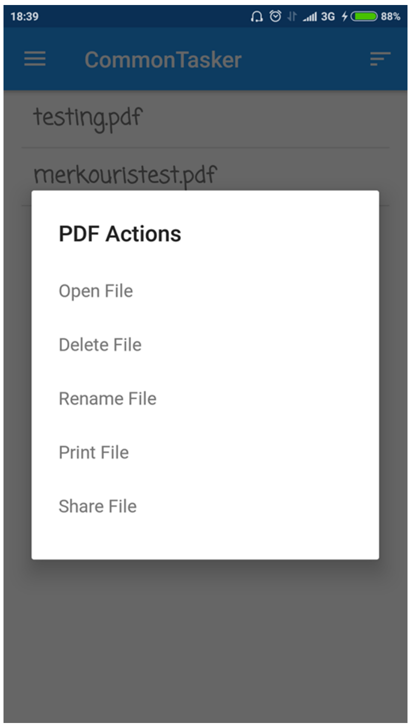


(b) Filter user per village in the main rank page.

Figure 4.8: Filtering capabilities in the scoreboard module.



(a) Main converter page of the mobile app



(b) Possible ways to act on PDF files

Figure 4.9: Exporting post entries as .pdf files in the CommonTasker app.

information in the scoreboard according to age group or village of origin (Fig. 4.8). There is also the possibility to aggregate scores across villages and depict how villages as a whole rank in the contest of sharing posts.

4.3.1.6 The export module converting posts to PDF files

The different post entries of the user can be captured as pictures and be exported as .pdf files. The entries can be inserted in an internal local files in the mobile phone for long-term use (user history). These files may be processed (e.g., printed, deleted and shared), as shown in Fig. 4.9.

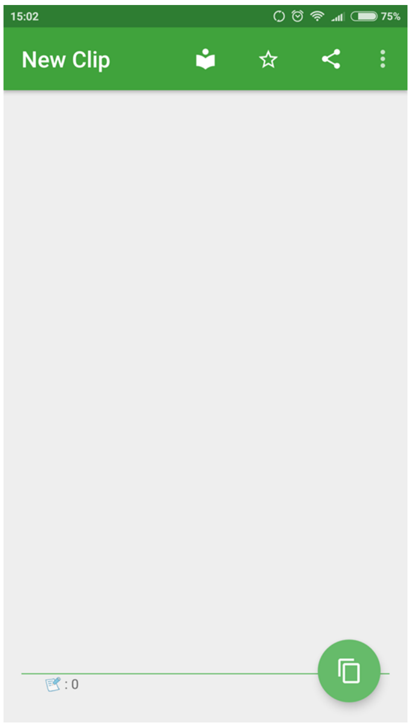
4.3.1.7 The notes-taking plug-in

This is a generic add-on letting the user make notes on-the-go about tasks/things she does not want to forget. Each entry can be marked as simple or persistent; in the latter case, it stays on the list and is not deleted. Fig. 4.10 reports the normal read mode (a) and a note marked persistent (b).

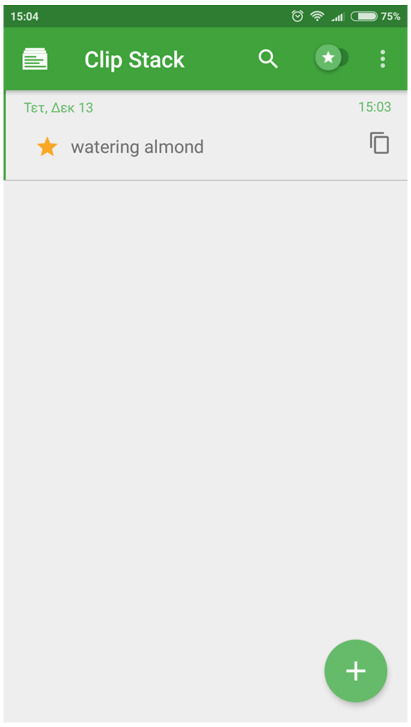
4.3.1.8 The community module

By pressing the button “community” in the homepage, the user opens the social networking functionality of the app. Essentially, the user gets access to a sub-application that gives him the opportunity to interact with fellow farmers in ways she/he is familiar with by her experience with mainstream social networking sites. This wrap-up sub-application includes:

- A photo-sharing component (Fig. 4.11 (a)): This piece of the (sub-)application aims to let users create and share moments of social everyday interaction. Each farmer can create photo snapshots and process

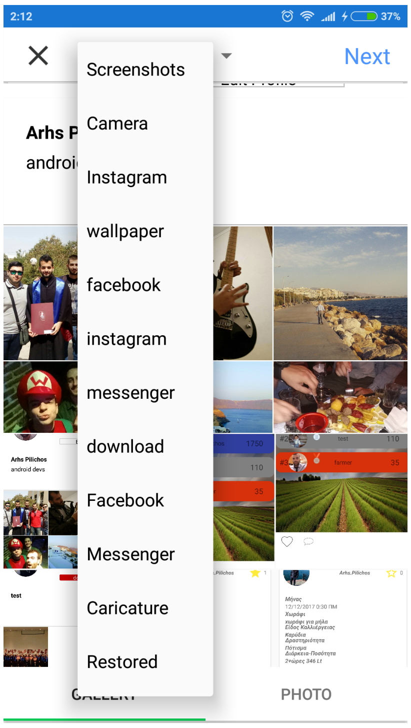


(a) Main page (read mode)

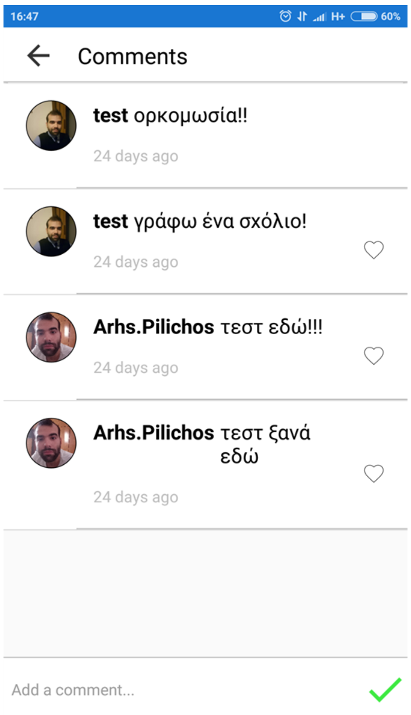


(b) Mark a note as persistent (edit mode)

Figure 4.10: Note-taking app plug-in



(a) Add a new image in newsfeed



test ορκομωσία!!
24 days ago

test γράφω ένα σχόλιο!
24 days ago

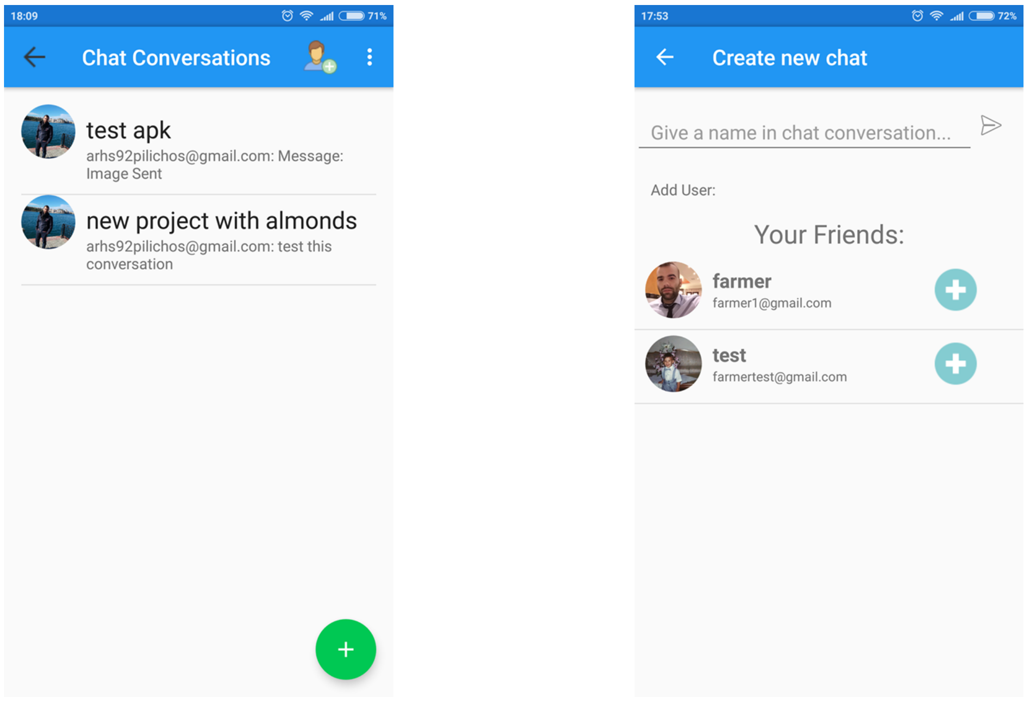
Arhs.Pilichos τεστ εδώ!!!
24 days ago

Arhs.Pilichos τεστ ξανά εδώ
24 days ago

Add a comment...

(b) Comment a news feed image

Figure 4.11: Photo-sharing and commenting



test apk
arhs92pilichos@gmail.com: Message: Image Sent

new project with almonds
arhs92pilichos@gmail.com: test this conversation

farmer
farmer1@gmail.com

test
farmertest@gmail.com

(a) Main page (b) Adding a friend to the chat

Figure 4.12: Chat interface

them with an appropriate commentary on a photo. She can also act on the photo, i.e., add a Like or an encouraging comment;

- A chat component (Fig. 4.11 (b)): This let a user create a chat either point-to-point or in group mode. He/She can dynamically change his friends and automatically create groups with users who better fit to the current topic of discussion.

Indeed, this sub-app features its own distinct profiling capabilities. The user can determine which other users to follow (assisted by rich search functionality), which users view what she shares, and create multiple groups for sharing different things. Fig. 4.12 and Fig. 4.13 report four snapshots showing the main page of the chat interface (Fig. 4.12 (a)), the action of adding a friend (Fig. 4.12 (b)), a chat with embedded images (Fig. 4.13 (a)), and finally the action of sending an image on the chat (Fig. 4.13 (b)).

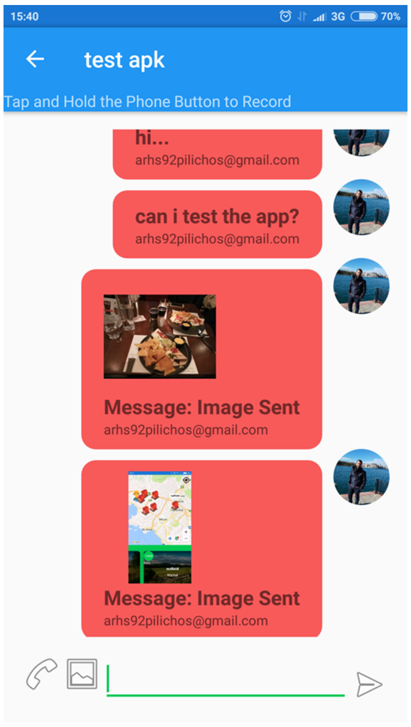
4.3.1.9 The user profile pages

This is the generic profile of the user containing the global information the user wants to share on the app (Fig. 4.14 (a)). By tapping the 'My fields' button, she/he can navigate over her fields in Google maps, while a sliding list at the bottom of the screen summarizes the basic information (area, crop) for each of them, following the user focus (Fig. 4.14 (b)).

The profile can be edited by tapping the pencil on the right. The user can update his/her personal information (Fig. 4.14 (c)), while a selection on the left hand side gives him the opportunity to move towards one of three directions: either to the main homepage, either to the community or the track with his file records.

4.3.2 Backend server

The backend server is currently implemented on Firebase as shown in Fig. 4.15. Firebase is a mobile and web application development platform, made up of complementary features that developers can mix-and-match to


15:40 3G 70%

test apk

Tap and Hold the Phone Button to Record

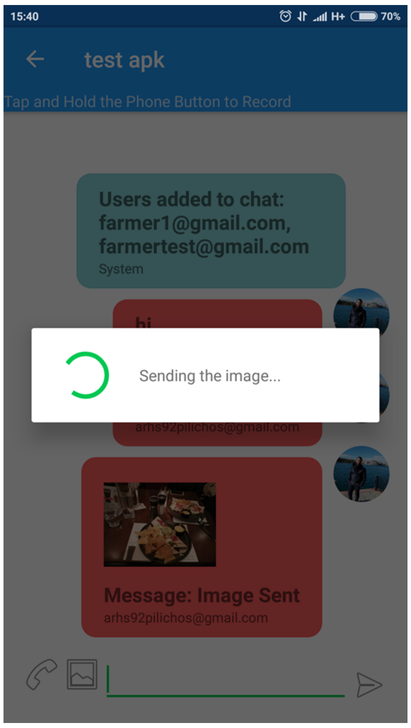
hi...
arhs92pilichos@gmail.com

can i test the app?
arhs92pilichos@gmail.com

Message: Image Sent
arhs92pilichos@gmail.com

Message: Image Sent
arhs92pilichos@gmail.com

(a) Chatting interface


15:40 H+ 70%

test apk

Tap and Hold the Phone Button to Record

Users added to chat:
farmer1@gmail.com,
farmertest@gmail.com
System

hi

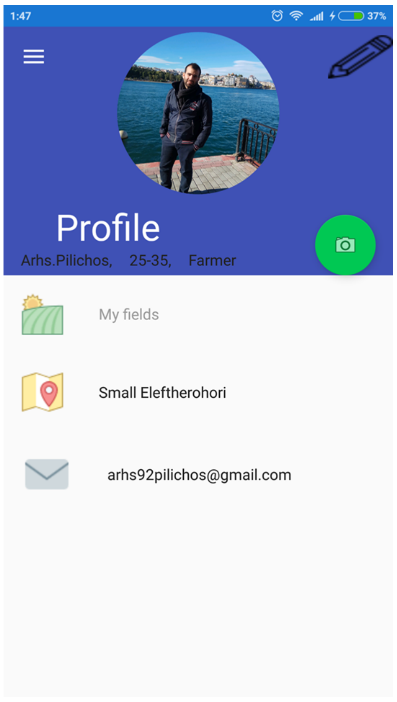
Sending the image...

arhs92pilichos@gmail.com

Message: Image Sent
arhs92pilichos@gmail.com

(b) Sending the image

Figure 4.13: Chatting with embedded images


1:47 37%

Profile

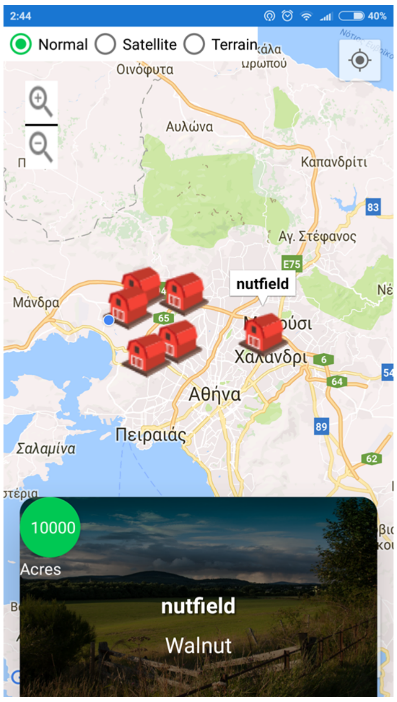
Arhs.Pilichos, 25-35, Farmer

My fields

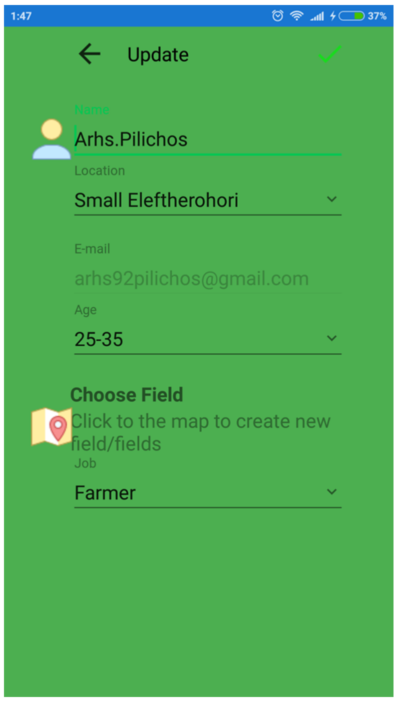
Small Eleftherohori

arhs92pilichos@gmail.com

(a) Main page



(b) Field browser


1:47 37%

Update

Name
Arhs.Pilichos

Location
Small Eleftherohori

E-mail
arhs92pilichos@gmail.com

Age
25-35

Choose Field
Click to the map to create new field/fields

Job
Farmer

(c) Editing profile info

Figure 4.14: User profile

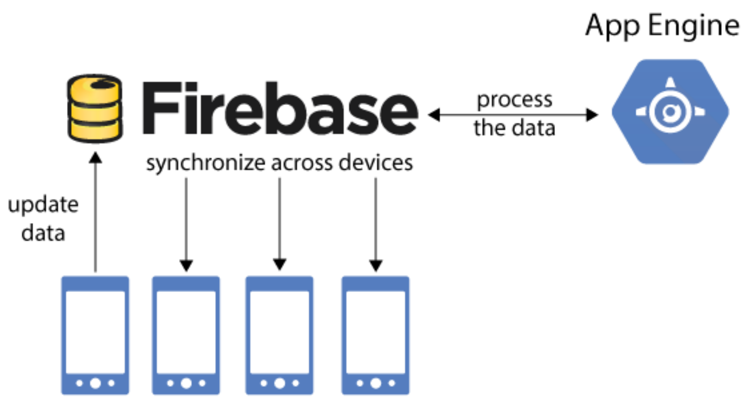


Figure 4.15: Firebase

fit their needs. CommonTasker utilizes Firebase services to store the received user inputs. Firebase allows for data storage in JSON format and immediate synchronization across all platforms and devices when changes are performed in the data. These desired features have enabled a flexible real-time version of CommonTasker. In fact, response data for the application are delivered within seconds (within 200 ms after the arrival on the asynchronous database).

The integration of Firebase libraries includes the following steps:

1. Adding rules to the build.gradlefile, to include google-services plugin;
2. In app/build.gradle file add: applying plugin: 'com.google.gms.google-services';
3. Adding dependencies using packages com.google.firebase:firebase-auth:11.0.2 for Authentication, com.google.firebase:firebase-database:11.0.2 for Real-time Database and com.google.firebase:firebase-storage:11.0.2 for Storage.

Additional services of Firebase used in the updated version of CommonTasker are presented in detail below.

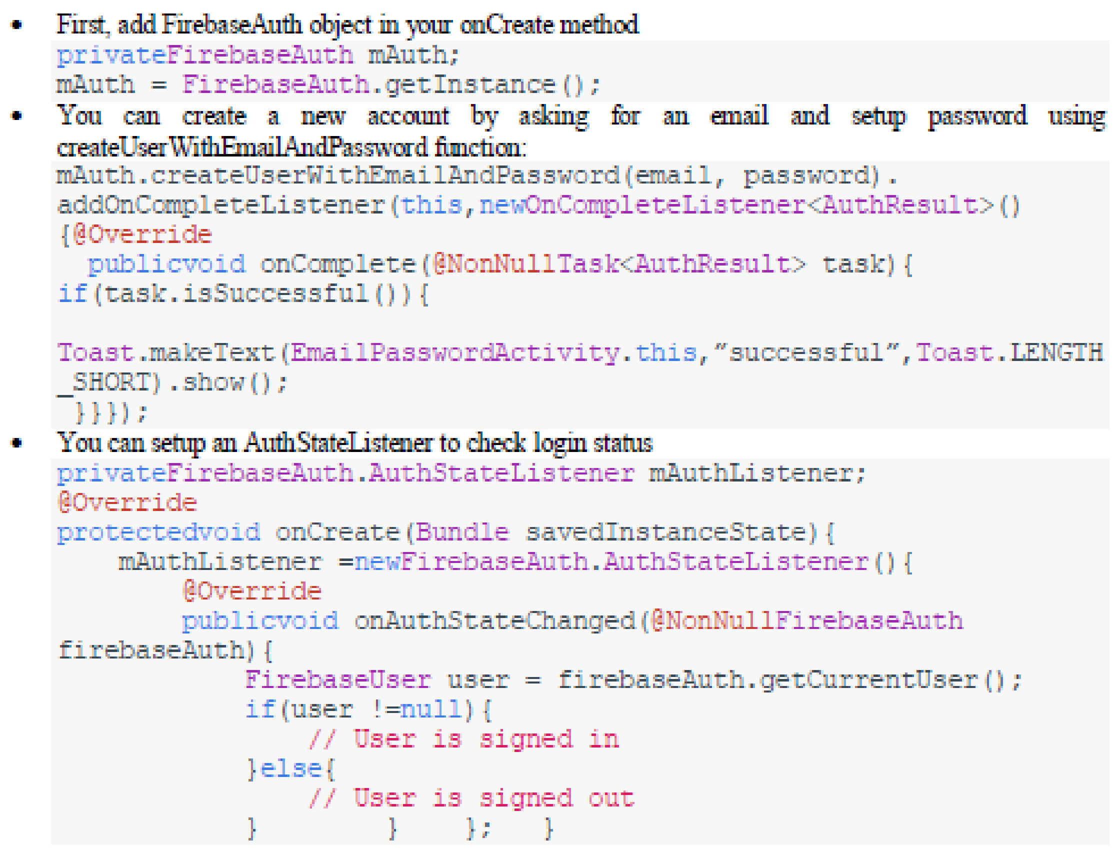
4.3.2.1 Authentication

Authentication is performed using the Firebase Auth service and grants write-access to the owner of this user account, whose uid must exactly match the key (getUid), and read access to any user who is logged in with an email and password.

Firebase Auth is a service that can authenticate users using only client-side code. Additionally, it includes a user management system that enables user authentication via email and password login stored in Firebase. It is possible to trigger a function in response to the creation and deletion of user accounts via Firebase Authentication. In detail, the app can send a welcome email to a user who has just created an account using the following steps. First, obtain an instance of this class by calling getInstance(). Then, sign up or sign in as user with one of the following methods:

- createUserWithEmailAndPassword(String,String)
- signInWithEmailAndPassword(String,String)
- signInWithCredential(AuthCredential)
- signInAnonymously()
- signInWithCustomToken(String)

When users sign in to the app, their sign-in credentials (for example, their username and password) are sent to



- First, add FirebaseAuth object in your onCreate method

```
privateFirebaseAuth mAuth;
mAuth = FirebaseAuth.getInstance();
```

- You can create a new account by asking for an email and setup password using createUserWithEmailAndPassword function:

```
mAuth.createUserWithEmailAndPassword(email, password).
addOnCompleteListener(this,newOnCompleteListener<AuthResult>()
{@Override
  publicvoid onComplete(@NonNullTask<AuthResult> task){
if(task.isSuccessful()){

Toast.makeText(EmailPasswordActivity.this,"successful",Toast.LENGTH
_SHORT).show();
 }}});
```

- You can setup an AuthStateListener to check login status

```
privateFirebaseAuth.AuthStateListener mAuthListener;
@Override
protectedvoid onCreate(Bundle savedInstanceState){
    mAuthListener =newFirebaseAuth.AuthStateListener(){
        @Override
        publicvoid onAuthStateChanged(@NonNullFirebaseAuth
firebaseAuth){
            FirebaseUser user = firebaseAuth.getCurrentUser();
            if(user !=null){
                // User is signed in
            }else{
                // User is signed out
            }        }     };    }
```

Figure 4.16: User Authentication and creation of login ID.

the authentication server. The server checks the credentials and returns a custom token if they are valid. After a user signs in for the first time, a new user account is created and linked to the credentials—that is, the user name and password, phone number, or auth provider information—the user signed in with. This new account is stored as part of the Firebase project, and can be used to identify a user across every app in your project, regardless of how the user signs in.

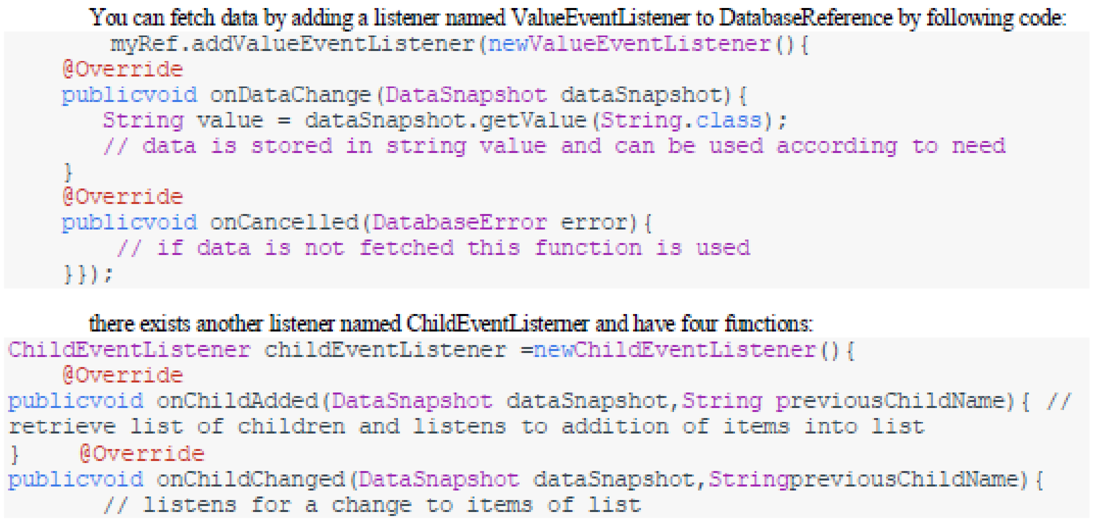
After having added Firebase and Authentication dependencies to CommonTasker, we can let the user create login id through the following steps described in Fig. 4.16.

4.3.2.2 Real-time Database

The Real-time Database service provides an API that allows application data to be synchronized across clients and stored on Firebase Cloud. The ReST API uses the Server-Sent Events protocol, which is an API for creating HTTP connections for receiving push notifications from a server.

The Real-time database supports functions that manage authentication credentials. A Firebase reference represents a particular location in your Database and can be used for reading or writing data to that database location. This class is the starting point for all database operations. After one initialises it with a URL, she can use it to read data, write data, and to create new DatabaseReferences. With Cloud Functions, you can handle events in the Firebase Realtime Database with no need to update client code. Cloud Functions lets run database operations with full administrative privileges, and ensures that each change to the database is processed individually. Functions enable handling database events at two levels of specificity: one can listen specifically for creation, update, or deletion events only; or one can listen to a path for a change of any kind.

Cloud Functions supports the following event handlers for Realtime Database:



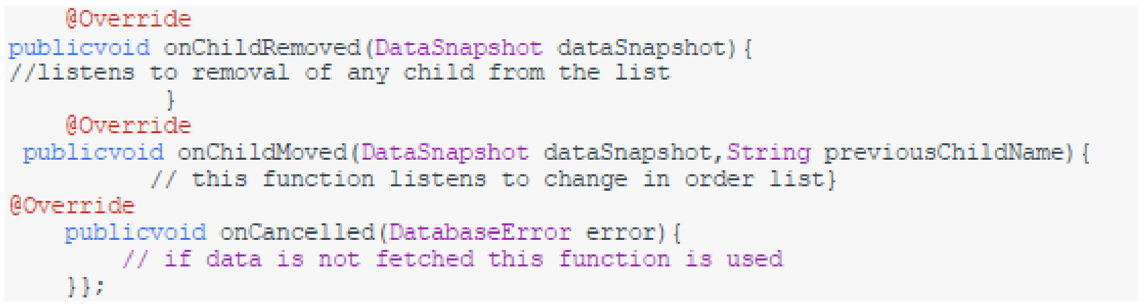
You can fetch data by adding a listener named ValueEventListener to DatabaseReference by following code:

```
      myRef.addValueEventListener(newValueEventListener(){
    @Override
    publicvoid onDataChange(DataSnapshot dataSnapshot){
       String value = dataSnapshot.getValue(String.class);
       // data is stored in string value and can be used according to need
    }
    @Override
    publicvoid onCancelled(DatabaseError error){
        // if data is not fetched this function is used
    }});
```

there exists another listener named ChildEventListerner and have four functions:

```
ChildEventListener childEventListener =newChildEventListener(){
    @Override
publicvoid onChildAdded(DataSnapshot dataSnapshot,String previousChildName){ //
retrieve list of children and listens to addition of items into list
}     @Override
publicvoid onChildChanged(DataSnapshot dataSnapshot,StringpreviousChildName){
       // listens for a change to items of list
```

Figure 4.17: Real-time Database: Listeners code.



```
    @Override
publicvoid onChildRemoved(DataSnapshot dataSnapshot){
//listens to removal of any child from the list
           }
    @Override
 publicvoid onChildMoved(DataSnapshot dataSnapshot,String previousChildName){
          // this function listens to change in order list}
@Override
    publicvoid onCancelled(DatabaseError error){
        // if data is not fetched this function is used
    }};
```

Figure 4.18: Real-time Database: Child Event code.

- onWrite(), which is triggered when data is created, destroyed, or changed in the Realtime Database;
- onCreate(), which is triggered when new data is created in the Realtime Database;
- onUpdate(), triggered when data is updated in the Realtime Database;
- onDelete(), which is triggered when data is deleted from the Realtime Database. To control when a function should be triggered, call ref(path).

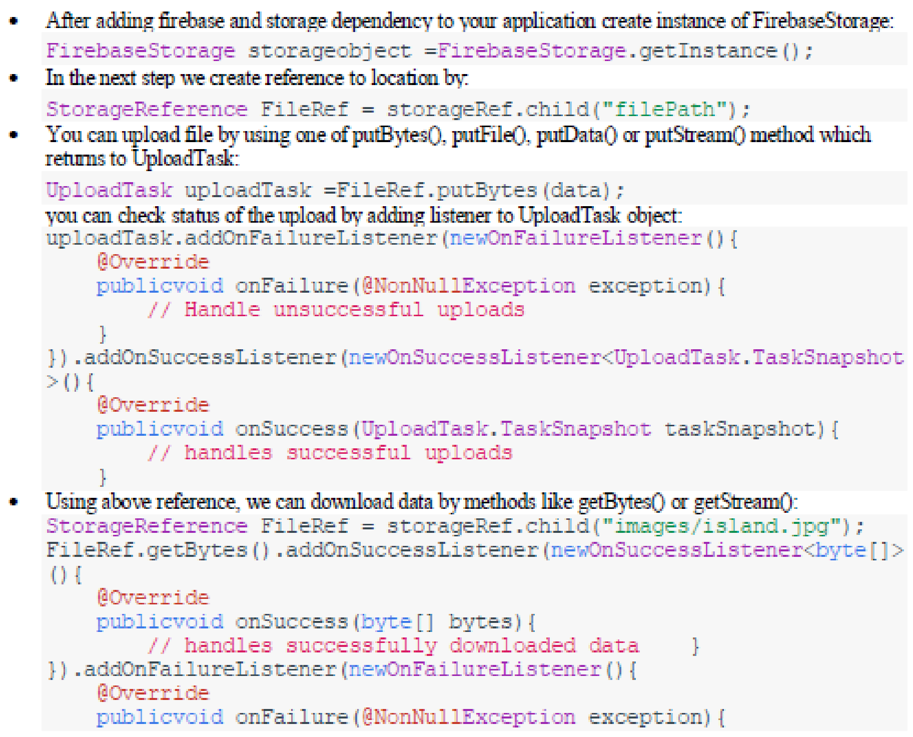
The Firebase Realtime Database can be accessed directly from a mobile device or web browser, i.e., there's no need for an application server. Security and data validation are available through the Firebase Realtime Database Security Rules, expression-based rules that are executed when data is read or written.

Data can be fetched by adding a listener named ValueEventListener to DatabaseReference by following the codes in Figure 4.17 and 4.18.

Moreover, data can be retrieved in sorted order by using one of the following functions:

- orderByChild() - orders results by the value of child key;
- orderByKey() - orders results by child keys;
- orderByValue()- orders results by child values.

Lastly, data can be deleted using current DatabaseReference by calling removeValue() function.



- After adding firebase and storage dependency to your application create instance of FirebaseStorage:

```
FirebaseStorage storageobject =FirebaseStorage.getInstance();
```

- In the next step we create reference to location by:

```
StorageReference FileRef = storageRef.child("filePath");
```

- You can upload file by using one of putBytes(), putFile(), putData() or putStream() method which returns to UploadTask:

```
UploadTask uploadTask =FileRef.putBytes(data);
```

you can check status of the upload by adding listener to UploadTask object:

```
uploadTask.addOnFailureListener(newOnFailureListener(){
    @Override
    publicvoid onFailure(@NonNullException exception){
        // Handle unsuccessful uploads
    }
}).addOnSuccessListener(newOnSuccessListener<UploadTask.TaskSnapshot
>(){
    @Override
    publicvoid onSuccess(UploadTask.TaskSnapshot taskSnapshot){
        // handles successful uploads
    }
```

- Using above reference, we can download data by methods like getBytes() or getStream():

```
StorageReference FileRef = storageRef.child("images/island.jpg");
FileRef.getBytes().addOnSuccessListener(newOnSuccessListener<byte[]>
(){
    @Override
    publicvoid onSuccess(byte[] bytes){
        // handles successfully downloaded data     }
}).addOnFailureListener(newOnFailureListener(){
    @Override
    publicvoid onFailure(@NonNullException exception){
```

Figure 4.19: Storage Upload item code.

4.3.2.3 Storage

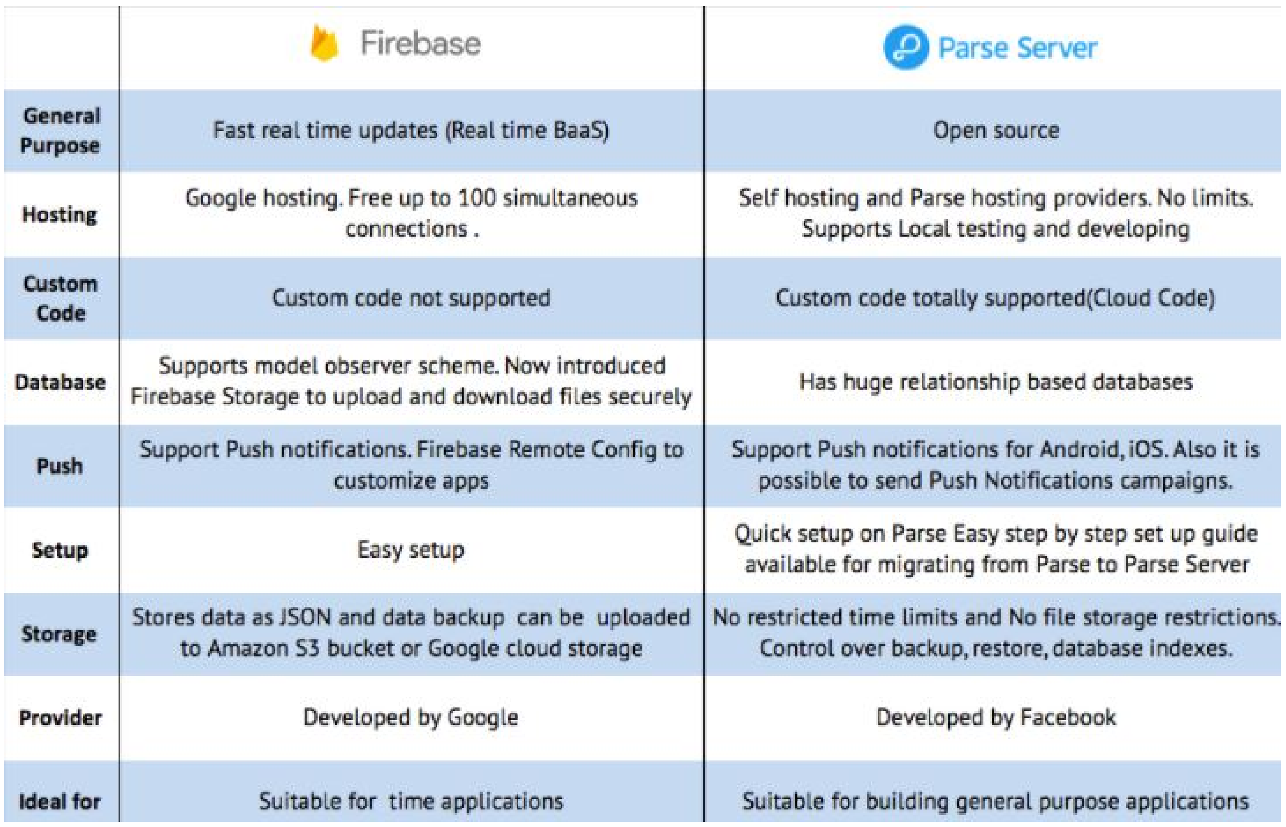
Firebase Storage provides secure file uploads and downloads for Firebase apps, regardless of network quality. The storage feature enables storing files such as images, audio, video. Data is stored in a highly secure and robust environment and upload/download operations can resume from a previous updated point in case a network error occurs. The steps reported in Fig. 4.19 are followed to use the storage feature in the android application.

4.3.2.4 Some forward-thinking about the backend

Open-source alternatives to Firebase: Firebase is a powerful tool that features a rich set of capabilities and enjoys trust from many app developers. Its development is a vivid project, at least for the time being, and it is led by Google. The company hosts the service in the cloud, supporting its operation and updates. Given the additional functionalities it leverages from other Google services and its attractive pricing policy (free up to 1 GB storage space and 10 GB downloads monthly, 25 USD per month for more intensive usage), it emerged as the apparent backend choice for developing the mobile app.

However, this decision was not the outcome of exhaustive search and comparison. Firebase has some drawbacks in our case, related mainly to the flexibility about managing the app data. It is not open source and does not cater for self hosting, which would allow the deployment of the backend service at some local server in the Sarantaporo.gr CN.

One task that will be carried out in the remainder of the project, is a feasibility study of substituting Firebase with an open-source backend solution. One of the most promising solutions in this respect is Parse Server. The tool was originally developed and supported by Facebook under the label Parse. However, the project was ceased by Facebook in end Jan 2017 and at the same day an open source substitute, with limited capabilities, was released under the name Parse Server. The community of developers that maintains Parse Server at github have restored much of the original functionality of the Parse since then. A brief comparison of the two tools is given in Fig. 4.20.



	Firebase	Parse Server
General Purpose	Fast real time updates (Real time BaaS)	Open source
Hosting	Google hosting. Free up to 100 simultaneous connections .	Self hosting and Parse hosting providers. No limits. Supports Local testing and developing
Custom Code	Custom code not supported	Custom code totally supported(Cloud Code)
Database	Supports model observer scheme. Now introduced Firebase Storage to upload and download files securely	Has huge relationship based databases
Push	Support Push notifications. Firebase Remote Config to customize apps	Support Push notifications for Android, iOS. Also it is possible to send Push Notifications campaigns.
Setup	Easy setup	Quick setup on Parse Easy step by step set up guide available for migrating from Parse to Parse Server
Storage	Stores data as JSON and data backup can be uploaded to Amazon S3 bucket or Google cloud storage	No restricted time limits and No file storage restrictions. Control over backup, restore, database indexes.
Provider	Developed by Google	Developed by Facebook
Ideal for	Suitable for time applications	Suitable for building general purpose applications

Figure 4.20: High-level comparison between Firebase and Parse-Server, now an open-source project without the support of Facebook.

The breadth and depth of the feasibility study and the subsequent steps to be taken, beyond the time horizon of the project, will ultimately depend on the outcome of interactions with the Sarantaporo.gr community in the context of the participatory design process. Namely, how strong their concerns will be about where data are stored as well as what are the available resources locally for supporting the application.

Catering for external service providers like GAIA: In Sec. 4.1 we mentioned the possibility to make the app data available to external service providers (e.g., GAIA), in the context of strategic synergy of the Sarantaporo.gr CN with them. In that case, the CommonTasker should be enhanced with additional interfaces that depend on the application at hand (Fig. 4.21).

In the specific example of interfacing with the GAIA Sense platform, the minimum additional functionalities of CommonTasker would include:

- The **API that needs to be defined between the GAIA provider cloud and Firebase** (or any other mobile app backend that might substitute Firebase);
- **The synchronization of users' accounts in the GAIA web app and their profiles in CommonTasker**. This requires the flexibility to sign-in users (using multiple providers, like Facebook Login or Google Sign-in) and then have the authentication system recognize the user even if they tried signing in via a different method. When authenticating with an email/password, one can store in Shared Preferences an authentication token, which can be used later to restore a session even after the app is killed. Upon creating a new user, the user can be registered and then logged in realtime database. Firebase will then retrieve farmer data from the GAIA Cloud through the API, match them with the auth token and join with the Score database reference (Fig. 4.22 and Fig. 4.23).

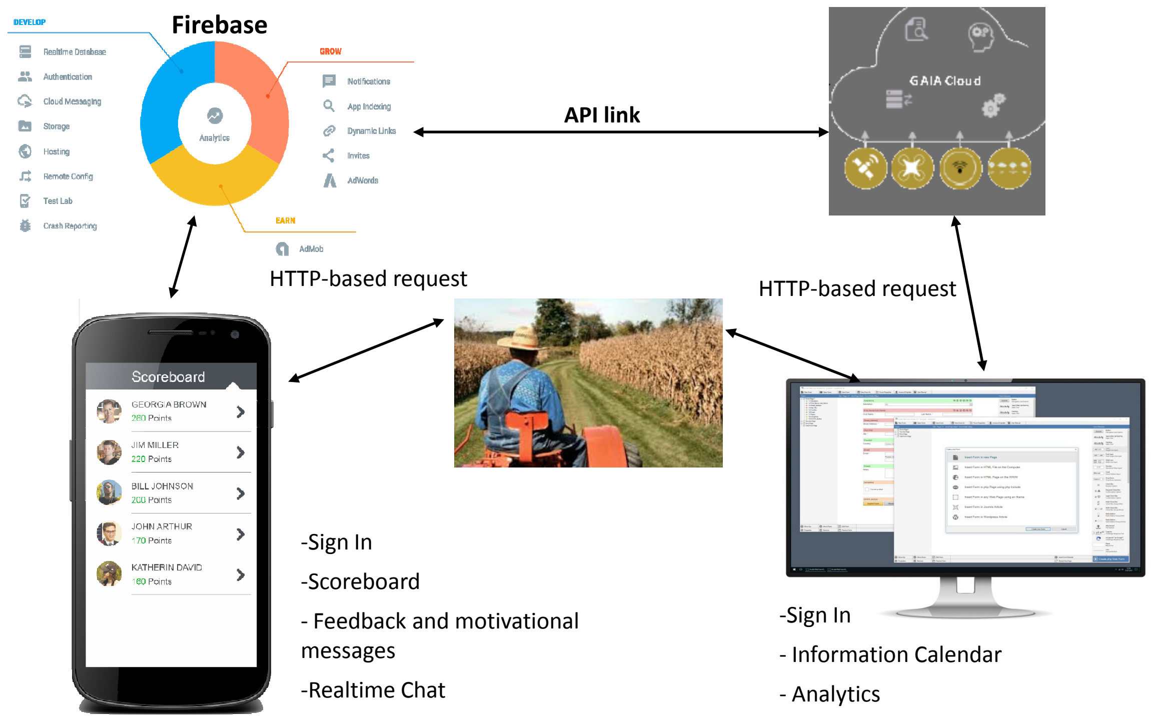


Figure 4.21: Communication between mobile app and web app.

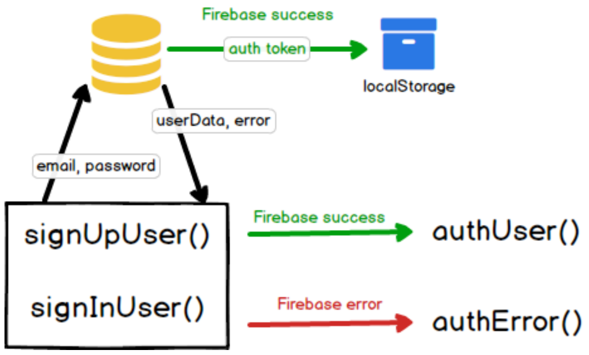


Figure 4.22: Authentication of a user during Sign In/Up processes.

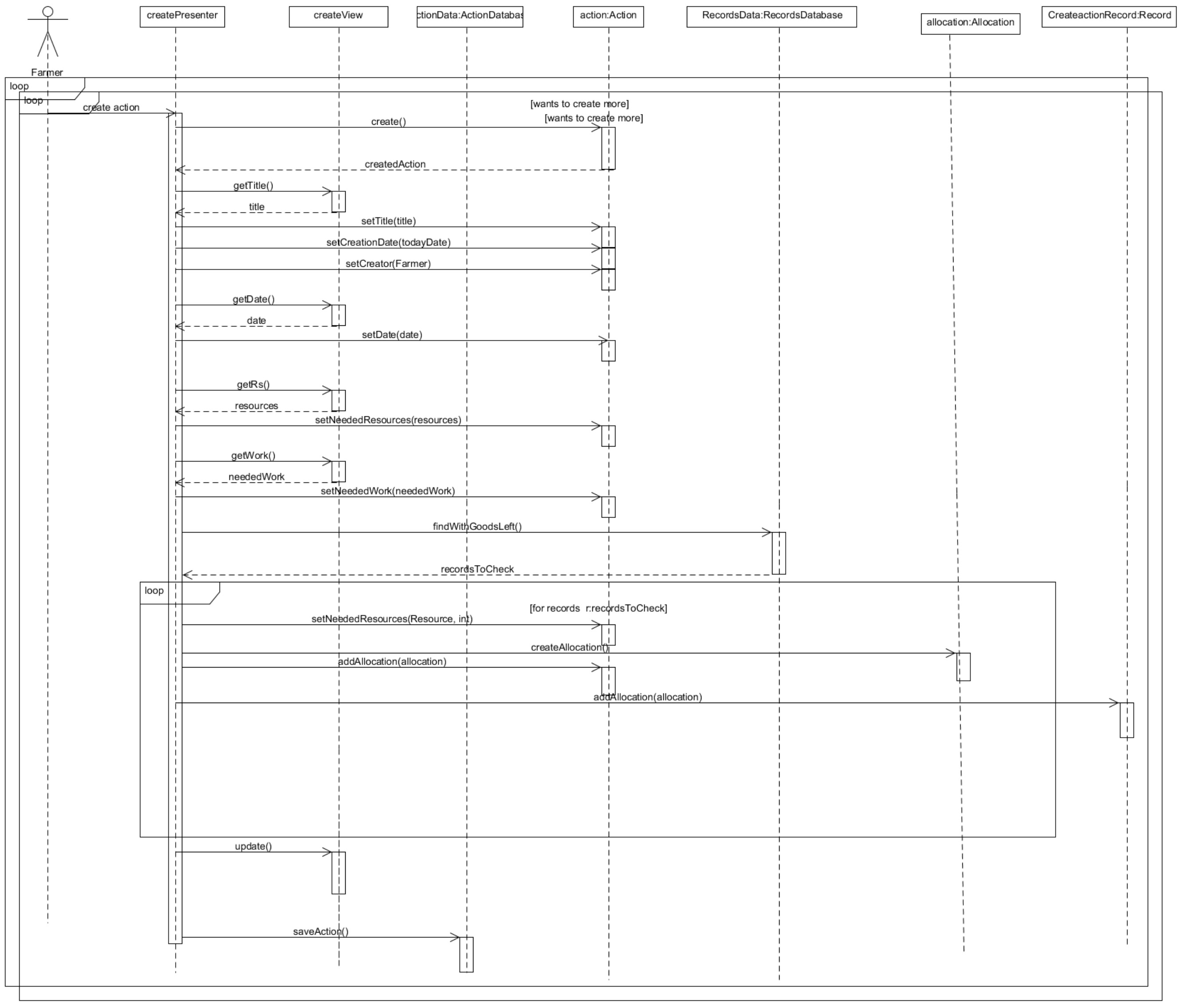


Figure 4.23: Sequence diagram to create an action of their daily work.

Data collection and logging	🟩 RDC1	🟩 RDC2	🟩 RDC3	🟩 RDC4	🟩 RDC5	🟩 RDC6	🟩 RDC7
Information feeding	🟩 RI1	🟧 RI2					
Gamification	🟩 RG1	🟩 RG2	🟩 RG3	🟩 RG4	🟧 RG5	🟥 RG6	
Social networking	🟩 RS1	🟩 RS2	🟩 RS3				
Data management and privacy control	🟧 RP1	🟧 RP2	🟥 RP3				
User profiling	🟩 RPR1	🟥 RPR2	🟩 RPR3	🟩 RPR4			
User interface and graphic design	🟩 RU1	🟩 RU2	🟩 RU3	🟧 RU4			

🟩 complete

🟧 ongoing development work

🟥 design questions need to be clarified

Figure 4.24: Requirements for CommonTasker and level of fulfillment by current development work.

4.4 Roadmap for the last year of the project

4.4.1 Remaining development work

The development work that has been carried out so far has addressed almost all requirements related to the data collection and logging module (at least for selected cases of crops), as well as the gamification, social networking and the user profiling modules. It has also implemented most of the requirements that relate to the user interface and graphic design. On the contrary, moderate progress is achieved so far with respect to the data privacy and management requirements. Further interactions with the community are required to capture the sensitivity of the locals to data control and ownership issues. These issues are interrelated with the extent to which the rich functionality of the Firebase backend could be substituted by an open-source backend implementation that is portable to the local (CN) infrastructure.

The status of the current development work for the CommonTasker app in terms of realizing the requirements in Sec. 4.2 is summarized in Fig. 4.24.

4.4.2 Remaining development work and distribution of the app to the local community

Three sequential phases are envisaged for the work on the application in Y3 of the project.

Phase 1: The development work on the CommonTasker will continue addressing the requirements that are not yet satisfied (see 4.4.1). This involves the interaction with the local community and the Sarantaporo.gr management team and the resolution of open issues. The details of this interaction are given in netCommons D3.3 [2].

Phase 1 is intended to last till mid March 2018. By then, the plan is to hold another meeting in the area of Sarantaporo, to present the first test release of the app to the local community. The exact format of the meeting is to be defined, i.e., whether it will be open to the whole local community or targeted to selected members, as the case was with the November ’16 workshop.

One of the meeting tasks will consist in defining a beta testing team that will be constantly downloading and testing new releases of the app.

Phase 2: The definition of the beta testing team marks the starting point of Phase 2 in the app progress. The development work will continue, only now it will be largely driven by the feedback received by the intended users of the app. The design iterations and ramifications, in response to interactions with Sarantaporo.gr, will also continue, only they are expected to be more sporadic than in Phase 1.

Finally, in July, possibly by the time of the last netCommons plenary meeting, the app will be launched and become available to the local community.

Phase 3: With the launch of the app Phase 3 starts, during which the app will be used by farmers and will be assessed with respect to the impact it can have on their daily routine.

5 Conclusions and Year 3 Work

This deliverable has given a detailed record on the state of development of the source code realized by the three development tasks of WP3. All the tasks are mostly aligned with what was expected at M24, with some small delays that can be easily recovered in the third year, but also with many interesting and un-predicted interesting developments.

The third year of the project, in fact, is devoted to the active dissemination of the software products in existing community networks, which were selected during the first two years and with which we kept continuous contacts and interactions. With all the selected communities (Guifi, ninux, and Sarantaporo.gr) we will discuss in order to encourage them to use our software, and we will make ourselves available for testing and improving the code and react to their feedback. All the tasks are ready to offer to the communities at least a subset of the final features to test, and keep developing the rest, if needed.

In the third year, the interaction with the community will be guided by the results of D3.3 [2], which describes a generic methodology for the co-creation of applications in CNs and their efficient deployment and adaptation. We will use the methodology to guide the interactions, set the ground for our cooperation and continue with the development and improvement of the source code. Conversely, we will also give feedback to T3.1 in ways to improve the methodology based on our experience, which will be reported in the newly introduced deliverable D3.6 at M31.

D3.5, at the end of the project, will finalize the work of WP3, it will contain an update on the state of the open source applications, a report on their reception from the chosen community networks and the feedback to the methodology.

Bibliography

[1] P. Antoniadis, I. Apostol, P. Micholia, G. Klissiaris, V. Chryssos, and M. Karaliopoulos, "Multi-Disciplinary Methodology for Applications Design for CNs, including Design Guidelines and Adoption Facilitation (v1)," netCommons Deliverable D3.1, Jan. 2017. http://netcommons.eu/?q=content/multi-disciplinary-methodology-applications-design-cns-including-design-guidelines-and

[2] P. Antoniadis, I. Apostol, and A. Papageorgiou, "Multi-Disciplinary Methodology for Applications Design for CNs, including Design Guidelines and Adoption Facilitation (v2)," netCommons Deliverable D3.3, Mar. 2018. https://www.netcommons.eu/?q=content/multi-disciplinary-methodology-applications-design-cns-including-design-guidelines-and-0

[3] N. Facchi, F. Freitag, L. Maccari, P. Micholia, and F. Zanini, "Release of All Open Source Software for all Applications (v1)," netCommons Deliverable D3.2, Dec. 2016. http://netcommons.eu/?q=content/release-new-open-source-software-all-applications-v1

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16. http://doi.acm.org/10.1145/2342509.2342513

[5] B. Varghese, N. Wang, D. S. Nikolopoulos, and R. Buyya, "Feasibility of fog computing," *CoRR*, vol. abs/1701.05451, 2017. http://arxiv.org/abs/1701.05451

[6] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[7] R. Baig, R. Pueyo, F. Freitag, and L. Navarro, "On edge microclouds to provide local container-based services," in *Global Information Infrastructure and Networking Symposium (GIIS)*, 2017.

[8] M. R. Casters, R. Bouman, and J. van. Dongen, *Pentaho Kettle Solutions: Building Open Source ETL Solutions with Pentaho Data Integration*. Wiley, 2010.

[9] N. Apolónia, F. Freitag, L. Navarro, S. Girdzijauskas, and V. Vlassov, "Gossip-based service monitoring platform for wireless edge cloud computing," in *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*, May 2017, pp. 789–794.

[10] B. Lantz *et al.*, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[11] R. R. Fontes *et al.*, "Mininet-wifi: Emulating software-defined wireless networks," in *Network and Service Management (CNSM), 2015 11th International Conference on*. IEEE, 2015, pp. 384–389.

[12] M. Selimi, L. Cerdà-Alabern, M. Sánchez-Artigas, F. Freitag, and L. Veiga, "Practical service placement approach for microservices architecture," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 401–410.

[13] S. Traverso, L. Abeni, R. Birke, C. Kiraly, E. Leonardi, R. Lo Cigno, and M. Mellia, "Neighborhood Filtering Strategies for Overlay Construction in P2P-TV Systems: Design and Experimental Comparison," *IEEE/ACM Transactions on Networking*, vol. 23, no. 3, pp. 741–754, Jun. 2015. http://dx.doi.org/10.1109/TNET.2014.2307157

[14] L. Baldesi, L. Maccari, and R. Lo Cigno, "Optimized Cooperative Streaming in Wireless Mesh Networks," in *In Proc. of The 15th IFIP Networking Conference (NETWORKING)*, May 2016. http://www.sciencedirect.com/science/article/pii/S1574119216303443

[15] L. Maccari, N. Facchi, L. Baldesi, and R. Lo Cigno, "Optimized P2P Streaming for Wireless distributed

Networks," *Journal of Pervasive and Mobile Computing,Volume 42, pages 335-350*, Dec. 2017, Elsevier. https://www.sciencedirect.com/science/article/pii/S1574119216303443

[16] "Stirling Number of the Second Kind," http://mathworld.wolfram.com/StirlingNumberoftheSecondKind.html.

[17] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 122–132.

[18] A. Sathiaseelan and J. Crowcroft, "Lcd-net: lowest cost denominator networking," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 2, pp. 52–57, 2013.

[19] A. Sathiaseelan, J. Crowcroft, M. Goulden, C. Greiffenhagen, R. Mortier, G. Fairhurst, and D. McAuley, "Paws: Public access wifi service," in *The Third Digital Economy All-hands Meeting: Digital Engagement (DE), Aberdeen, Scotland/United Kingdom*, 2012.

[20] Fon. (2016) Fon corporation. [Online; accessed 10-February-2017]. https://fon.com

[21] A. Abujoda, D. Dietrich, P. Papadimitriou, and A. Sathiaseelan, "Software-defined wireless mesh networks for internet access sharing," *Computer Networks*, vol. 93, Part 2, pp. 359–372, 2015.

[22] E. Dimogerontakis, R. Meseguer, and L. Navarro, "Internet Access for All: Assessing a Crowdsourced Web Proxy Service in a Community Network," in *Passive and Active Measurement Conference (PAM)*, LNCS, vol. 10176, 2017, Springer.

[23] E. Dimogerontakis, J. Neto, R. Meseguer, and L. Navarro, "Client-Side Routing-Agnostic Gateway Selection for heterogeneous Wireless Mesh Networks," in *IFIP/IEEE International Symposium on Integrated Network and Service Management*, Lisbon, Portugal, May 8-12, 2017. http://ieeexplore.ieee.org/abstract/document/7987301/

[24] E. Dimogerontakis, R. Meseguer, L. Navarro, S. Ochoa, and L. Veiga, "Design trade-offs of crowdsourced web access in community networks," in *IEEE 21st International Conference on Computer Supported Cooperative Work in Design*, Wellington, NZ, April 26–28, 2017. http://ieeexplore.ieee.org/document/8066665/

[25] ——, "Community Sharing of Spare Network Capacity," in *IEEE International Conference on Networking, Sensing and Control*, Calabria, Italy, May 16-18, 2017. http://ieeexplore.ieee.org/document/8000108/

[26] L. Abeni, C. Kiraly, A. Russo, M. Biazzini, and R. Lo Cigno, "Design and Implementation of a Generic Library for P2P Streaming," in *ACM Workshop on Advanced Video Streaming Techniques for Peer-to-peer Networks and Social Networking*, Oct. 29, 2010, pp. 43–48. http://doi.acm.org/10.1145/1877891.1877902

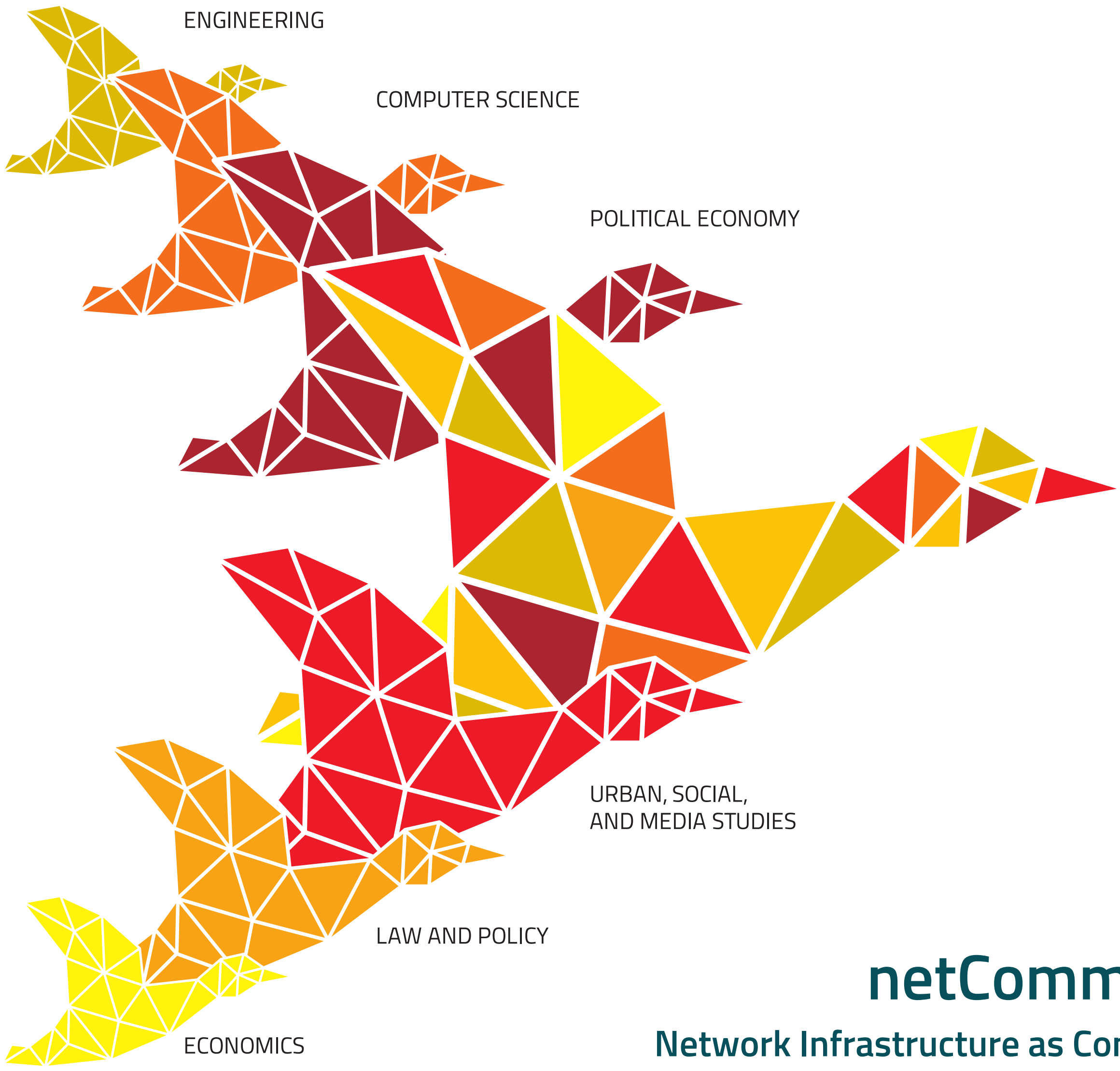
[27] R. Birke, E. Leonardi, M. Mellia, A. Bakay, T. Szemethy, C. Kiraly, R. Lo Cigno, F. Mathieu, L. Muscariello, S. Niccolini, J. Seedorf, and G. Tropea, "Architecture of a network-aware p2p-tv application: the napa-wine approach," *IEEE Communications Magazine*, vol. 49, no. 6, pp. 154–163, June 2011.

[28] A. Russo and R. Lo Cigno, "Delay-Aware Push/Pull Protocols for Live Video Streaming in P2P Systems," in *2010 IEEE International Conference on Communications*, May May 2010, pp. 1–5.

[29] C. Kiraly, L. Abeni, and R. Lo Cigno, "Effects of P2P Streaming on Video Quality," in *2010 IEEE International Conference on Communications*, May May 2010, pp. 1–5.

[30] M. Handley, V. Jacobson, and C. Perkins, "Sdp: Session description protocol," RFC 4566 (Proposed Standard), Internet Engineering Task Force, Tech. Rep. 4566, jul 2006.

[31] L. Baldesi, L. Maccari, and R. Lo Cigno, "On the Use of Eigenvector Centrality for Cooperative Streaming," *IEEE Communications Letters, Vol. 21, Issue. 9, pages 1953–1956,*, Dec. 2017. http://ieeexplore.ieee.org/document/7942071/



netCommons

Network Infrastructure as Commons

Release of New Open Source Software for All Applications

Deliverable Number D3.4
Version 1.0
March 2, 2018

netCommons.eu