Horizon 2020

Project title: Network Infrastructure as Commons

Release of Open Source Software for all Applications (Companion Document)

Deliverable number: D3.2

Version 1.0

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

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:	Other (O)
Dissemination level:	Public (PU)
Contractual Delivery Date:	December 31, 2016
Actual Delivery Date	December 31, 2016
Number of pages:	57 excluding covers
Keywords:	Software, Applications, Open Source, Cloudy, PeerStreamer, CommonTasker
Authors:	Nicolo Facchi - UniTn Felix Freitag - UPC Merkouris Karaliopoulos - AUEB-RC Iordanis Koutsopoulos - AUEB-RC Leonardo Maccari - UniTn Panagiota Micholia - AUEB-RC Aristide Pilichos - AUEB-RC Francesco Zanini - UniTn
Peer review:	Leandro Navarro - UPC Renato Lo Cigno - UniTn

Executive Summary

This Deliverable contains the high level description of the open source applications developed in the first year of netCommons. Three open source applications have been produced: a new version of the Cloudy distribution, a new version of the PeerStreamer Peer-to-Peer (P2P) video streaming software and a mobile app that uses community networks to crowdsource activities. The Cloudy distribution has been "dockerized"; namely, the many applications it supports run in Docker containers. This makes it easier to extend and migrate them, but also opens the way to new applications and simplifies their adaptation to the specific needs of a community network. The PeerStreamer application was split in two parts, one containing the P2P engine that can be installed directly on network routers, and another serving as the front-end to video playback. Finally, the first version of a third application, built from scratch within the netCommons project, has become available. The CommonTasker application implements mobile crowdsourcing and sharing economy principles to address specific needs of the communities behind the networks and, thus, generate added value for the networks themselves. Much effort during this first year was devoted to application design iterations, which took advantage of the close interactions with the Sarantaporo.gr team.

This document explains in detail the developments undertaken for the already existing software (Cloudy, PeerStreamer), as well as the design decisions made and the development work undertaken for the new software (CommonTasker). It also points to the public repositories that store the source code make it accessible to third parties.

The code itself, which is the main body of the deliverable, is available both as compressed archives together with this document and on the github repository of netCommons as tagged versions of the relative projects:
https://github.com/netCommonsEU

Contents

1 Introduction 1

2 The Cloudy Distribution 3

2.1 Introduction 3
2.2 Community network clouds 4
2.3 Deployed community network cloud 4
2.3.1 The Guifi.net community cloud 4
2.3.2 The Cloudy distribution 5
2.3.3 Service performance 7
2.4 Operation and analysis 7
2.4.1 Service availability 7
2.4.2 User participation 7
2.4.3 Final Remarks 8
2.5 Docker integration in Cloudy 9
2.5.1 Experiments with the deployment of collaborative services 11
2.5.2 Next steps in Docker integration 13
2.6 Service placement 13
2.6.1 Motivation and goals 14
2.6.2 Placement algorithm 14
2.6.3 Network behaviour and algorithm performance 15
2.6.4 Deployment in a real production Community Network 17
2.7 Takeaway 18

3 The PeerStreamer Platform 21

3.1 Peer-to-peer networks and PeerStreamer 21
3.1.1 PeerStreamer Architecture and Background 22
3.2 PeerViewer: a Web-enabled media player 23
3.3 First approach: WebRTC technologies 24
3.3.1 WebRTC Concepts 24
3.3.2 Implementing a WebRTC frontend 25
3.4 An HTML5-based approach 26
3.4.1 Architecture design 26
3.4.2 Front end user interface 28
3.5 Source Code 31

4 The CommonTasker Mobile App 33

4.1 Mobile crowdsourcing and sharing economy for CNs 33
4.2 Architecture 34
4.2.1 The client app 35
4.2.2 The backend server 36
4.3 CommonTasker for Sarantaporo.gr 37

4.3.1 Feedback from the Sarantaporo.gr team - steps 1 and 2 37
4.3.2 Feedback from the local community and future plans - step 3 38
4.4 First release of the CommonTasker software 38
4.4.1 Client capabilities 39
4.4.2 Firebase 40
4.4.3 Libraries 41
4.5 Source code and testing 43

5 Conclusions and Future Work 45

List of Figures

2.1 Community cloud in Guifi.net running the Cloudy distribution on diverse hardware. . 5
2.2 Cloudy distribution Web user interface. 6
2.3 Long-term monitoring of Cloudy instances and published services. 8
2.4 Enterprise menu to enable Docker-based services. 10
2.5 A list of Docker-based applications available in a Cloudy instance. 10
2.6 With *Docker form* the Cloudy administrator configures an ownCloud container installation. 11
2.7 ownCloud application started in Cloudy as a Docker container. 12
2.8 A Cloudy user finds four service provider publishing Docker-based applications. . . 13
2.9 ECDF of node availability 16
2.10 ECDF of node latency 16
2.11 PASP-Availability 16
2.12 PASP-Latency 16
2.13 PASP-Closeness 16
2.14 QMP topology 18
2.15 Average read time on clients' side 18

3.1 A scheme of the PeerStreamer architecture reproduced from [1]. 22
3.2 PeerViewer architecture diagram 27

4.1 Client and back end side of the application in a wireless CN. 35
4.2 CommonTasker architecture. 35
4.3 User-app interface: introducing and welcoming users to CommonTasker. 39
4.4 Create and perform errands. 40
4.5 Ask and answer. 41
4.6 Objects. 42

List of Acronyms

AAA	Authentication, Authorization, and Accounting
API	Application Programming Interface
CN	Community Network
CSS	Cascading Style Sheets
DOM	Document Object Model
DTLS	Datagram Transport Layer Security
GRAPES	Generic Resource-Aware P2P Environment for Streaming
HTML5	HyperText Markup Language 5
HTTP	Hypertext Transfer Protocol
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
JSX	JavaScript eXtended
ML	Messaging Layer
NAT	Network Address Translation
NPM	Node Package Manager
P2P	Peer-to-Peer
RTCP	Real Time Control Protocol
RTP	Real-time Transport Protocol
SDP	Session Description Protocol
SRTP	Secure Real-time Transport Protocol
SSL	Secure Sockets Layer
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TURN	Traversal Using Relays around NAT
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WCN	Wireless Community Network
WP3	Work-Package 3
WebRTC	Web Real-Time Communication
ES6	ECMAScript 6

1 Introduction

Applications tailored to work efficiently into Community Networks (CNs), without dependence on Internet connectivity, are broadly considered a key requirement for the success of CNs. One of the goals of netCommons is exactly to produce applications that generate added value for **CNs!** (**CNs!**). Work-Package 3 (WP3) is devoted to the development of ”local” applications that can enhance the value of a CN so that people use them not only as alternative access networks to the Internet but also for their rich local service offerings.

The main body of Deliverable 3.2 consists of the open source code freely available in the netCommons github repository[1]. This companion document briefly explains the goals underlying the development of each application and the advances made over their previous releases. WP3 has three separate tasks (T3.2, T3.3 and T3.4) that develop respectively a software platform for CN nodes, a video streaming application, and a mobile application to coordinate the crowdsourcing of tasks available by, and for, the local communities. The fourth task of WP3, Task 3.1, is instead devoted to a more generic goal: the development of a methodology to tightly engage developers, researchers, and CN members in the participatory creation of their own applications.

This Task reports in separate deliverables, namely D3.1 at M12 and its follow-up D3.3 at M24. Clearly, there are overlaps between this task and the software development tasks in WP3. However, and since tasks T3.2 and T3.3 build on existing software, the initial effort and goals have been in some sense already defined. Thus, in these first months of development, the influence of T3.1 on these two tasks has not been very large. Differently, the aim of the third application in T3.4 is the design of a novel application from scratch according to the needs of a specific community. Hence, the app development serves as a case study, where the participatory design work in T3.1 is put into practice. Whereas this is extremely valuable for the progress of the participatory design work, the software development loops become longer since they need to collect and respond to the feedback from the community.

Task 3.2 takes care of **Cloudy**. The **Cloudy** GNU/Linux distribution is a software distribution that has been designed to host local applications inside a CN. It was first released as the product of the CLOMMUNITY FP7 project and it is now further developed in T3.2. One of its limitations by the end of CLOMMUNITY project was the complexity in adding and maintaining new applications in the bundle of applications present in and supported by the distribution. T3.2 further developed Cloudy in order to “dockerize[2]” all the applications so that now every application runs in a separate docker container. This eases the addition, removal and management of new applications and thus makes it easier for communities to use Cloudy in their own network environment to support specific applications they need. Moreover, the first year work in T3.2 includes an experimental analysis of the network service placement problem in the context of CNs.

Task 3.3 is dedicated to Peer-to-Peer (P2P) video streaming. In particular T3.3 has selected **PeerStreamer** as the base platform for further development. PeerStreamer is a P2P live video streaming

[1]See https://github.com/netCommonsEU

[2]For information and details on the docker Platforms see https://www.docker.com/; the key idea of docker is the isolation of each application so that interactions between different applications are kept to a minimum and maintenance can be done separately without affecting the other applications that can remain active and running

platform developed in the EU FP7 NAPA-WINE project[3]. T3.3 further develops PeerStreamer with the goal to ship PeerStreamer with the OpenWRT/LEDE operating system that is used by thousands of nodes in many CNs and make it available as a standard application for any CN. To achieve this, T3.3 splits the PeerStreamer P2P engine (that is lightweight and can be installed in low-power devices) from the playback application, which is instead heavier to run and calls for user terminals.

Finally, T3.4 deals with the development of the **CommonTasker** mobile application. Herein, we report on a preliminary release of this new application, whose goal is to crowdsource tasks in a CN. The general idea is that CN users can use the application to jointly organize events and actions, exchange knowledge and tips on their work, get help with manual tasks, and even share tools and machinery with each other. The application is meant to work over CNs without any support from public Internet infrastructure and serve two purposes: (a) become an online assistant for getting help with many daily tasks of the community, including network-related maintenance tasks; (b) strengthen the community links and cooperation values among the network participants. Both purposes are important for the sustainability question faced by all CNs. The first release of the application contains the description of the app, the design choices that came out of the participatory design process so far, and the first skeleton of the application with basic functionality (e.g., submit a new task, browse ongoing tasks, announce an event etc). Since the participatory design process is still on-going, the current version of the app is a prototype that demonstrates the different capabilities offered by the technology. Then, the selection of the most important functionalities that will address specific needs of our target community, the farmers and breeders in the Sarantaporo valley, will be subject to the outcomes of the participatory design process. The lessons learned from our first visit to the area are described in detail in D3.1. In the following months Tasks 3.1 and 3.4 will be in close collaboration to define the next steps in terms of the development of the "real" application, tailored for the Sarantaporo.gr Community Network.

[3]After the end of NAPA-WINE PeerStreamer has been maintained as an Open Source P2P video streaming platform `http://peerstreamer.org`, and it has been used in several research works [1, 2, 3]

2 The Cloudy Distribution

This Chapter is devoted to the work on community clouds as part of the work of WP3 in netCommons, specifically task T3.2. The work on community clouds has not started from scratch, but builds upon the results achieved by the European research project CLOMMUNITY (2013-15)[1]. A description of the status of the deployment of this community cloud by the end of that project in June 2015 can be found in [4]. Other relevant publications with more details on the different types of results achieved by the CLOMMUNITY project are highlighted in the project's publications section.

The work that lead to the software delivery and that presented in this Chapter is based on the following publications and extends them further

1. "Towards Network-Aware Service Placement in Community Network Micro-Clouds", M. Selimi, D. Vega, F. Freitag, and L. Veiga, Proc. of the *22nd International European Conference on Parallel and Distributed Computing (EuroPar 2016)*, August 2016.
2. "Bandwidth-aware Service Placement in Community Network Clouds", M. Selimi, L. Cerdà-Alabern, L. Wang, A. Sathiaseelan, L. Veiga, and F. Freitag, Prof. of the 41st Annual IEEE Conference on Local Computer Networks (LCN 2016), November 2016.
3. "CLOUDY: Community Cloud Edge Distribution in Guifi.net", F. Freitag and M. Selimi, Demo at the *41st Annual IEEE Conference on Local Computer Networks (LCN 2016)*, November 2016.

2.1 Introduction

Cloudy is a community cloud platform which resulted from the CLOMMUNITY research project. Afterwards, it has grown independent from the project, and it started living as an Open Source community effort without being tied to a specific project., therefore it is hosted separately[2] and continues to be maintained on the original Github repository. Within netCommons we have set up an additional Github repository[3] to enable working with project-specific developments of Cloudy.

This Chapter is organized as follows. Dection 2.2 introduces the concept of community clouds, and Section 2.3 presents the deployment of the community cloud in guifi.net, and analyses the results of its deployment (2.4). Section 2.5 discusses the integration of Docker containers into a community cloud, and it shows how the Docker extension enables users to provide customized services. We experiment in Section 2.5.1 with the achieved features to provide personal and shared services for a collaborative community cloud. Section 2.5.2 outlines next steps. In a second part corresponding to the first and second publication we explain in Section 2.6 our work conducted to gain insights and understanding for improving service placements in a community network cloud. We describe our goals, the solution we proposed and its evaluation. Section 2.7 draws some initial conclusions and sketches future work.

[1]http://clommunity-project.eu/
[2]http://cloudy.community/
[3]https://github.com/netCommonsEU/Cloudy-netCommons

2.2 Community network clouds

Community networks are a communication infrastructure model in which local communities of citizens build, operate and own open IP-based networks. Community networks often originated for providing Internet access to the population of areas which were ignored by commercial telecom operators. These networks are set up through a collective effort, using off-the-shelf equipment for wireless communication between nodes, and are maintained by the contributions of time and knowledge the communication network members [5, 6].

A community cloud is a cloud deployment model in which a cloud infrastructure is built and provisioned for use by a specific community of consumers with shared concerns, goals and interests. It is owned and managed by the community or by a third party or a combination of both [7]. A community cloud offers features that are tailored to the needs of the specific community it addresses. The difference between one community cloud and another is that the provision of certain features, e.g. performance, security, ease of usage of the cloud, are emphasized. Community clouds exist today for different industry sectors and are commercially operated.

The community cloud we address is a cloud hosted on community-owned computing and communication resources providing services of local interest. It is a particular case of a community cloud, a cloud for community wireless networks, tailored to the specific requirements and conditions of such a community and adapted to the technical challenges of these decentralised and WiFi based networks. We leverage the concept of community clouds, but focus on a community cloud that is collectively built and maintained by citizens. While the foundational elements of such a community cloud solution have been discussed before [8], deployed implementations have just started only recently.

2.3 Deployed community network cloud

2.3.1 The Guifi.net community cloud

The Guifi.net community cloud consists of distributed heterogeneous computing devices contributed by participants that run the Cloudy software [9]. This community cloud is heterogeneous since the users can contribute to the cloud with many types of hardware. Often, these are inexpensive devices that can be classified as mini-PCs with low energy consumption to operate in 24/7 mode. The infrastructure is distributed, since most of the contributed devices are located at the users' premises (e.g. inside the home, garage, storage or service rooms in buildings). In addition, some (more powerful) hardware may be located at municipality installations together with other city infrastructures (e.g. local data-centers, warehouses, street cabinets).

Figure 2.1 illustrates the Guifi.net community cloud. Different hardware devices are used as cloud nodes. The Cloudy distribution is installed on each of the devices connected to the community cloud. The cloud nodes are spread over the community networks and on different geographic locations. Desktop PCs are mainly in municipality buildings, most mini-PCs are at user premises.

The cloud infrastructure in Guifi.net is not bound to a certain number of devices, but grows organically with every new node that community members add to the infrastructure. A similar situation holds for services. While there is a pre-configured set of services integrated in the Cloudy distribution, this community cloud is open to provide new services. The range of services built over that infrastructure targets mainly platform and application oriented services, where specialization and customization of services has a higher potential.

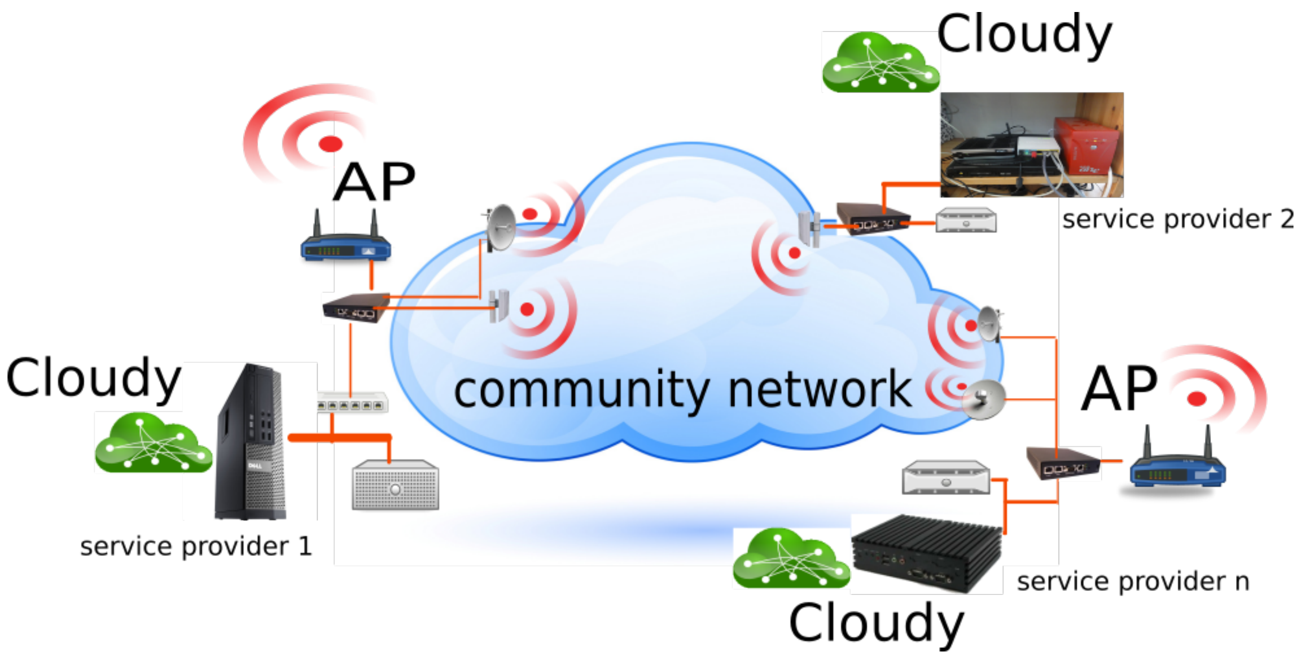


Figure 2.1: Community cloud in Guifi.net running the Cloudy distribution on diverse hardware.

2.3.2 The Cloudy distribution

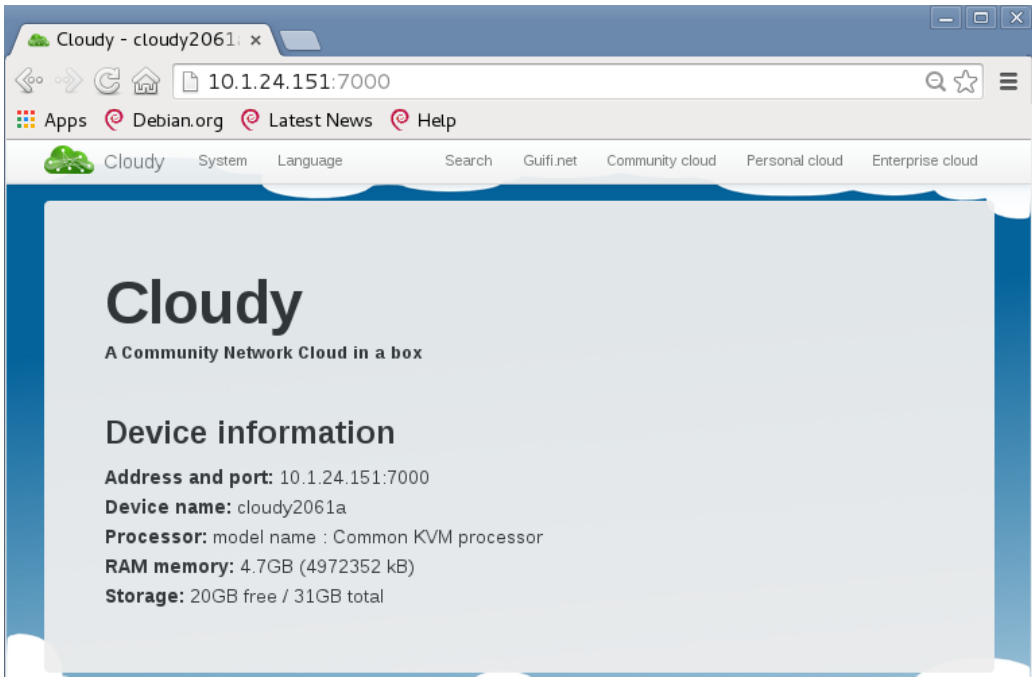
The Cloudy distribution was developed to enable community network clouds[4]. Cloudy is based on Debian GNU/Linux. It's software can be freely downloaded from public repositories[5]. The recommended way to install Cloudy is to run the *cloudynitzar.sh* script[6] on a freshly installed Debian distribution.

Users of the Guifi.net community cloud are expected to install this distribution on contributed cloud nodes. Making Cloudy the default system for community network clouds ensures homogeneity in terms of a basic set of common services, which are needed for every participant to join and interact in the community cloud.

Figure 2.2 shows the Cloudy web user interface after installation at the user's device. The Cloudy distribution provides a set of service categories, grouped as Search, Guifi.net, Community cloud, Personal cloud and Enterprise cloud.

The *Search* service allows users to find the Cloudy instances that are part of the community cloud, and to discover the public services deployed by the other Cloudy instances. It also makes it possible for a new Cloudy node to be discovered from other nodes in the community cloud. The search service is implemented through Serf[7]. Serf uses gossiping to distribute information among its network of nodes. Once a user installs Cloudy, he/she can activate the Serf daemon, which will connect to the Serf bootstrap server and thus integrate the new Cloudy instance in the Serf overlay network. One or more bootstrap server need to be hosted in the community network. The Serf daemon on each Cloudy node continuously updates each other about the currently published services and connected nodes. This way, at each Cloudy instance the user has an up-to-date view on the available services and nodes

[4]For the principles of the development and its goals see the FP7 EU project "A Community networking Cloud in a box – CLOMMUNITY", http://clommunity-project.eu
[5]http://cloudy.community/download/
[6]https://github.com/Clommunity/cloudynitzar
[7]https://www.serf.io/



Cloudy

A Community Network Cloud in a box

Device information

Address and port: 10.1.24.151:7000
Device name: cloudy2061a
Processor: model name : Common KVM processor
RAM memory: 4.7GB (4972352 kB)
Storage: 20GB free / 31GB total

Figure 2.2: Cloudy distribution Web user interface.

in the community cloud.

The *Guifi.net services* in Cloudy allow users to install a set of community network management services. The following three Guifi.net services have been integrated in Cloudy:

- DNS Service to participate in the guifi.net DNS system for the resolution of internal addresses, built with BIND.
- Network monitoring instance to contribute to the network monitoring system, built with SNMP feeding RRDtool buffer rings.
- Web proxy as part of hundreds of Internet gateways contributed by volunteers. That way any validated user can access any of the federated web proxies for Internet service. The service is based on the Squid proxy.

Community cloud services consist currently of a set of distributed applications that run on the Cloudy nodes. They are already available on the Cloudy nodes (although the user is free to decide if she wants to activate them). The following applications are currently integrated in Cloudy:

- PeerStreamer: A peer-to-peer media streaming framework with a streaming engine for the efficient distribution of media streams, a source application for the creation of channels, and a player application to visualize streams. This application is further described in Chapter 3.
- Tahoe-LAFS: A fault-tolerant encrypted decentralized cloud storage system which distributes user data across multiple servers in replicated data chunks. Even if some of the servers fail or are taken over by an attacker, the entire file store continues to function correctly while preserving user's privacy and security.
- WebDAV server: A set of extensions to the HTTP protocol which allows users to collaboratively edit and manage files on remote web servers. Implemented with the Apache Web server DAV module.

Personal cloud services refer to specific applications for the user, which may not necessarily be shared with the community, different to the community cloud services. The current personal service consists of one application:

- Syncthing: A decentralised cloud storage system with cryptographic features which gives users full control over where their data is replicated and shared with a group.

Enterprise cloud refers to services which the user may install to enable third-party applications (which may be for-profit or not). These services are installed as Docker containers. We describe the Docker integration in detail in section 2.5.

2.3.3 Service performance

Cloudy is an open platform to deploy services in community networks. As an open platform, it offers both a set of pre-installed applications which the user can activate through Cloudy's web interface, and it allows users to install additional services, either for personal use, or for shared usage by the community (this part is elaborated in detail in sections 2.5 and 2.5.1).

Regarding the pre-installed applications, i.e., those that are shipped as part of Cloudy, before integrating them in the Cloudy distribution, we conducted several experiments on their performance in the community network, to make sure that the user experience they achieve is acceptable. We published the results, in particular for Tahoe-LAFS, PeerStreamer and the search service, in [10, 11, 12]. These evaluations included the deployment of the applications under realistic conditions on several nodes in the community network.

2.4 Operation and analysis

2.4.1 Service availability

We measure the Cloudy instances deployed and services provided. The values of this metric are obtained through long-term monitoring of the service availability in the community cloud seen from a specific Cloudy instance. Instantaneous values can be obtained at any moment from a publicly available Cloudy instance[8].

Figure 2.3 shows the evolution of the number of Cloudy instances and services during two and a half months (from May 2016 to July 2016). The availability of the main services of Cloudy are shown. Roughly three groups of services can be seen, which are identified as follows: the group at the top corresponds to the services named SERF and On-line represent the number of active Cloudy instances. While in the measured time frame the values go up to a maximum of 45 and down to a minimum of 15, we can estimate that on averages around 30 Cloudy instances are online.

The next group in the middle corresponds to the Guifi.net services (i.e., Proxy, Graph server, DNS). We can observe a fairly stable availability of these services with around 10 services instances of each on average.

Finally, at the bottom of Figure 2.3, the lowest graph corresponds to end user-related services. Syncthing is the most numerous with around 10 instances on average.

2.4.2 User participation

Overall user take-up: The search service discovered over the measured period around 30 on-line Cloudy instances on average. While this absolute number is difficult to judge as success or not for

[8]http://demo.cloudy.community User: *guest*, Password: *guest*

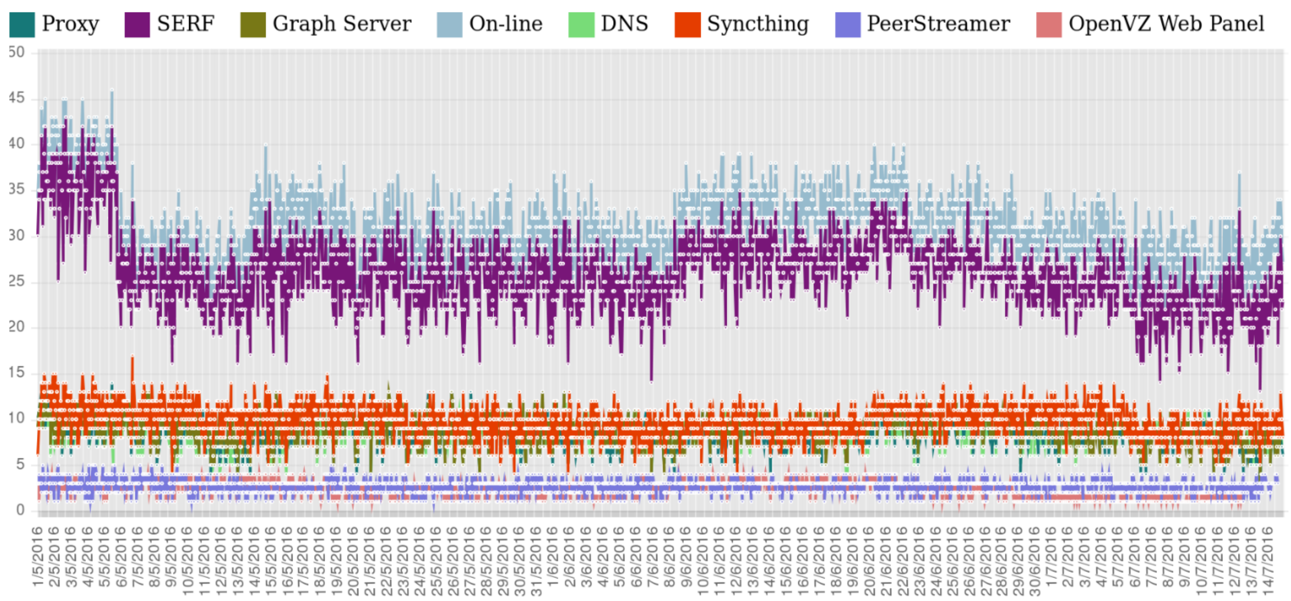


Figure 2.3: Long-term monitoring of Cloudy instances and published services.

a volunteer-driven community cloud system, it nevertheless indicates that the community cloud has achieved to be self-maintained and operated by volunteers and members of the community network. In addition, the Cloudy distribution demonstrated to be stable enough to allow such a permanent operation of the services.

After more than one year of operation, the mailing list subscription by July 2016 consists of 64 users for the cloudy-users mailing list and 31 users for the Cloudy development mailing list. On a deeper analysis of the messages exchanged during one year, we observed several months of significant usage of the mailing lists with interchange and discussion of experiences, especially in the cloudy-users list, which hints to a positive acceptance of this tool by the community network members.

Community service take-up: It could be seen that the Syncthing service is the most popular of the community services. This might be due to the fact that its folder synchronization capability is a known feature also available in commercial services, e.g., Dropbox. The Peerstreamer and Tahoe-LAFS services, however, have not found a significant user take-up. Live streaming though Peerstreamer, although it has been used to stream-cast a few community events within the Guifi.net network, seems not to have found a permanent usage beyond that. The experience and lessons learned have motivated the improvement of the service described in Chapter 3. The specific features of the Tahoe-LAFS storage service, e.g., resilience through fault-tolerant storage and privacy via user-side encryption, do not seem to raise in the community enough interest to spawn a strong take-up of this service.

Guifi.net services: Guifi.net services are not end-user services, but help to manage the network and enable Internet access through the guifi-proxy service. The possibility to activate these services depends not only on the willingness of the user, but also on the type of device on which Cloudy is installed, and the type of network node the users has. Therefore, we consider the achieved take-up of the Guifi services has been satisfactory.

2.4.3 Final Remarks

This long-term analysis of the services in the community cloud has shown the willingness of the community members to host Cloudy devices. It can be observed that users were particularly interested in activating the services related to network management. The pre-installed end-user services, however, were less used. The obtained results and the discussion with community members about opportunities for collaboration and local content sharing among participants suggests the opportunity to improve Cloudy in the end user service offer.

As a consequence, we envision the capability to offer a higher level of service personalization and customization as an opportunity for this community cloud. As will be described in the following sections, the approach chosen was to integrate Docker containers in Cloudy.

2.5 Docker integration in Cloudy

Service personalization can allow the Cloudy service offer to adapt to the individual needs of the users. It is likely that a user will contribute a device as Cloudy host to the community cloud if she is able to run community *and* personal services [13]. Given that Cloudy was conceived as an open platform, this openness should support users to install their own applications, have the flexibility to choose which kind of services they want to run, and which ones of them to run for personal use, and which ones to share with the community.

We have chosen Docker as the technology to enable support for service personalization and customized service deployments. Docker has recently gained strong interest from research and industry, and major industries have started promoting container technology for cloud computing[9].

The Docker tools provide many convenient features that makes the use of containers in Linux more easy, which in turn has contributed to increase the community of Linux container users. Still, Docker is not meant for usage by unskilled users. Rather the software developer and IT experts, the so called Device Operators (DevOps), of the community has become an important target group of Docker users.

Cloudy however is operated by end users, with different levels of technical skills. Therefore, an important requirement for Docker in Cloudy is that the integration of the control and configuration of Docker is such that the average end-user is able to install and run Docker containers. While a technically skilled person such as a DevOps engineer may run Docker using command lines in a terminal, our goals is to have Docker easily used by an average end user. Therefore, Cloudy must provide an end-user friendly interface. The approach we followed is to hide the technical complexity of the Docker usage behind graphical web interfaces in Cloudy, more suitable for being managed by end users.

Figure 2.4 shows the Docker extension developed for the Cloudy GUI. It allows the Cloudy administrator to enable (install) Docker with a single click in the web interface. Once installed, within the *Enterprise cloud* service category, the sub-menus *Docker form* and *Docker* become available.

In the *Docker* sub-menu the Cloudy administrator can find a list of pre-configured applications available that can be started as Docker containers (Figure 2.5). These applications can be installed and uninstalled with a click on the web interface. The user can also choose if these applications should be published (shared) to the community network cloud. Applications kept private will not be published, so the application will only be known to the user and therefore can be also personalized. If the application is published, then it will be discovered by the search service at other Cloudy instances and therefore it becomes known in the community. This is how any member can contribute and share new applications with the community.

In the *Docker form* sub-menu, the Cloudy administrator of a node can install new services through

[9]Cloud Native Computing Foundation https://cncf.io/

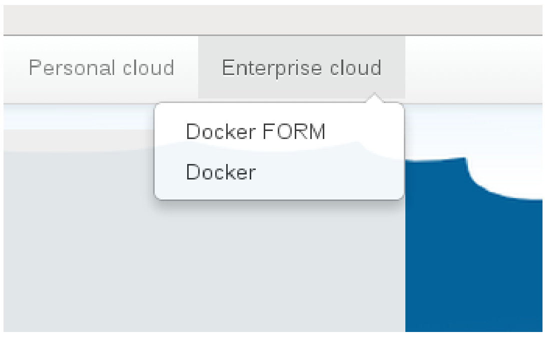
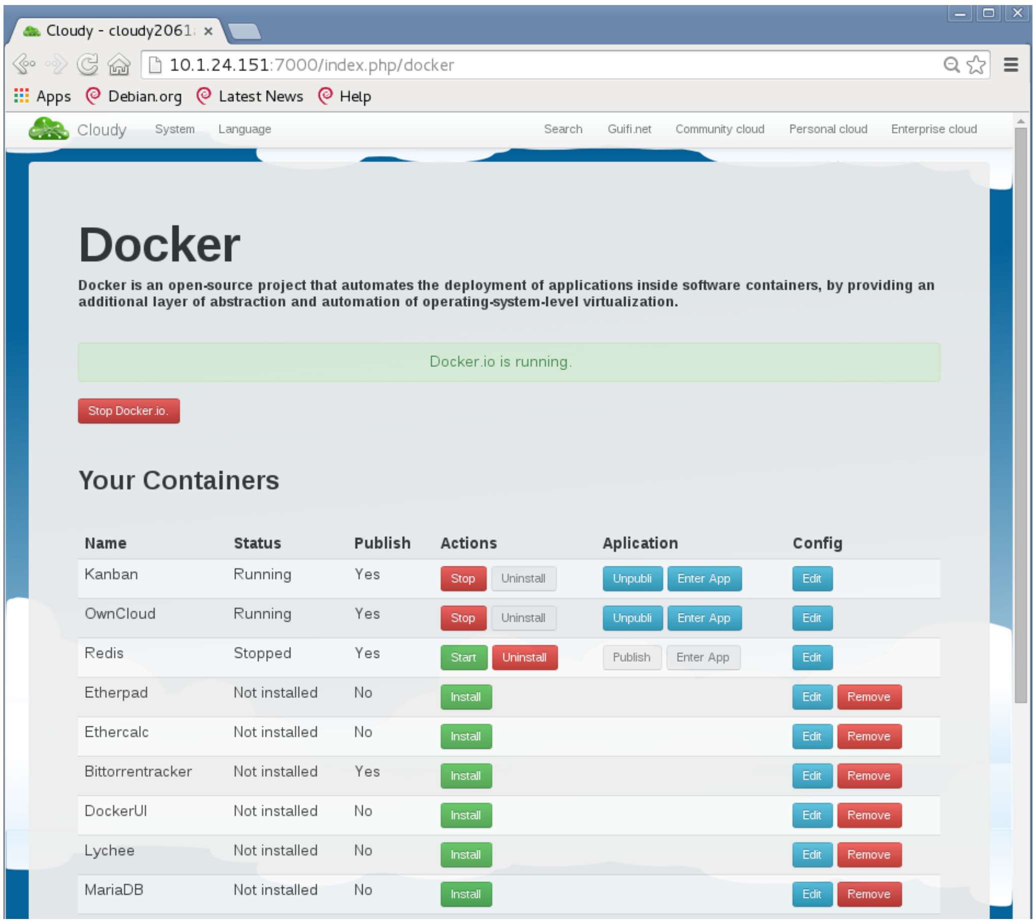


Figure 2.4: Enterprise menu to enable Docker-based services.


Cloudy - cloudy2061 ×

10.1.24.151:7000/index.php/docker

Apps Debian.org Latest News Help

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

Docker

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization.

Docker.io is running.

Stop Docker.io.

Your Containers

Name	Status	Publish	Actions	Aplication	Config
Kanban	Running	Yes	Stop Uninstall	Unpubli Enter App	Edit
OwnCloud	Running	Yes	Stop Uninstall	Unpubli Enter App	Edit
Redis	Stopped	Yes	Start Uninstall	Publish Enter App	Edit
Etherpad	Not installed	No	Install		Edit Remove
Ethercalc	Not installed	No	Install		Edit Remove
Bittorrentracker	Not installed	Yes	Install		Edit Remove
DockerUI	Not installed	No	Install		Edit Remove
Lychee	Not installed	No	Install		Edit Remove
MariaDB	Not installed	No	Install		Edit Remove

Figure 2.5: A list of Docker-based applications available in a Cloudy instance.

Docker containers. Figure 2.6 shows an example of how this form is configured to install the ownCloud application as a Docker container. Principally, any Docker image can be specified and its deployment configured. Currently, the specified images are downloaded from Dockerhub[10].

The Docker integration in Cloudy is still evolving and may benefit in the future from additional tools becoming available as results from the standardization efforts recently started around the Docker ecosystem[11]. However, it can already be seen that a Cloudy device in the community cloud is able to deploy, on behalf of third-parties, additional services that are provided as Docker images. This is

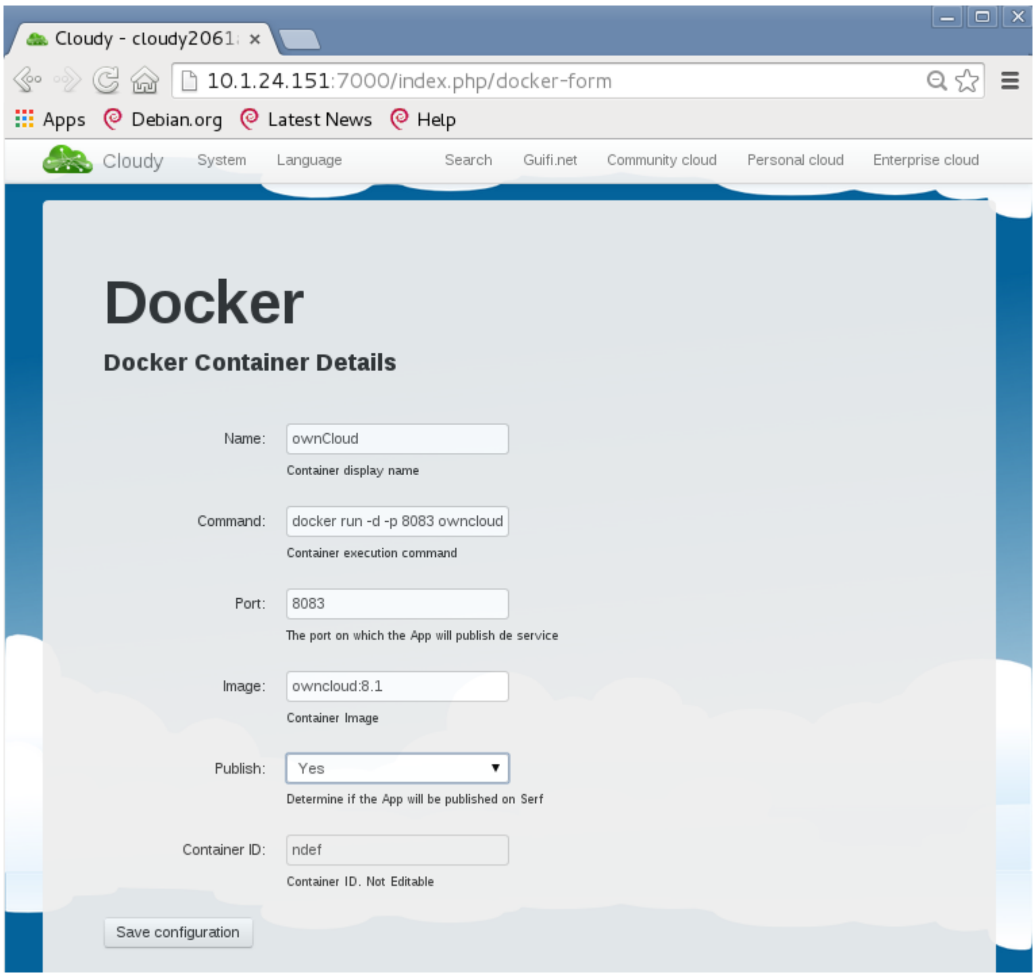
[10]https://hub.docker.com/
[11]Cloud Native Computing Foundation https://cncf.io/

an important feature for enabling new cloud-based services at the network edge. Cloudy users could support with their devices the deployment and operation of commercial containerized services.

2.5.1 Experiments with the deployment of collaborative services

We consider the scenario when a Cloudy user is willing to install and run a new application in its Cloudy node and publish it to the other Cloudy users. The steps which the user has to conduct are installing and deploying the Docker container of the application in Cloudy and publishing this new application, thus it will be know to the other Cloudy users. In the following we carry out and explain these steps in detail by installing *ownCloud*, an open-source application with some similar features as the popular Dropbox service.

Figure 2.6 shows the first step a user has to take, which is introducing some information about the ownCloud Docker container. For this the user has opened the *Docker FORM* item, which is available in the *Enterprise cloud* menu as shown in Figure 2.4. The content of these fields provide the information needed for Cloudy to obtain the Docker image and run the container. This information is typically provided in the Docker Hub along with the description of the Docker image, and it can be copied by the user into the fields of the form. Once completed, saving the configuration will make the new application appear in the list of applications ready to be installed, as shown in Figure 2.5.



Docker

Docker Container Details

Name:	ownCloud Container display name
Command:	docker run -d -p 8083 owncloud Container execution command
Port:	8083 The port on which the App will publish de service
Image:	owncloud:8.1 Container Image
Publish:	Yes Determine if the App will be published on Serf
Container ID:	ndef Container ID. Not Editable

Figure 2.6: With *Docker form* the Cloudy administrator configures an ownCloud container installation.

Installation: If the user wants to run the application, then he/she clicks in the column *Action* (see Figure 2.5) to *install* the application. Cloudy runs the previously introduced configuration instructions to install the image. The output of the detailed steps, e.g. downloading the Docker image if not available locally, is hidden from the end user, and the end user is notified once the installation has finished. At this point, the application is ready to be used.

Starting the application: In the column *Action* the button *Start* is provided to run the newly installed Docker image as container. Once started, the Start button transforms into a Stop button. This way the user can easily switch between starting and stopping the application with a click in Cloudy's web interface.

Accessing the application: For accessing a started applications, the user clicks on the *View app* button of the column *Application* as shown in Figure 2.8. Cloudy thus connects the user's browser to the URL of the application, enabling the application to be used such as shown in Figure 2.7 for the the described ownCloud case.

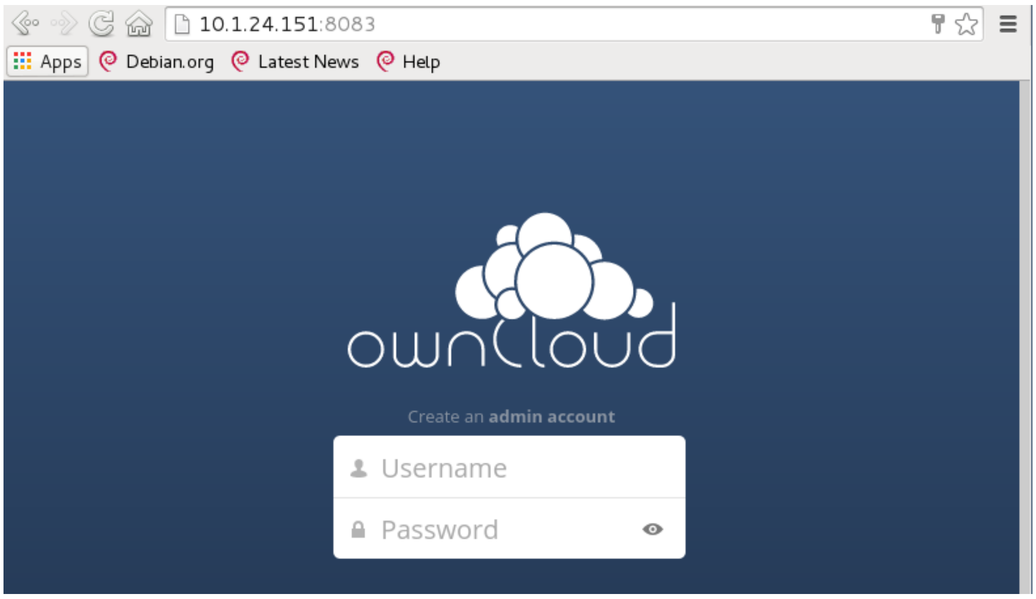
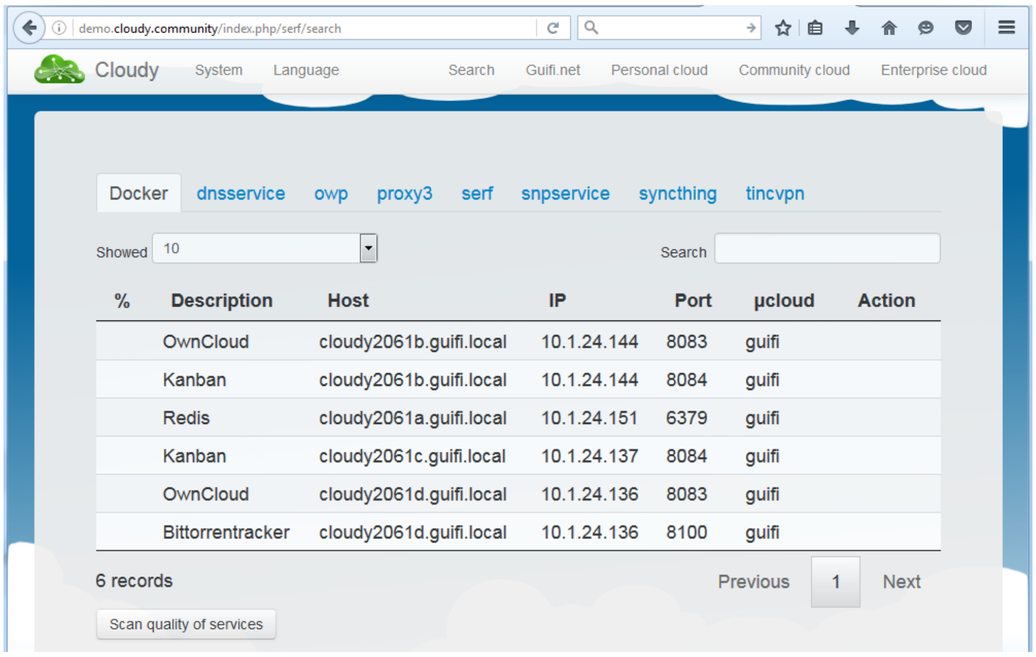


Figure 2.7: ownCloud application started in Cloudy as a Docker container.

Sharing the application: As part of the the service personalization implementation in Cloudy, the applications running in a Cloudy instance can be kept private for personal use or made public, i.e. are published. Publishing the applications makes them known to other users, and this way the usage of the application may be shared among them. If the user wants to make for instance its ownCloud application public, then he/she activates this option by clicking on "Yes" in the *Publish* column. This step results in the Cloudy Serf daemon to propagate the information about the new application in the messages it gossips among the Cloudy nodes.

Figure 2.8 shows how a Cloudy user using the *Search service* finds several applications deployed as Docker containers in the cloud, which are shared by their owners within the community. We can see four different applications that are found, i.e. ownCloud, Kanban, Redis, Bittorrentracker, installed as Docker containers on other Cloudy instances, of a total of six shared applications. It can be seen that for two of these applications, i.e. ownCloud and Kanban, there are two instances published with different IPs, corresponding to instances of different owners. In this case, an user interested in using this application could choose between the two providers. Finally, four different IPs are in the list of found applications, meaning that there are four Cloudy instances acting as service providers.



demo.cloudy.community/index.php/serf/search

%	Description	Host	IP	Port	µcloud	Action
	OwnCloud	cloudy2061b.guifi.local	10.1.24.144	8083	guifi	
	Kanban	cloudy2061b.guifi.local	10.1.24.144	8084	guifi	
	Redis	cloudy2061a.guifi.local	10.1.24.151	6379	guifi	
	Kanban	cloudy2061c.guifi.local	10.1.24.137	8084	guifi	
	OwnCloud	cloudy2061d.guifi.local	10.1.24.136	8083	guifi	
	Bittorrentracker	cloudy2061d.guifi.local	10.1.24.136	8100	guifi	

6 records

Figure 2.8: A Cloudy user finds four service provider publishing Docker-based applications.

2.5.2 Next steps in Docker integration

The attractiveness of services at the network edge is key for motivating users and encouraging volunteer activity. As we reported in section 2.3, our analysis of the initially pre-installed end-user services of Cloudy showed low user interest, which motivated us to develop the Docker integration.

From further tests with service deployments with Docker in Cloudy, we have identified multi-service deployments as an issue which may be considered to be addressed further: Cloudy currently applies the *docker run* command for service provision. With this approach, a single Docker container is deployed and it works well for single independent services/applications. Complex applications however, which depend on the availability of other services in the system, for instance the availability of a database, have difficulties with such approach. Either a Docker file is needed which already describes these dependences to deploy everything in a single container, or a sequence of containers is needed to be deployed (difficult to keep simple for end users), or an additional module needs to be developed to manage and solve in the background these dependencies for the user. As a solution we consider is the integration in Cloudy of *docker-compose* for running these applications with dependencies on several Docker containers.

2.6 Service placement

The physical location where servers (and the services they support) are placed in a network has a huge impact on the service performance and on the load the services impose on the underlying network. In order to understand better the options and design for service placement CNs, we conducted several experiments in Guifi.net measuring several performance metrics as service availability, latency, closeness, which are fundamental for QUality of Experience (QoE). Thus, this work addresses a long-term perspective of Cloudy software development in the sense that its results will contribute to the design of better software and installations scripts that will enhance the QoE of services in the community network cloud. We focus in the following on highlighting a set of aspects and results fundamental for the Cludy development, while additional details con be found in the related publication [14].

2.6.1 Motivation and goals

Since Guifi.net nodes and the connected servers are geographically distributed, addressing QoE on the long term in this community network cloud requires to be decide where services should be placed in a network. If the underlying network resources are not taken into account, a service may suffer from poor performance, e.g, by sending large amounts of data across slow wireless links while faster and more reliable links remain underutilized. Therefore, a key challenge in community network micro-clouds is to determine the location of the service deployments, i.e. which servers to place at a certain geographic points in the network. Due to the dynamic nature of community networks and usage patterns, it is challenging to calculate an optimal placement.

The key goal is understanding the impact of network-aware service placement decisions on end-to-end client performance. The main contributions can be summarized as follows:

- We introduce a service placement algorithm that provides optimal service overlay selection without the need to verify the whole solution space. The algorithm finds the minimum possible distance in terms of the number of hops between two furthest selected resources, and at the same time fulfill different service type quality criteria.
- We extensively study the effectiveness of our approach in simulations using real-world node and usage traces from Guifi.net nodes. From the results obtained we are able to determine the key features of the network and node selection for different service types.
- Subsequently, we deploy our algorithm in a real production community network and quantify the performance and effects of our algorithm with a distributed storage service.

2.6.2 Placement algorithm

The Guifi.net community network consists of a set of *nodes* interconnected through mostly wireless equipment that users, companies, administrations must install and maintain in addition to its links, typically on building rooftops. The set of nodes and links are organized under a set of mutually exclusive and abstract structures called administrative *zones*, which represent the geographic areas where nodes are deployed.

In order to generalize the placement model for community services, we made the following assumptions that give to our model the flexibility to adapt to many different types of real services. In our case, a *service* is a set of S generic processes or replicas (with different roles or not) that interact or exchange information through the community network. The service can also be a composite service (e.g., three-tier service) built from simpler parts. These parts or components of a service would create an overlay and interact with each other to offer more complex services. Each of the service replicas or components will be deployed over a node in the network, where each node will host only one process no matter which service it belongs to.

We designed an algorithm that explores different placements searching for the local minimal service overlay diameter while, at the same time, fulfilling different service type quality parameters. The algorithm iterates using Breadth-First-Search algorithm (BFS) in the network graph, taking as root the given node and selecting the first $S-1$ closest resources to it. The node with high degree centrality is initially chosen as root. Degree centrality is the fraction of nodes that a particular node is connected to. In the case of several nodes at the same distance, nodes are selected randomly, distributing thus uniformly. The algorithm is described in detail in [14].

2.6.3 Network behaviour and algorithm performance

The service placement algorithm detailed in [14] is used to simulate the placement of different services in Guifi.net. Our goal is to determine the key features of the network and its nodes, in particular to understand the network metrics that could help us to design new heuristic frameworks for smart service placement in CNMCs.

From the data obtained, our first interest is to analyse the availability and latency of Guifi.net nodes. This can be used as an indirect metric of quality of connectivity that new members may expect from the network.

Figure 2.9 depicts the Empirical Cumulative Distribution Function (ECDF) of the node availability in terms of reachability, defined as the probability of contacting a node, measured as the average number of ping replies from a nearby monitor node, which is part of the distributed network monitoring infrastructure. Base-graph nodes have higher availability because they are closer to users, and is of high interest to users to take care of them. It can be observed that 40% of the base-graph nodes are reachable from the network 90% or less of the time. It is interesting to note that 20% of the core-graph nodes have availability between 98-100%, and those are most probably the nodes that comprise the backbone of the network and connect different administrative zones. Service placement is done on the core-graph nodes. For stable service provision nodes with higher availability need to be chosen. For details about the base and core graph and a detailed analysis of network structure and resilience see [15].

Figure 2.10 depicts the ECDF plot of the node latency, also measured as time to receive ping replies from a nearby monitor node. Similar to the availability case, the latency of base-graph nodes is slightly better. For both cases, 30% of the nodes have latency of 480 ms or less, which makes the other 70% of the nodes to have higher latency. The availability and latency graph demonstrate the importance of, and indeed the need for, a more effective, network-aware placement in CNMCs. By not taking the performance of the underlying network into account, applications can end up sending large amounts of data across slow wireless links while faster and more reliable links and nodes remain under-utilized.

Figure 2.10 depicts the ECDF plot of the node latency. The availability and latency graph demonstrate the importance of, and indeed the need for, a more effective, network-aware placement in CNMCs. Similar to the availability case, the latency of base-graph nodes is slightly better. For both cases, 30% of the nodes have latency of 480 ms or less, which makes the other 70% of the nodes to have higher latency. Not taking the performance of the underlying network into account, implies that applications can end up sending large amounts of data across slow wireless links while faster and more reliable links and nodes remain under-utilized.

In order to see the effects of the network-aware placement in the solutions obtained, we compare two versions of our algorithm. The first version i.e., Baseline, allocates services just with the goal of minimizing the service overlay diameter without considering node properties such as availability, latency or closeness. The second version of the algorithm called PASP, tries to minimize the service overlay diameter, *while* taking into account these node properties.

The availability and latency of the Baseline solutions are calculated considering the average number of nodes in optimal solutions (after the optimal solution is computed), which minimizes the service overlay diameter.

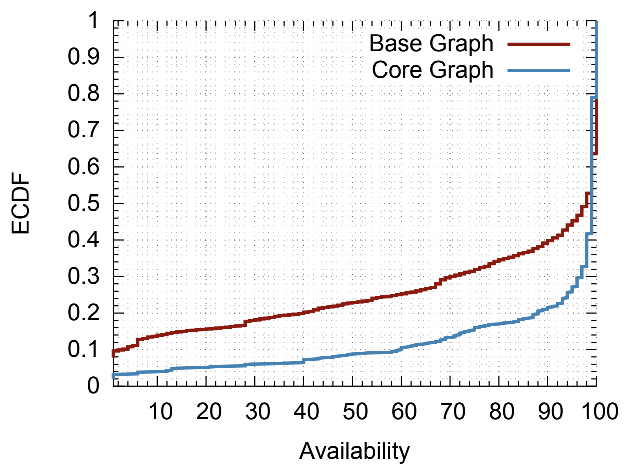


Figure 2.9: ECDF of node availability

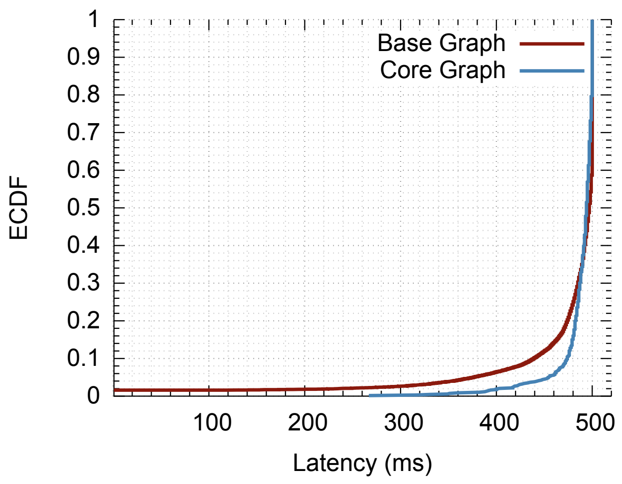


Figure 2.10: ECDF of node latency

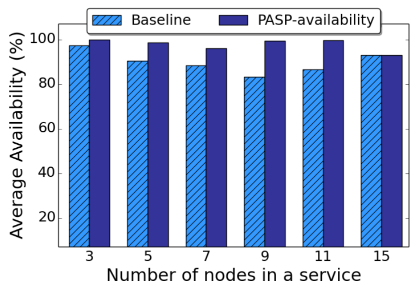


Figure 2.11: PASP-Availability

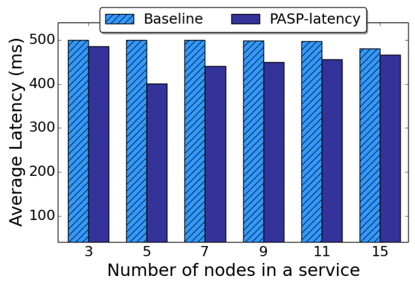


Figure 2.12: PASP-Latency

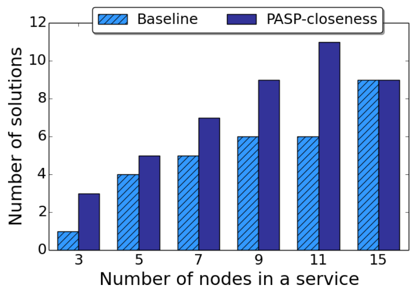


Figure 2.13: PASP-Closeness

We allocate services of size 3, 5, 7, 9, 11 and 15. Figure 2.11 and Figure 2.12 reveal that nodes obtained in the solutions with PASP have higher average availability and lower latency than with Baseline, with minimum service overlay diameter. On average, the gain of PASP over Baseline is 8% for the availability, and 45 ms for the latency (5-20% reduction).

We find that our PASP algorithm is good in finding placement solutions with higher availability and lower latency, however the service solutions obtained might or might not be very close (in terms of number of hops) to base-graph clients. Because of this we also developed another flavour of PASP algorithm called PASP-closeness. Figure 2.13 shows the number of solutions obtained that are 1-hop close to the base-graph clients. When PASP-closeness algorithm allocates three services, on average there are three solutions whose internal nodes (e.g, any of the nodes) are at 1-hop distance to any of the base-graph client nodes, contrary to the Baseline where on average there is one solution whose nodes are at 1-hop distance to base-graph clients.

Overall, in the two algorithms, there is a trade-off between latency and closeness. For bandwidth-intensive applications closeness seems to be more important when allocating services (e.g., PASP-closeness can be used), while for latency-sensitive applications it is the latency the one that naturally seems to be more important (e.g., PASP-latency can be used).

Moreover, from working with the Guifi.net data, we observed some patterns in the node features that conforms optimal allocations. We saw that the solution overlay diameter depends on the degree centrality of the nodes. Minimum degree centrality can be used to select the first node that composes the service (the solution). We saw that most solutions obtained are concentrated on a small set of average centrality values. Selecting the next nodes in a particular range of closeness centrality (for bandwidth-intensive services) and betweenness centrality (for latency-sensitive services) is specially useful to obtain more optimal overlays.

We find that our service placement algorithm (PASP) is good in finding placement solutions with higher availability and lower latency, however the service solutions obtained might or might not be very close (in terms of number of hops) to base-graph clients. Because of this we also developed another flavor of the PASP algorithm called PASP-closeness.

2.6.4 Deployment in a real production Community Network

In order to understand the gains of our network-aware service placement algorithm in a real production community network, we deployed our algorithm in real hardware connected to the nodes of the QMP network, which is a subset of Guifi.net located in the city of Barcelona. Figure 2.14 depicts the topology of the QMP network. Furthermore, a live QMP monitoring page updated hourly is available in the Internet[12].

We use 16 servers connected to wireless nodes in QMP. The nodes and the attached servers are geographically distributed in the city of Barcelona. The hardware of the servers consists of Jetway devices, which are equipped with an Intel Atom N2600 CPU, 4 GB of RAM and 120 GB SSD. They run an operating system based on OpenWRT, which allows running several slivers (VMs) on one node simultaneously implemented as Linux containers.

The slivers host the Cloudy[13] operating system. Cloudy contains some pre-integrated distributed applications, which the community network user can activate to enable services inside the network. Services include a streaming service, a storage service and a folder synchronizing service, among others. For our experiments, we use the storage service, which is based on Tahoe-LAFS[14]. Tahoe-LAFS is an open-source distributed storage system with enforced security and fault-tolerance features, such as data encryption at the client side, coded transmission and data dispersion among a set of storage nodes.

As the controller node we leverage the experimental infrastructure of Community-Lab[15]. Community-Lab provides a central coordination entity that has knowledge about the network topology in real time. Out of the 16 devices used, three of them are storage nodes and 13 of them are clients (chosen randomly) that read files. The clients are located in different geographic locations of the network. The controller is the one that allocates the distributed storage service in these three nodes and clients access this service. On the client side we measured the file reading times. We monitored the network for the entire month of January 2016. The average throughput distribution of all the links for one month period was 9.4 Mbit/s.

Figure 2.15 shows the average download time for various file sizes (2-64 MB) perceived at the 13 clients, after allocating services using Random algorithm (i.e., currently used at Guifi.net) and using our PASP algorithm. The experiment is composed of 20 runs, where each run has 10 repetitions, and averaged over all the successful runs. Standard deviation error bars are also shown.

Allocation of services using a Random algorithm by Controller is done without taking into account the performance of the underlying network. It can be seen for instance that when using our PASP algorithm for allocation, it takes around 17 seconds for the clients on average to read a 8 MB file. In the random case, the time is almost doubled, reaching 28 seconds for reading a file from the clients

[12]http://dsg.ac.upc.edu/qmpsu/index.php
[13]http://cloudy.community/
[14]https://tahoe-lafs.org/trac/tahoe-lafs
[15]https://community-lab.net/

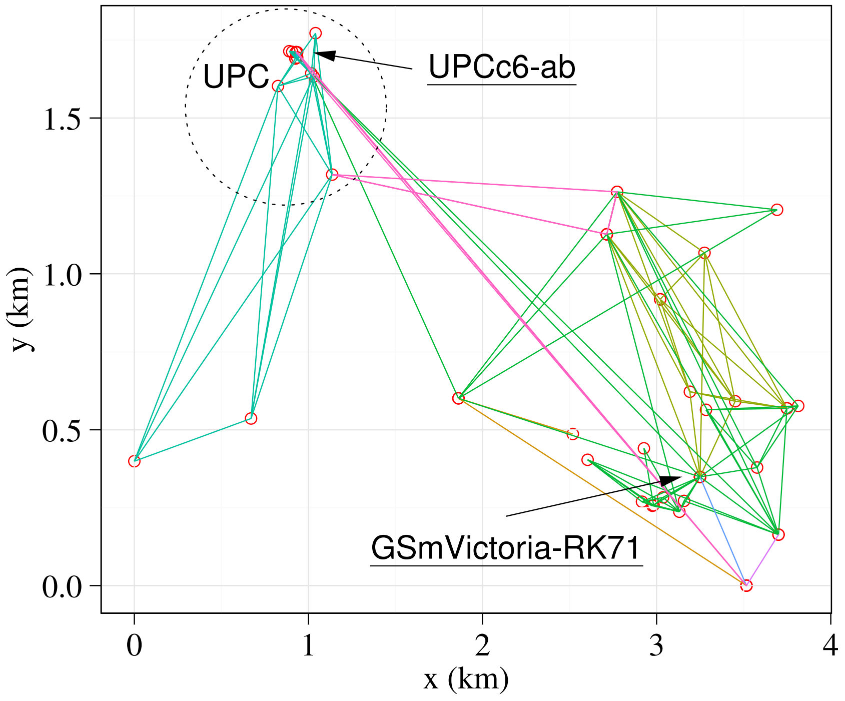


Figure 2.14: QMP topology

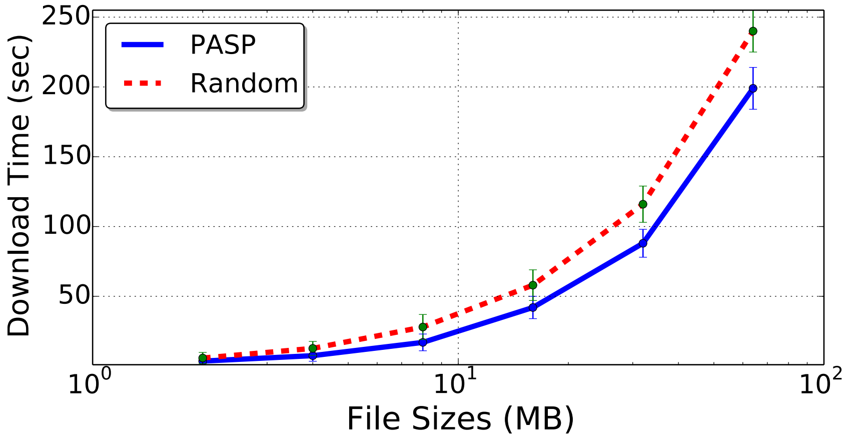


Figure 2.15: Average read time on clients' side

side. We observed therefore that when allocating services, taking into account the closeness and availability parameters in the allocation decision, on average (for all clients) our algorithm reduces the client reading times for 16%. Maximum improvement (around 31%) has been achieved when reading larger files (64 MB). When reading larger files client needs to contact many nodes in order to complete the reading of the file.

2.7 Takeaway

As Cloudy is now entering a mature phase, it is necessary to understand how to manage it so that the services installed in Cloudy are useful and adoptable for users. In other words, rather than developing new software, the development that is more needed in Cloudy is related to usability, service isolation, service placement.

In this first year of work, we addressed the need for network-aware service placement in community network micro-cloud infrastructures. We looked at a specific case of improving the deployment of service instance on micro-servers for enabling an improved distributed storage service in a community network.

As services become more network intensive, the bandwidth, latency etc., between the used nodes becomes the bottleneck for improving performance. In community networks, the limited capacity of nodes and links and an unpredictable network performance becomes a problem for service performance. Network awareness in placing services allows to chose more reliable and faster paths over poorer ones.

We introduced a service placement algorithm that provides improved overlay service selection for distributed services considering service quality parameters, without the need for exploring the whole solution space. For our simulations we employed a topological snapshot from Guifi.net to identify node traits in the optimal service placements. We deployed our service placement algorithm in a real network segment of Guifi.net, a production community network, and quantified the performance and effects of our algorithm. We conducted our study for the case of a distributed storage service. In experiments we showed that with our service placement algorithm, we were able to improve the total file reading time compared to the currently used random placement.

The capability to offer tailored cloud services at the network edge brings community network clouds a strong potential to complement the current generic cloud services of large data centers. The steps for integrating container-based service provision materialize this direction towards providing an operational open-edge cloud computing system, easy to use and for personal and public services.

Performance guarantees and solutions for the smart usage of scarce resources, as found in community networks, are needed to engage a larger number of stakeholders in community network clouds, but we also witnessed the need for better description of the services and their potential advantages (privacy, reversibility, control, . . .) to end-users, who often are not willing to exchange the commodity of very popular services for services that offer some technical advantage, but are less intuitive to use or are less adaptive as they leave users more freedom of choice.

It is clear, however –and this will be in the agenda of WP3 in the next year– that local services in CNs must address relevant use cases, i.e., services that are asked by the community, and must achieve a very high Quality of Experience for users to become interested and adopt them.

3 The PeerStreamer Platform

PeerStreamer is a P2P live video streaming application developed initially during the NAPA-WINE FP7 research project[1]. In the context of netCommons it is being developed further to become compatible with the technical features of a CN, and to improve its usability as a standard service rather than as an experimental and development platform. This Chapter introduces the concept at the base of P2P software and PeerStreamer, and it details the reason why it needs to be tailored to become integral part of a CN. It then introduces PeerViewer, the companion software of PeerStreamer realized in T3.3 in the first year of activity. The adaptation of the PeerStreamer software architecture for tailoring CNs and the development of PeerViewer represent the first efforts for extending the PeerStreamer framework beyond the original use case scenarios of local events live broadcasting and P2P-TV. In fact, PeerViewer, as explained in Section 3.2, leverages state-of-the-art web technologies that will make the PeerStreamer functionalities easily accessible also to non-technical people through any modern web browser. This is a fundamental feature for envisioning the extension of PeerStreamer to new usage scenarios like e-learning platforms and P2P group-video conferencing.

3.1 Peer-to-peer networks and PeerStreamer

A peer-to-peer network is an overlay that interconnects peers that share resources, tasks and workloads among each other without the use of a centralized administrative system. Peers make a portion of their resources, such as processing power, disk storage or network bandwidth, available to other network participants without the need for central coordination by servers or stable hosts. In addition, peers are both suppliers and consumers of resources; in fact, a peer-to-peer network is designed around the notion of equal peer nodes, simultaneously functioning as both “clients” and “servers” to the other nodes on the network. This model of network arrangement differs from the client–server model where communication is usually to and from one or more central servers.

Peer-to-peer networks generally implement a virtual overlay network on top of the physical network topology, where the nodes in the overlay are a subset of the nodes in the physical network. Data is still exchanged directly over the underlying IP network, but at the application layer peers are able to communicate with each other directly via the logical overlay links. Overlays are used for indexing, peer discovery, signaling, and sometimes, as in PeerStremer also for data exchange, and make the P2P system independent from the physical network topology. Based on how the nodes are linked to each other within the overlay network and how resources are indexed and located, it is possible to classify networks as unstructured or structured, or as a hybrid between the two. Unstructured peer-to-peer networks do not impose a particular structure on the overlay network by design, but are formed by nodes that randomly connect one to the other. Unstructured networks are easier to build than structured topologies and due to the same role of all peers, they are highly robust. PeerStreamer uses an unstructured topology, and implements a very robust topology manager that is resilient to churn, nodes, and link failures and, to some extent, to external attacks to the overlay.

[1]See http://peerstreamer.org for the original development and its further maintenance as an Open Source platform

3.1.1 PeerStreamer Architecture and Background

PeerStreamer is one of the few, if not the only one, Open Source P2P streaming applications and is being used by many research groups to experiment with innovative P2P streaming systems and solutions. Indeed, PeerStreamer is not a single, monolithic application, but a flexible composition of building blocks that enable the construction of different architectures for P2P streaming. Some basic features and architectural choices are fixed, such as using a mesh topology and distributing the video on an unstructured swarm. Others, as the characteristics of the topology, the neighbor selection, the chunks' and peers' scheduling strategies can be freely changed when building the actual application. The general philosophy and description of the NAPA-WINE and PeerStreamer architecture can be found in [1], here we recall only the basic principles of its architecture. Details and features on PeerStreamer can be found in [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26], while its initial use in community networks, done under the project OSPS[2] can be found in [25, 27, 28, 29]

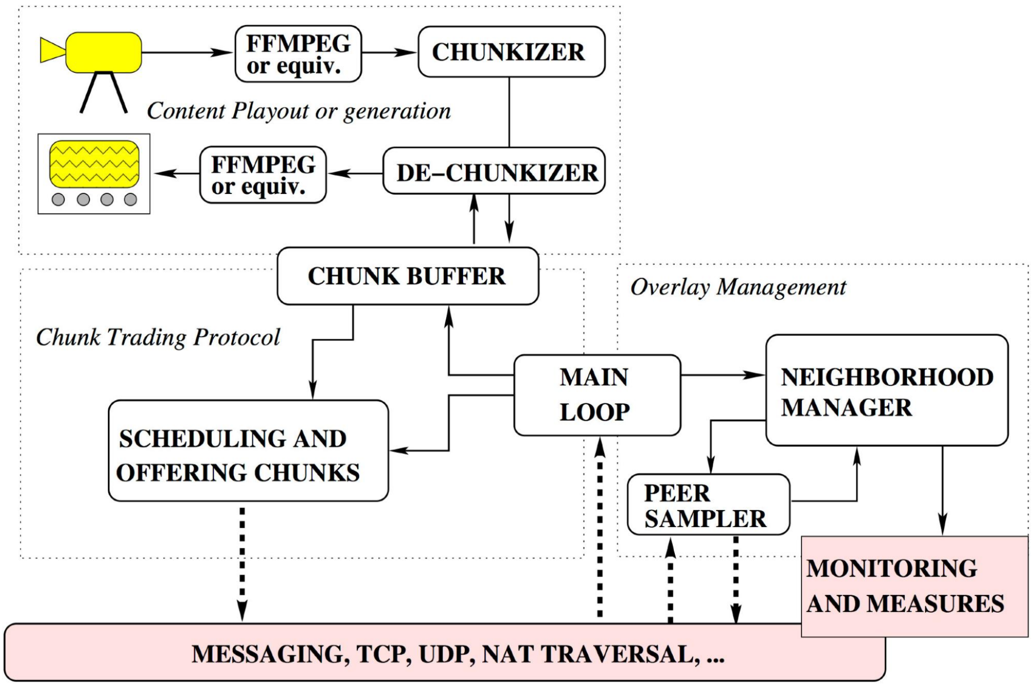


Figure 3.1: A scheme of the PeerStreamer architecture reproduced from [1].

Figure 3.1 reports the general architecture of PeerStreamer, which is based on chunk-based diffusion. The streaming application is split in three macro-blocks (the dotted rectangles): the I/O part, the chunk trading and the overlay management. Focusing on the chunks trading, the peers offer a selection of the chunks they own to some peers in their neighborhood. The receiving peer acknowledges the chunks it is interested in, thus avoiding multiple transmissions of the same chunk to the same peer. Different signaling protocols can be used, but the negotiation and chunk transmission phase is based on signaling exchanges with "Offer" and "Select" messages. For chunk scheduling, Offers are sent to neighbors in round-robin. They contain the buffer-map of the recent chunks the sender possesses at that time. Chunk selection is based on a "latest useful" strategy. The number of offers per second a peer sends is a parameter dynamically controlled to adapt to the network conditions and capacities. The source is a standard peer (no special software or hardware required). It simply injects some

[2]Open Source P2P Streaming http://osps.disi.unitn.it/doku.php an Experimental Project funded in the Open Call 1 of CONFINE https://wiki.confine-project.eu/experiments:overview

copies of new chunks. It implements a “chunkiser” to process the media stream (e.g., an encoded file, or a live stream coming from a web-cam, or Real-time Transport Protocol (RTP)/Real Time Control Protocol (RTCP) audio/video flows). In case of RTP/RTCP flows, the chunkiser is implemented as a lightweight C library and it is independent of the actual multimedia codecs used (e.g., H.264, AAC, VP8, Opus). Receivers dechunkise the stream and push the stream to the play-out system. The main loop at the center of Fig. 3.1 implements the application timing and orchestrates the execution of both semi-periodic tasks as sending offers, and asynchronous activities, as reception of chunks and signaling. The management and construction of the overlay topology in PeerStremer is fully distributed: each peer builds its own neighborhood following only local measures, rules and peer sampling. The outcome is represented by a directed graph (i.e., links are not symmetric) in which the peer at the edge head receives chunks from the peer at the edge tail, which is the one sending offers. Using directed links guarantees a greater flexibility in topology management than algorithms that impose the reciprocity between peers. In Fig. 3.1 the topology management is split into two separate functions. The “peer sampler” has the goal of providing good samples of all the peers in the swarm and the “neighborhood manager”, that selects the most appropriate peers for actual interaction. The complete streaming application includes also the “messaging” and “monitoring and measures” modules. The messaging module is a network abstraction layer that frees the application from all details of the networking environment, e.g., the presence of NAT, middle-boxes and other communication details. It offers a connection-oriented service on top of UDP, with a lightweight retransmission mechanism that allows the recovery of lost packets without high retransmission delay. The monitoring and measures module extracts network information by running passive and/or active measurements.

3.2 PeerViewer: a Web-enabled media player

In order to ease the adoption of PeerStreamer in CNs, we decided to split the functions of PeerStreamer in two, and design an architecture that separates the I/O function from the rest of the platform. This is a key feature to deploy PeerStreamer in real community networks, because the I/O module is the one that requires more computing resources. It does re-encoding and playback and thus it needs a reasonably powerful device to run. The chunk trading and overlay management modules instead have a very limited memory fingerprint and do not depend on heavyweight libraries, thus they can be embedded in low-power hardware. With this functional separation we will be able to install PeerStreamer directly in the network nodes of community networks, and leave a separate application for the playback of the video. As a consequence, the P2P overlay may be formed by hundreds or thousands of nodes in the same community network, even if only few of them will be watching the video. Having a high node density helps the streaming function in many ways, because it makes the overlay network more similar to the underlay network, with an advantage in the level of optimization it can reach. Furthermore, if PeerStreamer runs on the router nodes it can access information on the network topology that it could not otherwise access to and improve the performance even more [28, 30]. This functional separation is also necessary to go into the direction of integrating PeerStreamer in media center dedicated hardware, and to fully integrate it not only the Cloudy distribution, but also in other self-hosting service distribution like Yunohost or Freedombox[3].

PeerViewer is a web application that interfaces with PeerStreamer to provide media content to final user devices. It enables every device with a browser to receive and play real-time media content exchanged by PeerStreamer peers, irrespective of media composition.

[3]See respectively http://yunohost.org and https://freedomboxfoundation.org/

PeerViewer is able to reproduce audio and video streams and a mix of them, by taking advantage of HyperText Markup Language 5 (HTML5) features like the `<video>` and `<audio>` HTML tags. While PeerStreamer itself tries to be media-agnostic, by simply exchanging chunks, its companion PeerViewer inevitably need to be aware of the transported media. As noted before, a chunk is a generic "media container" that is able to encapsulate RTP[4] packets, among many different formats. PeerViewer implements a receiving pipeline that parses only chunks composed by one or more RTP packets. This behaviour is intentional and was chosen for simplicity and standardization of the RTP protocol, already implemented in many other media frameworks and applications.

As it is later described PeerViewer needs to unpack the RTP media packets and recombine them in an unique stream delivered to its Hypertext Transfer Protocol (HTTP) clients on the same Transmission Control Protocol (TCP) connection used for requesting the resource. This involves the synchronization of multiple media that RTP and PeerStreamer transport independently but also some machinery to reorder packets according to their headers.

During the realization of PeerViewer two architectures were tested, and one was finally adopted as detailed and explained in Section 3.3 and Section 3.4.

3.3 First approach: WebRTC technologies

The goal of the Web Real-Time Communication (WebRTC)[5] project is to enable browsers of Real-Time Communications capabilities. Applications developed for the browser, mobile platforms and Internet of Things devices can take advantage of the set of tehnologies and protocols defined by WebRTC to create rich and high-quality communication features.

3.3.1 WebRTC Concepts

WebRTC is a recent web technology that allows browser and mobile applications with functionality such as audio/video calling, chat, peer-to-peer file sharing, without any additional third-party software or plugins. Google published it as an open source technology back in May 2011 and it includes fundamental components for real-time communication on the web [31].

WebRTC is not a single Application Programming Interface (API) proposal for the browser, instead it is defined as a collection of APIs and protocols, developed and reviewed by the World Wide Web Consortium (W3C), and the Internet Engineering Task Force (IETF). Their goal is to join their contributions to support different browsers and operating systems. The development brought to life three major components of the WebRTC API are:

- `MediaStream`, allows a web browser to access the camera, microphone and screen.
- `RTCPeerConnection`, sets up calls (video or audio).
- `RTCDataChannel`, establishes peer-to-peer connections and send data through them.

The `MediaStream` API has a control over the synchronized streams of media. Each MediaStream has an input and output element. The input source may be a physical device, such as the camera or microphone or a virtual device like a window or the whole screen. The output can be forwarded to one or more destinations, such as video elements or PeerConnections.

[4]Real-time Transport Protocol, RFC 3550

[5]Web Real-Time Communication. https://webrtc.org/

The `RTCPeerConnection` API allows two peers to communicate directly, browser to browser. Its transport mechanism is responsible for Secure Real-time Transport Protocol (SRTP)[6], P2P signaling and Interactive Connectivity Establishment (ICE)[7].

The `RTCDataChannel` API represents a bi-directional data channel between two peers and enables the exchange of arbitrary data between them. Each RTCDataChannel has two properties that characterize its functioning: reliable or unreliable delivery of messages and in-order or out-of-order delivery of messages.

The Internet addressing system is still widely using IPv4 and because of that, most of the connected devices are behind one or more layers of Network Address Translation (NAT)[8]. WebRTC technology developers can take advantage of ICE which will abstract the complexity of the Internet addressing scheme. The API requires to know the ICE server Uniform Resource Locator (URL). By communicating with the ICE server, the browser will try to find the best path to connect with the remote party. ICE ises two types of servers, Session Traversal Utilities for NAT (STUN)[9] and Traversal Using Relays around NAT (TURN)[10].

3.3.2 Implementing a WebRTC frontend

WebRTC seemed a perfect fit as a frontend for the PeerStreamer project. With an always increasing support inside browsers, the possibility of transporting RTP packets from the P2P network to the user with almost zero modifications was very appealing. However, WebRTC isn't directly compatible with PeerStreamer in few points.

The first problem was with the Session Description Protocol (SDP)[11] descriptors. PeerStreamer, by the time of writing, doesn't transport any information about the streams it exchanges, it rather relies on the configuration of each node. PeerStreamer, in this sense, assumes that the configuration is distributed outside of the P2P network it creates. This creates a minor incompatibility with WebRTC, where each stream and its details must be declared inside the descriptor.

As stated before, WebRTC uses ICE to to find ways for two endpoints to talk to each other as directly as possible. ICE itself relies on STUN and TURN protocols when a direct connection is not possible. This implies that adopting WebRTC in PeerStreamer project also required adding and implementing ICE, TURN and STUN as core components.

Furthermore, the goal of PeerStreamer is to keep the binary size and number external dependencies as small as possible to enhance portability and to run on embedded devices, such as the cheap wireless equipment deployed in a typical Wireless Community Network (WCN). At this point of the evaluation, having WebRTC as a core front-end, involves developing or including an external SDP parser with non-standard extensions, the ICE, STUN and the TURN protocol. Unfortunately this list isn't finished yet.

WebRTC standards require mandatory encryption for every data or media streaming traveling between two endpoints. This means importing in PeerStreamer an entire cryptography suite such as

[6]Secure Real-time Transport Protocol, RFC 3711
[7]Interactive Connectivity Establishment, RFC 5245
[8]Network Address Translation, RFC 2663
[9]Session Traversal Utilities for NAT, RFC 5389
[10]Traversal Using Relays around NAT, RFC 5766
[11]Session Description Protocol, RFC 4566

OpenSSL[12] or GnuTLS[13]. It is also necessary to extend the SDP parser and serializer to load and send the appropriate encryption parameters to the clients requesting a stream and implementing one or more Datagram Transport Layer Security (DTLS)[14]/SRTP sessions for each client. PeerViewer doesn't support encryption between the server and the clients at current stage. Nevertheless, although we recognize the need to introduce security primitives in PeerStreamer, especially for usage scenarios like group-video conferencing, we believe that at this stage of development the main priority should be to provide a solid back-end with an easy to use front-end. Once these technical aspects are in a stable state, the framework will be extended for introducing security features not only for guaranteeing confidentiality and integrity of the data exchanged on the network, but also for managing Authentication, Authorization, and Accounting (AAA) of people involved in the CN. These security features will take into consideration the basic privacy requirements and will rely on privacy enhancing technologies (e.g., anonymous credentials).

Returning to the discussion about WebRTC, even by decoupling the core PeerStreamer from the WebRTC frontend doesn't solve the challenge of creating a lightweight front-end. While the front-end could run on another computer or platform with more power and more storage, the development is slowed down by the fast evolution of the WebRTC standard and the lack of an usable, external implementation of the standard.

3.4 An HTML5-based approach

PeerViewer was designed to take advantage of the media features available inside browsers, besides WebRTC. The HTML5 standard, introduced `video` and `audio` elements as a part of the newly available features of the Document Object Model (DOM) [32]. Browsers implementing the W3C recommendation are able to natively reproduce media files by reaching them at the Uniform Resource Identifier (URI) specified in the `src` attribute. While the recommendation doesn't specify which set of protocols must be supported, all browsers implement HTTP with optional TLS. Moreover, HTML5 doesn't specify which codecs are to be implemented, leaving browser vendors free choice of supporting their own preferred codecs. Fortunately, free and open source codecs are available and spreading across many commonly used browsers. In particular, PeerViewer takes advantage of VP8 and VP9 for video encoding and Opus or MP3 for the audio counterpart.

Browsers retrieving media content using HTTP are unaware of the origin of the content server by the web server. More generally they assume that the content flowing on the underlying TCP connection comes from a file-like interface. For this reason, PeerViewer emulates the equivalent of a file access over HTTP, with minor modifications, like omitting the `Content-Length` HTTP header.

3.4.1 Architecture design

The current implementation of PeerViewer is written in Go[15]. Go is an open source programming language that makes it easy to build simple prototyping software and provides a toolchain of components that allow to compose different libraries and external software. In this initial experimentation

[12]OpenSSL is an open source project that provides a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. https://www.openssl.org/

[13]GnuTLS is a secure communications library implementing the SSL, TLS and DTLS protocols and technologies around them. http://www.gnutls.org/

[14]Datagram Transport Layer Security, RFC 6347

[15]The Go Programming Language. https://golang.org/

phase the Go language provided the right trade-off between speed of execution and flexibility to test different designs and allowed us to experiment with various solutions. Nevertheless, the Go language is quite new and it needs its whole toolchain, plus, some language bindings towards C libraries for embedding into PeerViewer specific PeerStreamer and GStreamer features. For this reason, in the next versions of PeerStreamer we are going to evaluate other languages too and the choice of Go may be changed in the future.

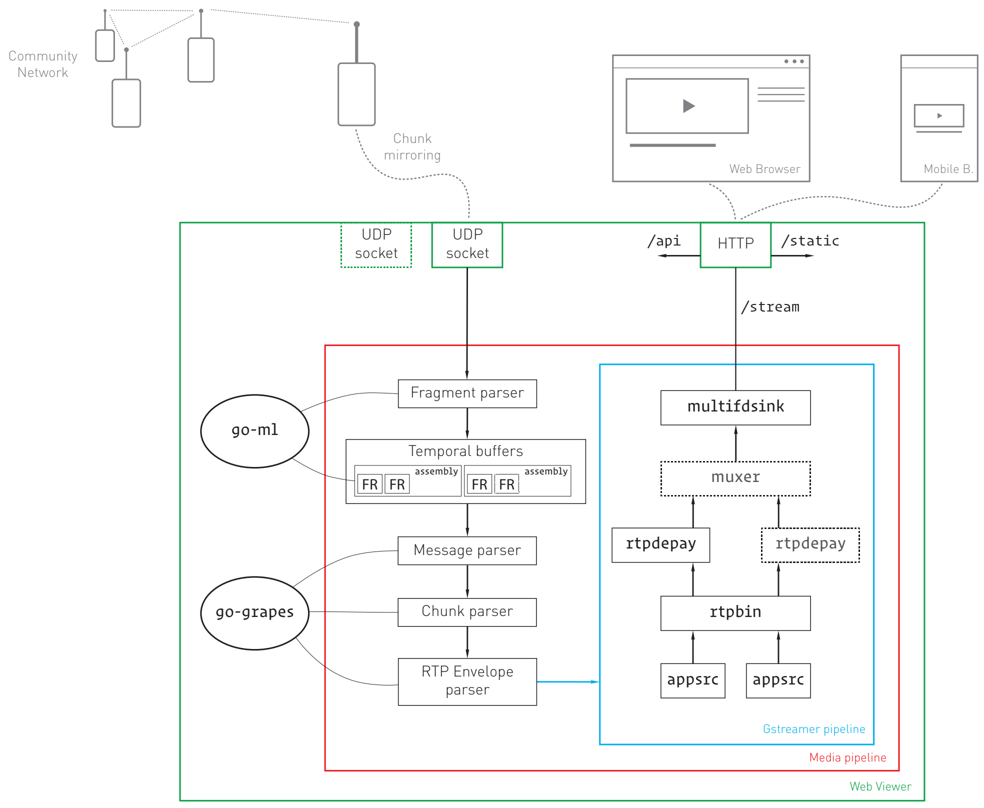


Figure 3.2: PeerViewer architecture diagram

The PeerViewer architecture was designed to allow maximum flexibility in its composition and configuration. Its development was subdivided in three different repositories, to allow maximum code reuse between current and future projects. PeerViewer contains a native Go implementation of the Generic Resource-Aware P2P Environment for Streaming (GRAPES) and Messaging Layer (ML) Libraries, originally written in C and embedded in the PeerStreamer project. In fact, PeerViewer assumes that its companion PeerStreamer instance is compiled and run with the GRAPES and ML libraries. As it is possible to see in the architecture diagram, PeerStreamer passes the datagrams received on the UDP sockets to the fragment parser. The fragment parser is part of the `go-ml` library. As all the other parsers visible in the picture, its function is to take the datagram header and reconstruct a structure that maps each field of the header to the corresponding value. Furthermore, the structure containing the header's fields, keeps a pointer to the packet content. Among other things, the fragment parser extracts the sequence number and the offset of the message from the received datagram. These information are used by the assemblies kept in memory by PeerViewer which function is to rebuild the complete messages from the content the fragments are carrying. These assemblies are kept memory for a limited amount of time in order to reduce the memory usage. In particular, without freeing the assemblies, a lot of incomplete buffers may accumulate caused by lost fragments. The temporal buffers facility is directly provided by PeerViewer. On the contrary, the fragment parser and the assembly are provided by the `go-ml` library.

The reconstructed message is then passed to the GRAPES message parser. A GRAPES message can itself contain one or more chunks, defined as the primitive that carries media content over the P2P network. Every chunk contained in the message is parsed and its content is unpacked and delivered to the appropriate message queue of Gstreamer. Every chunk can contain one or more RTP and RTCP packets belonging to different media streams. This is the case of streams composed by different media, usually a combination of audio and video. The GRAPES message, chunk and RTP envelope parsers are provided by the `go-grapes` library, that has the same goal of its C counterpart. The RTP envelope represents the smallest packet of the chain, and its header contains just the ID of the RTP session this packet belongs to.

For each PeerStreamer instance declared into the PeerViewer configuration file, an instance of the whole pipeline is initialized. This pipeline includes a part managed by Gstreamer. This part is unique to the stream kind. In fact, this is the only varying part based on the content handled by PeerViewer [16]. The rest of the application is content-agnostic. The only exception to the rule is the page containing either a `video` tag or an `audio` tag.

Parallel to the media handling pipelines, there's a full-featured web server serving different types of requests described in the next sections. The web server is able to serve multiple requests concurrently thanks to the goroutine mechanism. The concurrency model of the HTTP server is hidden in the standard library, for this reason, from the point of view of the Go developer the programming model looks completely synchronous. It is possible to handle I/O-heavy operations or complex database queries without taking too much care of not slowing down the rest of the application. However, a request may remain blocked because of long operations run inside its own goroutine. PeerViewer needs only one web server, shared by all the streams declared in the configuration file. Moreover, given PeerViewer's common use case, TLS is not implemented on the HTTP socket.

3.4.2 Front end user interface

The user interface has been created using web technologies, leveraging the power of different JavaScript-based libraries and frameworks. The User Interface (UI) is a sub-project of PeerViewer, located inside the `public` folder of the repository. The front end is not just a collection of HTML, JavaScript and Cascading Style Sheets (CSS) files, on the contrary the application must be first compiled and packed. In fact, the UI is completely dynamic, rendered by JavaScript. The `index.html` file is static and it is needed just to bootstrap the whole application.

```
<!doctype html>
<html>
<head> <title>PeerViewer</title> </head>
<body>
  <div id="root"></div> <script src="app.js"/>
</body>
</html>
```

[16]At this stage of development PeerViewer supports two types of GStreamer pipelines: one for handling VP8 video and one for handling Opus audio.

The `app.js` file is a bundle containing all the required JavaScript modules that compose the interface. Node.js modules are JavaScript files that export one or more symbols by following a naming convention. More precisely, every JavaScript file that assigns some value to the global variable called `module.exports` is a module that can be `required` from other files.

```
 // in file a.js
function mySayHello() { return "Hello, World!"; }
module.exports = { sayHello: mySayHello };

// if file b.js
var greetings = require('./a.js');
console.log(greetings.sayHello());
```

The bundle `app.js` is a set of all the modules used inside the application plus the equivalent of a `main()` function, the so called "entrypoint". No other modules should try to `require()` the entrypoint. On the contrary, the entrypoint is the module that bootstraps the application by importing and initializing other modules. While modules are the primitive for separating code and create semantic interfaces between parts of an application, it is often useful to group modules (consequently files) in a tree like structure. Node.js supports this pattern by means of packages, defined as collections of modules organized in a file system sub-tree. The Node.js Package Manager offers a convenient way to include third-party packages in your application. On the other side, Node Package Manager (NPM) gives you the possibility of contributing new packages that can be imported by others.

Creating a valid NPM package is simple: it is necessary that the root directory of the package contains an appropriate `package.json` file declaring some attributes regarding the package. These includes: the package name, version, author, source code repository URL, license, etc. The file also list package's dependencies, organized in categories (generic, development only, test only). The file is managed by the tools provided by NPM itself and doesn't need to be hand-written. The application is a package itself and has all the advantages of any other package, for example importing other packages. This is a common pattern in modern JavaScript development, where all the information about any package is contained within the package as code. When installing a package, NPM takes care of recursively resolving all the dependencies, without human intervention. Another advantage of having metadata as code, is that it can be treated by tools built for managing software, from `diff` to version control systems like Git, building a better collaboration between developers.

In order to create a bundle containing all the dependencies, Webpack, a Node.js module bundler, is employed. Webpack statically analyzes the modules in order to reconstruct the dependency tree. The exploration begins from the file defined as the entrypoint. For every require statement found in the source code, it tries to resolve the dependency by looking in the standard import paths. It is possible to reference other modules with relative or absolute paths. In case a relative path is given, Webpack explores the local file system starting from the directory the requiring file is placed in. For absolute paths, Webpack first try to search for the module among the locally installed packages. If it is impossible to find a match, then Webpack tries to look into the global (system-wide) packages. Once a module corresponding to the import path is found, WebPack copies its content in the bundle and change the requiring code to reference the copy of the module just created. To reduce the final file size, some plugins are in charge of removing dead code, useless white spaces, mangling variable names and eventually remove duplicate required modules inserted twice into the bundle.

When building a single-page app like this, it is probable that not all the modules are required on the first loaded page. Taking the example of PeerViewer, the code needed to render the player page on the about page isn't immediately required when the user hits the homepage. Webpack supports splitting the bundle into multiple chunks that are automatically loaded when a specific page need to be loaded. Using this method it is possible to drastically speed up the loading of the application. The most important factor that slows down the loading of the application is certainly the limited availability of bandwidth. This is not the case of PeerViewer, as it is used on a local area network, wired or wireless. The JavaScript engine inside browsers can take seconds to interpret the whole bundle. Especially on mobile browsers, where CPU and memory are limited and shared between a lot of different running activities and services, a JavaScript bundle of just few megabytes can take beyond ten seconds.

By subdividing the bundle in chunks, even mobile browsers are able to load the application in an acceptable time. The downside of this is that when the bundle is divided by doing cuts on the different pages and their required modules, page transitions are slow. However, the chunk loading influences just the first time a page is requested, because then the browser can use the already interpreted module.

The PeerViewer bundle size is about 500 kB and takes advantage of the splitting mechanism. The bundle includes many libraries used to build the UI, a brief description of the most relevant ones follows:

React: an open-source JavaScript library providing a view for data rendered as HTML[17]. React views are typically organized in components, the basic unit for creating layouts. Components can be nested to create higher-level ones, that provide enhanced functionality. Components can then represent the real building blocks of the application and not the blocks imposed by the HTML language.

Redux: a predictable state container for JavaScript apps[18]. Complex JavaScript applications must manage complex state data. The so called "state" can include server responses and cached data, as well as locally created data that has not yet been persisted to the server. Redux stores a state of the application and makes it possible to access it from several apps in a coordinated way.

React router is a complete routing library for React, it keeps the UI in sync with the URL[19]. It can be further connected to Redux to make the current URL part of the application state. With this combination, requesting a URL change translates into dispatching an action that will create another version of the state with the new URL, which in turns allows complex workflows.

Material UI: is a set of React components that implement Google's Material Design guidelines[20]. Instead of creating a brand-new style or using Bootstrap or similar frameworks, these components wrap the underlying HTML elements into styled and ready to use higher-level responsive components.

Babel transpiler: PeerViewer uses the modern JavaScript syntax, formalized as ECMAScript 6 (ES6), released in 2015. Just few browsers natively implement the specification, hence the Babel "transpiler" is used during the bundle creation to transpile ES6 files into compatible ES5 files[21]. PeerViewer relies on the JavaScript eXtended (JSX) syntax for creating React Components using an HTML-like syntax. Facebook, creator of React, provides a transpiler for

[17]JavaScript library for building user interfaces. https://facebook.github.io/react/
[18]Predictable state container for JavaScript apps.
http://redux.js.org/
[19]Complete routing library for React.
https://github.com/reactjs/react-router:
[20]Set of React components that implement Google's Material Design.
http://www.material-ui.com/
[21]JavaScript compiler.
https://babeljs.io/

ECMAScript 5 source files, while there exists a contributed Babel plugin called “react” that does the JSX transformation before transpiling to ES5.

With all these implemented technologies it is possible to write modern and clean JavaScript applications. The challenge becomes writing reusable React components, and compose them accordingly to the needs of the application. This justifies the flat structure of the frontend project, organized in components, containers and views; the components directory contains React components that show or manipulate data by dispatching actions; the containers directory uses the Redux library to connect components to the application state; the views directory take components and containers and assemble them in pages.

3.5 Source Code

As stated above, the PeerStreamer framework is a flexible composition of different modules that enable the construction of different architectures for P2P streaming. All the code required for building different PeerStreamer’s instances is hosted on GitHub under the netCommonsEU organization[22]. There are currently seven repositories use by the PeerStreamer framework and each of them is briefly described below:

- `PeerStreamer-build`: this is the PeerStreamer build system and is used for automatizing the build process, the installation and the execution of simple test scenarios involving both PeerStreamer and PeerViewer. This is the perfect starting point for whoever is approaching the PeerStreamer framework for the first time. The code of this repository can be checked out at the following link:

  `https://github.com/netCommonsEU/PeerStreamer-build.git`

  The detailed instructions for building and installing PeerStreamer and PeerViewer can be found here: `https://github.com/netCommonsEU/PeerStreamer-build/blob/master/README.md`

  and the procedure to follow for testing the two applications in basic scenarios are reported at: `https://github.com/netCommonsEU/PeerStreamer/blob/master/testing/Testing.md`

  The basic tests involve the use of a single host where all communications are performed on the loopback interface. Thanks to the tests made available by the PeerStreamer build system it is possible to experience live video play-out using both PeerViewer or other tools like VLC of ffplay.
- `PeerStreamer`: this is the main repository of the PeerStreamer application and enables the construction of different architectures for P2P streaming. The code of this repository can be checked out at:

  `https://github.com/netCommonsEU/PeerStreamer.git`

  The code of this repository depends both on `PeerStreamer-grapes` and `PeerStreamer-napa-baselibs`
- `PeerStreamer-grapes`: this repository contains the code of the GRAPES (Generic Resource-Aware P2P Environment for Streaming) library. The library implements different modules required by the PeerStreamer application like the chunk buffer, the chunk trading and

[22]GitHub netCommonsEU organization: `https://github.com/netCommonsEU`

the chunkiser/de-chunkiser (see section 3.1.1). The code of this repository can be checked out at:

https://github.com/netCommonsEU/PeerStreamer-grapes.git

- `PeerStreamer-napa-baselibs`: this repository contains the NAPA libraries originally developed in the context of the NAPA-WINE FP7 research project. These libraries provide different features required by the PeerStreamer application like the implementation of the messaging layer (see section 3.1.1). The code of this repository can be checked out at:

  https://github.com/netCommonsEU/PeerStreamer-napa-baselibs.git
- `PeerStreamer-peerviewer`: this is the main repository of the PeerViewer application (see 3.2). The code of this repository can be checked out at:

  https://github.com/netCommonsEU/PeerStreamer-peerviewer.git

  The code of this repository depends both on `PeerStreamer-go-grapes` and `PeerStreamer-go-ml`
- `PeerStreamer-go-grapes`: this is a partial implementation of the GRAPES library in pure go and is currently use by the PeerViewer application. The code of this repository can be checked out at:

  https://github.com/netCommonsEU/PeerStreamer-go-grapes.git
- `PeerStreamer-go-ml`: this is a partial implementation in pure go of the messaging layer originally provided by the NAPA libraries. This library is currently used by the PeerViewer application. The code of this repository can be checked out at:

  https://github.com/netCommonsEU/PeerStreamer-go-ml.git

4 The CommonTasker Mobile App

CommonTasker is a prototype mobile application, whose development is the final goal of Task 3.4 of netCommons. Contrary to Cloudy and PeerStreamer, it is being built from scratch and does not rely on previous work, furthermore it is developed in close cooperation with Task 3.1 on participatory and multidisciplinary design and development.

4.1 Mobile crowdsourcing and sharing economy for CNs

Mobile crowdsourcing: The basic idea of the application is to provide a platform for *mobile crowdsourcing* and sharing economy practices in community networks. *Crowdsourcing* presents an alternative paradigm for coping with tasks and generating new services. It achieves this relying on small contributions (effort, data) from many different individuals. These contributions from the crowd of users are then combined together and processed to generate collective intelligence, i.e., knowledge about a quantity or a fact. The Wikipedia is one example of crowdsourced project in that the content of pages is composed by many different individuals writing about the things they know best.

Contributions to crowdsourcing applications, like Wikipedia, can be carried out while sitting at home or in an office, in front of a PC. Smart mobile devices have enabled the expansion of the concept in mobile environments. Users with smart devices, equipped with different types of sensors, can sense data as they move around and upload them to a platform. What is called *mobile crowdsourcing* (or, almost interchangeably, mobile crowdsensing and participatory sensing) can yield several types of applications addressing, among others, environmental conditions (area maps of noise, temperature, humidity), transportation (real-time information about traffic, parking availability and other events), and social journalism (e.g., reporting problems in the neighborhood to the municipality).

Sharing economy: On a different yet related note, *sharing economy* applications enable people to share different types of resources to maximize their utilization value and save money. Car sharing and parking space sharing are two such examples from the area of transportation. Applications, either mobile or not, coordinate the allocation of people to cars or the exchange of parking spots, with significant benefits for the involved individuals (cost savings) but also the broader social welfare (save CO_2 emissions). The sharing becomes all the more meaningful when it concerns (expensive) tools that are used by their owner only for a small fraction of time; for instance, an electric drill for home use or a tractor for farming work.

Another type of sharing economy applications relates to sharing manual tasks and work. Therefore, Mary could ask help for a demanding task (e.g. assembling a wardrobe) and get help from John in exchange of credits and under the expectation that later in time Mary could help John with something else (e.g. prepare a special Christmas meal).

The fundamental remark is that all these applications have in common a strong dimension of cooperation and sharing that renders them particularly attractive for a CN. After all, CNs are themselves crowdsourced networks in the sense that users contribute resources (nodes, radio equipment, or digging expenses) to make up the network. Moreover, community networks have utilized over the years different types of crowdsourced tools such as crowdfunding tools.

In light of this discussion, the original idea for CommonTasker is to develop an application supporting the following capabilities:

- **Information collection and sharing:** Relevant information varies from CN to CN. It may relate to environmental phenomena (e.g., humidity, noise), events taking place, problems in the road network or elsewhere in the city/village/neighborhood. Monitoring (home, farms) also requires such functionality.
- **Crowdsourcing tasks/jobs:** The application should provide community members with the capability to post a task they need help with (e.g., an agricultural task such as crop harvesting); and make themselves available for (specific) ongoing tasks.
- **Equipment/tool exchange:** Exchanging tools, machinery, storage devices for temporary use can result in non-negligible savings in a household budget, whether in a city or the countryside. Community members can collect credit by contributing to tasks and consume it by recruiting others to help them with their own. More importantly, the credit could be used for making up for the expenses of network equipment. Namely, a member could offer labor and time to pay its part of the annual subscription fee to the network.
- **Answering questions:** This a forum-like functionality, which lets individuals enter questions they have and get replies from other community members. These questions are of local interests, e.g., questions about technical issues of the CN, recommendations about places to visit, tips about farming practices).

Summarizing, CommonTasker is envisioned as a mobile app that can be deployed locally over CNs, independently of the public Internet. Functionally, it inherits features from existing commercial web and mobile apps such as Quora, Yahoo! Answers, and Task Rabbit. Its aim is to adapt these capabilities to the CN context in ways that a) they can facilitate everyday tasks within the communities, thus directly generating added value for the CN; b) they catalyze cooperation and community ideals, thus indirectly contributing to the sustainability of the CNs.

4.2 Architecture

The CommonTasker mobile application includes two parts, as shown in Fig. 4.1:

- **Client app:** The client side refers to the part of the application that is presented to the end user and runs on mobile devices. It is the actual interface between the user and the back end. Users interact with the client part via their mobile devices and produce data such as when they register with the application or create new entries. The design of the client side focuses on the the user interface requirements: it needs to be user-friendly, attractive and practical.
- **Backend server:** The backend server is in charge for the storage of data and coordination of the whole application. Data such as user, task, categories, time and location information are stored in the databases. It has to cross-check the users' login information for registration and authentication to take place. The back end communicates with the client side and handles, i.e., updates or processes, the data it receives in the corresponding databases.

More detail about the two components of the application is provided in what follows:

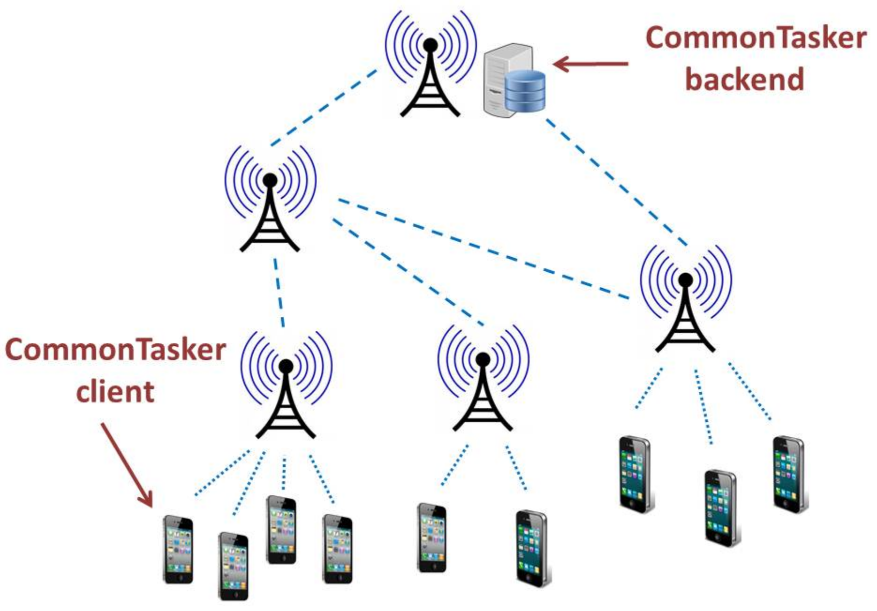


Figure 4.1: Client and back end side of the application in a wireless CN.

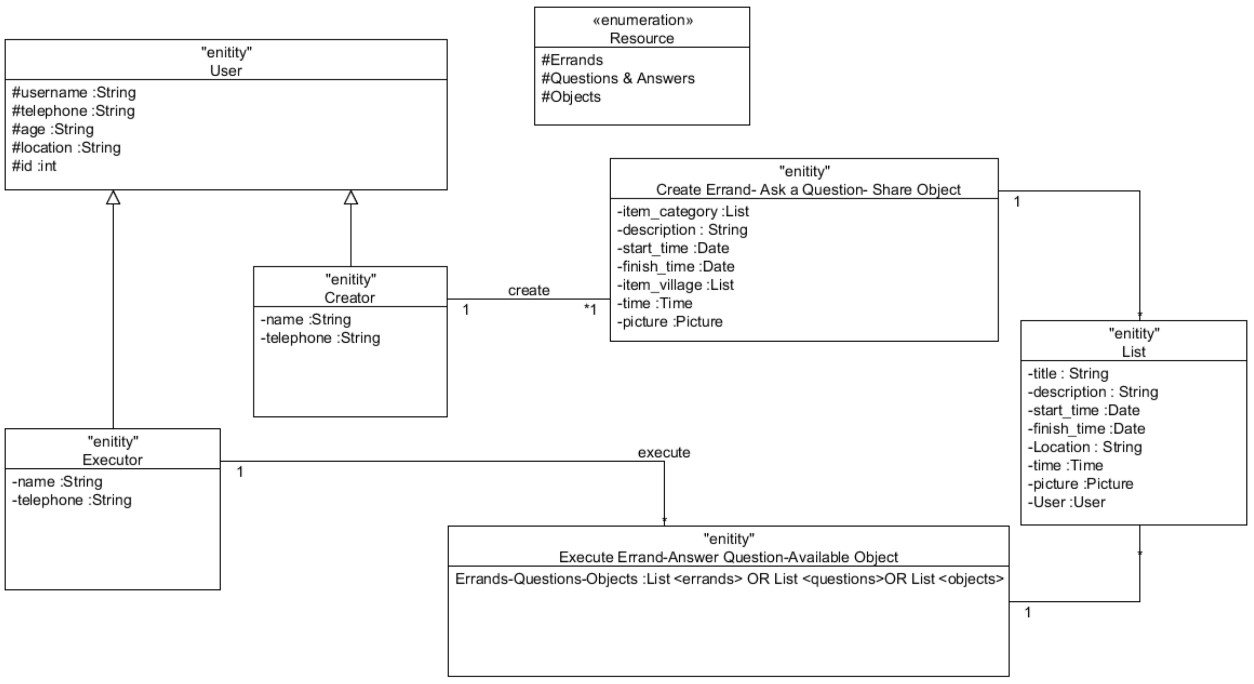


Figure 4.2: CommonTasker architecture.

4.2.1 The client app

In order to implement the application as described in the previous section, CommonTasker consists of different entities. The class diagram in Figure 4.2 denotes the connections between these entities.

- **User:** The entity User may either be a creator of a task or an executor. Certain information is retained for each user such as their (nick)name, a telephone, age (including checking users for being over 18), their location (useful for relating them to spatially distributed tasks) and the ID used by the application as identifier of the particular user.

- **Creator:** Creators are users with permissions to create entries, i.e. they can create and post new tasks, create and post a question or create and post an object available for sharing. Each of these created entries are grouped into specific categories depending on their context, i.e. indoor or outdoor tasks, questions related to food/politics/health, objects referring to fertilizers/machinery etc. Users can create new categories or map their entries to the default pre-defined ones. New entries contain a description text field, a start and ending time, location, time of the creation and a corresponding picture. The relevance of these fields depends on the type of the entry, e.g., the time fields may not be necessary when posting a question. Tasks are created by filling out forms.
- **Executor:** Executors are users with permission to execute entries that are created by creators. Executors select a task from the available listed items and the different categories available and explicitly accept to contribute to it; likewise, they may answer a question or select available objects shared by the creators.

From the methodology point of view, the development of the UI followed the LUCID model, which identifies several distinct phases in the development process:

- Concept definition;
- Analysis of needs and requirements;
- Design with standard basic screens;
- Revision of the original design and improvements; and
- Final system development

Our main concern has been to create a functional and extremely simplified user interface, which helps individuals-residents to interact with the device. To this end, it is important to distinguish the following categories of users when designing the app:

- Users with clumsy behavior on mobile devices and UIs;
- Users who would like more functionality from the application such as, for instance, measuring task completion times or displaying auxiliary video for a difficult job or annotating each task description with geographical information;
- Older users who find it difficult to type and navigate in mobile menus when key functionalities are not properly adjusted.

The app is customized separately for each of these categories.

The assumption is that the application will run on smartphones (or pads); hence, the user will rely primarily on the touch screen to navigate through the application menus, using where necessary the buttons available in the device, particularly the recoil button.

4.2.2 The backend server

The requirements/functionalities of the backend serve include the following:

- Posted entities on the client side such as image, location, date, time, are uploaded to the back-end server.Entries have to be made for the uploaded files to the associative real-time database, which stores all information about the posted components;
- The real-time database supports functions that manage authentication credentials;

- User profiles are maintained for security purposes but also for keeping track of the different data types and capabilities associated with them;
- The communication with the app is fast; response data shall be delivered no later than some seconds and preferably within 200 ms after the arrival on the database. This ensures that all necessary parameters have been propagated on the extendable lists the app for its three main types of entries (tasks, objects, questions).

4.3 CommonTasker for Sarantaporo.gr

The CommonTasker app has been chosen in netCommons as the app that will test the participatory design practices of T3.1. Therefore, a CN had to be chosen to carry out the process in cooperation with the Nethood team leading T3.1 and adapt the application for its needs. The chosen CN is the Sarantaporo.gr due to the close connections between the application developers (AUEB-RC team) and the group of volunteers that have initiated and up-to-date run and manage the CN[1].

Hence, the creation of the prototype application is a step-by-step procedure involving the following steps (for more details of the participatory design process, refer to netCommons D3.1 [33]:

1. informing the Sarantaporo.gr core members about the intended application and receive feedback about the context of the application;
2. adapting the functionality of the application according to this feedback;
3. presenting the resulting application skeleton to the local community;
4. incorporating their feedback and adapt the app features.

4.3.1 Feedback from the Sarantaporo.gr team - steps 1 and 2

Therefore, during the last six months, we had a series of face-to-face meetings with George Klissiaris, Vassilis Chryssos, and Rossie Simeonova, to iterate on the specific community needs and the features the application should have. The main information that came out in this process was as follows:

- **Target groups and requirements:** The intended group of CommonTasker users are people living in the coverage area of Sarantaporo.gr. These are basically the residents of all the 14 villages it expands to plus the visitors of the area. It is a community that has only recently been introduced to the Internet connectivity and ICT capabilities.

  The local population consists of primarily farmers and breeders that are particularly interested in getting informed about matters regarding their profession; young people familiar with technology that wish to enhance their education and communicate with their friends; elderly in need for assistance provision; local businesses that want to expand their contacts with their customers; visitors and tourists searching for information. A more detailed analysis of the area population and its original incentives to use the CN is given in netCommons D2.3 [34].
- **Application features:** The original idea from AUEB to the Sarantaporo.gr core members, was a crowdsourcing task application where tasks are either actual jobs or questions posted by users. These jobs could be announced by people and executed by those interested in participating in such a task. The questions will be posted by and answered by users of the CN. Users can use this feature for different aspects of the everyday life: to report malfunctioned roads, extreme weather conditions, get informed about matters of the local community and good farming practices, or

[1] In fact, two of the members of the Sarantaporo.gr NPO are working for netCommons under contract with AUEB-RC

even for entertainment reasons. The discussion with George Klissiaris and Vassilis Chryssos led to the addition of the sharing economy feature, i.e., the capability to share agricultural tools, equipment or fertilizers with those in need of them. Such a feature would be very handy to new or small farmers that cannot acquire expensive machinery and they can borrow for a specific time interval from those that have it but not use it.

The new feature was incorporated in the prototype of CommonTasker and the different types of functionalities were named “Errands”, “Questions” and “Sharing”. This CommonTasker skeleton, with the respective User Interface (UI) was presented to a selected representative group of locals on Nov 26th 2016, during a workshop organized in the Sarantaporo valley.

4.3.2 Feedback from the local community and future plans - step 3

The application was presented with emphasis on the capabilities, i.e., what could be used for.

The feedback received from the local community indicated that the their aspirations of locals are not fully in line with those of the design team (AUEB-RC, Nethood and Sarantaporo.gr). In more detail:

- The sharing economy feature has not been very convincing for the local farmers. They explained that some sharing is practised regarding tools, but among people that know well and trust with each other. Further interactions with the local community will show whether this feature is worth incorporating in the application to expand sharing practices beyond the small circles of trust.
- The capability to carry out micro-tasks for others (aka errands) did not receive much attention. Again, people rely on the word-of-mouth to get other people for performing errands for them within the village. They anticipated, however, that this cannot not always be possible across the 14 villages area.
- On the other hand, the core functionality of the application, the crowdsourcing of “information” did prove to be a useful core functionality for a case that was not within the original AUEB and Nethood priorities (crowdsourced smart farming). According to this, the application could be used to collect various types of data from farms and transfer them over the CN. Interestingly, this case implies the possibility to collaborate with a commercial IT company in this field with a lot of “credibility” in the eyes of the community.

The development of the application will proceed focusing and enhancing the information exchange feature. The goal of the modifications will be to consider the farming necessities of the locals. Providing video and audio tools to share local farm information will also be considered, i.e. a local audio broadcast channel in line with suggestions made by the locals.

4.4 First release of the CommonTasker software

The first version of CommonTasker has been designed in Android Studio 2.1.2 and is compatible with Android versions from 2 to 6. The focus at this initial stage has been to select the basic components of the application and design a friendly and practical UI according to the particularities of Sarantaporo.gr users.

The server side of the application has received much less attention so far. The respective functionality in the prototype is implemented using the Firebase platform, a tool available by Google for handling databases.

4.4.1 Client capabilities

The client app provides fundamental capabilities within each of the "errands", "questions" and "sharing" components (see Section 4.3.1).

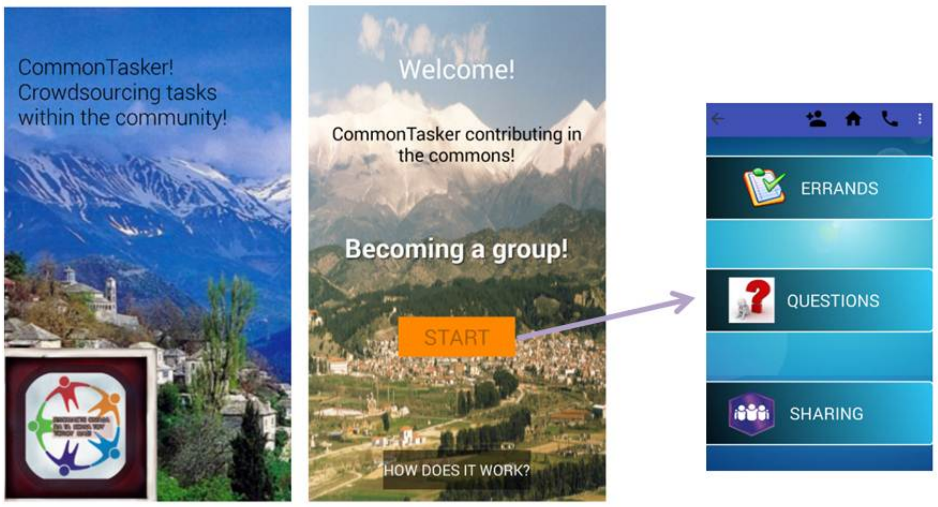


Figure 4.3: User-app interface: introducing and welcoming users to CommonTasker.

Errands: The idea is to find people able to assist or attend errands on behalf of the requesters. Interested users of the application are able to post an errand and request for people to contribute to it. The notion of errands here is mainly focused, but not limited to, agricultural jobs or tasks that can be attended by available individuals; for instance, assist in the harvesting of almond trees or cultivating or watering fields. Errands can be spatially distributed and usually demand from the user to travel to the errand's location.

User capabilities: Once users have installed CommonTasker in their mobile device, they are able to select "Errands" from the main menu of the application. Each user is provided with the opportunity to either create a new errand and post it for interested users to see or to select an errand from a list and execute it (Fig. 4.4). Errands are grouped into different categories. Users are able to select tasks from the corresponding categories, e.g., outdoor tasks, indoor tasks, to assist with or create their own categories as they see fit.

Questions: This feature aims to enhance the way that people communicate and exchange information and ideas within the community network. A user can select the "Questions" button in the main menu of the application. Examples of shared information include reporting road problems due to bad weather conditions, exchanging knowledge about the cultivation of herds and plants, receiving consultancy for sick animal treatments. The people that respond to the questions can be experts or non-experts in a specific field of questions.

Users select "Questions" from the main menu of the application and they can create a question and

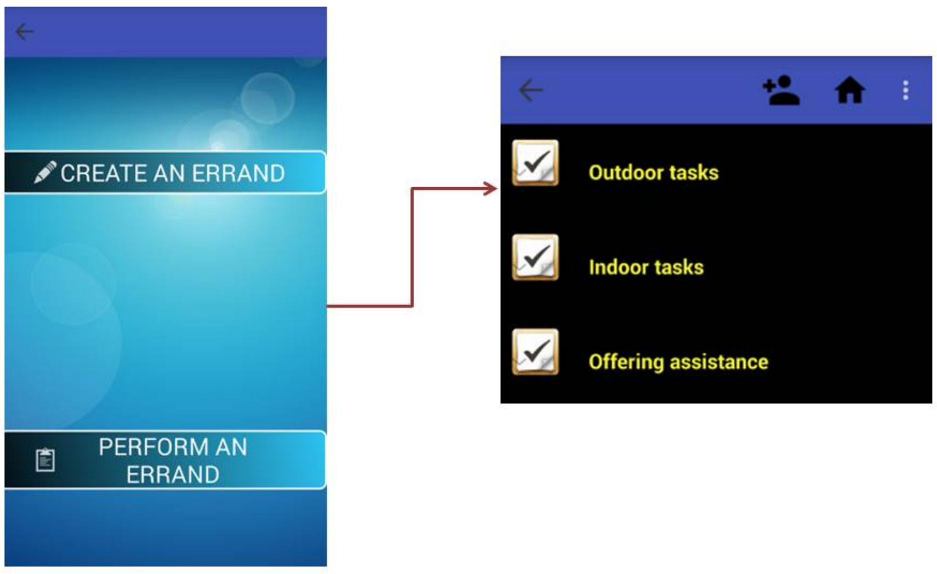


Figure 4.4: Create and perform errands.

post it (Fig. 4.5). The time of post and a picture can accompany the created entry. Interested users can answer or see all the available questions posted and decide to answer a question. Grouping into categories is also available in this component either using the default ones (food, politics, health, social, etc) or creating new categories. Once a question or an answer has been posted, interested users will be able to follow the discussions from their mobile devices.

Sharing: This component aims to implement a basic sharing economy idea. In this case, the objective is for the application to become a communication and coordination tool that offers a way for contacting and organizing the exchange of objects or equipment among users of the application.

User capabilities: Users can access this component through selecting ”Sharing” in the main menu. They able to create a new entry and provide a description of the corresponding object they want to share. Additional information will be asked from them such as the time interval that the equipment is available, the location of the object and a picture. Interested uses can check the list of available objects to be shared and select the one that best suits their needs (Figure 4.6). Object are grouped by default in categories of tools, machinery and fertilizers but the option to create a new category is also available when creating a new entry. Once users select the object they are interested in, a notification will be sent to the tool owner.

4.4.2 Firebase

Firebase[2] is a back end platform that provides several features for building mobile and web applications. It provides a wide variety of services such as cloud messaging, authentication, storage, hosting,

[2] https://firebase.google.com/

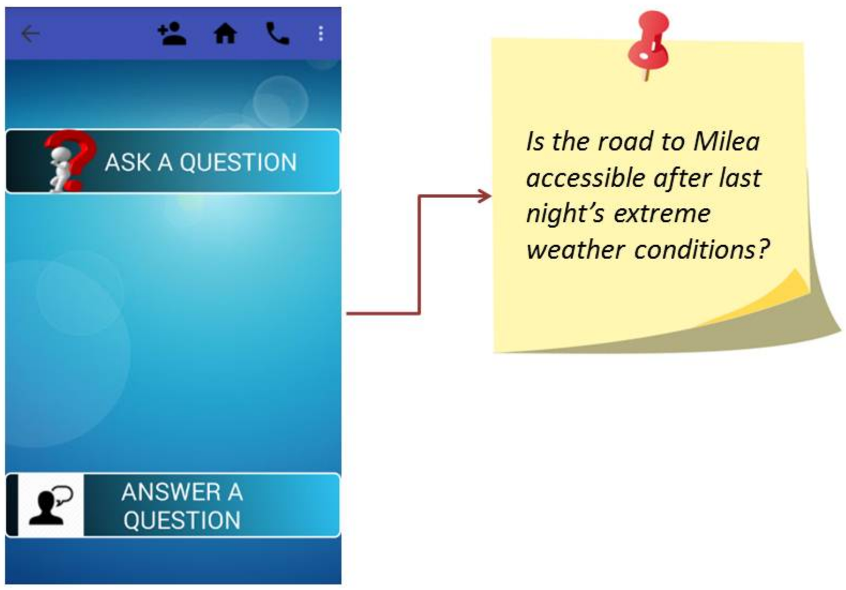

ASK A QUESTION
ANSWER A QUESTION
Is the road to Milea accessible after last night's extreme weather conditions?

Figure 4.5: Ask and answer.

crash reporting and test labs. Its three core services are: a real-time database, user authentication, and hosting.

In the first version of CommonTasker we have used the real-time database and the storage services of Firebase. The *real time database* provides a back end service and allows for all connected users to be synchronized and receive updates as they happen. The *Firebase storage* is used for storage of user-generated content, such as photos, videos.

4.4.3 Libraries

The list of libraries that have been used in the development work for this first release of CommonTasker include:

com.android.support:recyclerview-v7:23.4.0 The recyclerview library provides support for efficiently displaying large data sets i.e. handling large list items and includes RecyclerView.Adapter and RecyclerView.LayoutManager to be able to detect data set changes in batches during a layout calculation and handle animations.

com.android.support:cardview-v7:23.0.0 Cardview library deals with layout features and components of the application. It also permits for showing information inside cards in order to have a consistent look i.e. user profile and sub-entities and additional pieces. These cards are useful for material design implementations, and are used extensively in layouts for TV apps. In CommonTasker the layout for answering a question, accepting a task, pressing a list item and receiving task details are supported by the cardview library.

com.flaviofaria:kenburnsview:1.0.7 This is an Android library that provides classes for animation using the Ken Burns Effect. An example from using this library is the splash screen of

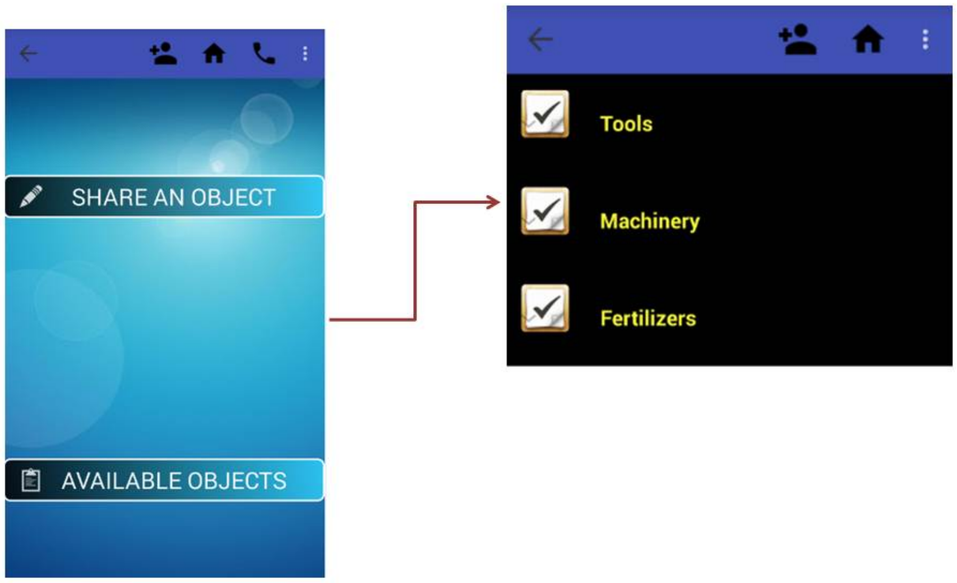


Figure 4.6: Objects.

CommonTasker.

com.android.support:support-v4:23.0.0 This is a core Android library. It includes a large set of APIs, including support for application components, user interface features, accessibility, data handling, network connectivity, and programming utilities. The application design and called functions are all part of this library.

com.android.support:appcompat-v7:23.4.0 This library adds support for the Action Bar user interface design pattern. The toolbar of CommonTasker is implemented using this the appcompat library.

com.android.support:design:23.0.0 This library includes components such as navigation drawer view, floating labels for editing text, a floating action button, snackbar, tabs, and a motion and scroll framework. It is practically a library that supports other design libraries i.e. cardview.

com.android.support:palette-v7:23.0.0 The v7 Palette support library includes the Palette class, which permits extracting prominent colors from an image. Colors used can either be defaults ones or created by the developer of the app.

com.squareup.picasso:picasso:2.3.2 Images add much-needed context and visual flair to Android applications. Picasso allows for hassle-free image loading.It loads and adjust images to the screen, crops them where needed etc. In CommonTasker adjustment of imaged can be found in the splash screen, the welcome screen, entries with pictures etc.

google.firebase:firebase-storage:9.4.0 Library Storage. Firebase Storage lets you upload and store user generated content, such as files, and images.

com.google.firebase:firebase-database:9.4.0 Library Realtime Database. The Firebase Realtime Database lets you store and query user data, and makes it available between users in realtime.

com.firebase:firebase-client-android:2.3.1 Library for client features on firebase system and compatibility.

com.firebaseui:firebase-ui-database: Library for user interface on database for compability

4.5 Source code and testing

The focus of this first version of the CommonTasker is the client side. The users can download the app-debug.apk file on their smartphones and check step-by-step a number of things. First of all, users can see whether the application has been installed and launched properly. Secondly, they can check the design of the interface and its functionalities, i.e. for every displayed screen observe its loading time, adaptation to the display size, readability of the texts, proper clickability of all the buttons, proper display and loading of integrated pictures and photos, animations, background and foreground colors, etc. Thirdly, users can observe the available actions of the toolbar and navigate within the application. Lastly, also of importance at this initial testing stage are the usability of the application under different broadband connection circumstances (Wi-Fi,3G/4G) and its response to connectivity interruptions.

The code and the corresponding apk file (app-debug.apk) to install the application can be downloaded from:
`https://github.com/netCommonsEU/CommonTasker/releases`
The detailed testing instructions of CommonTasker can be found at:
`https://github.com/netCommonsEU/CommonTasker-Testing/blob/master/Testing_instructions.md`
The source code for creating the first version of CommonTasker can be found at:
`https://github.com/netCommonsEU/CommonTasker`

5 Conclusions and Future Work

This document has summarised the work done in T3.2/3.3/3.4 during the first year of netCommons. Such work consists in open source code, measures of services deployment on real networks and the design of a participatory development process together with an active Community Networks. The work done so far will be extended in the next years in the following ways:

Cloudy Future development in netCommons will target several aims: maintenance, modularisation, resource pooling and service deployment, application integration, and coordination with other initiatives.

Maintenance: updates and fine-tuning to improve its reliability, performance, security, and user friendliness, according to the feedback from usage of the community of existing and new users.

Further *modularisation*, with a core part and the "containerisation" of remaining components to achieve higher isolation. This brings multiple benefits for a) the software development process to facilitate integration, b) for the operation, with performance and security advantages, and c) for the customizability by end-users that could select and activate new containers autonomously without any dependency or change on the base distribution.

Integration with the Community-Lab testbed for experimentation to allow *resource pooling and dynamic service deployment*: Cloud users could contribute raw computing and storage resources to a common pool that could then be then used by community cloud service providers to a) select a range of resources, b) deploy and operate multiple instances of a given service in multiple Cloudy instances, and c) dynamically adjust the number of instances according to service performance and demand.

Integration of further *applications* and services demanded by Community Networks (CNs). *Coordination* with other global initiatives (e.g. APC, DC3) in CNs to develop cloud infrastructures for service provision.

PeerStreamer The development of PeerStreamer will follow three directions in the second and third year of netCommons. As a first direction we will introduce into PeerStreamer the support for multiple video streams. Currently PeerStreamer supports only one source of video, but our intention is to test PeerStreamer as a P2P video conference system. This requires the introduction and management of multiple video source in the same P2P swarm which was never considered before in PeerStreamer. As a second direction we will consolidate the implementation of PeerViewer and test in in some real-world scenario. We are in contact with some Community Networks that have expressed their interest for PeerStreamer/PeerViewer and we will cooperate with them to have some real deployment. As a third direction we will consider the integration of PeerViewer in the Cloudy distribution. Currently only the original PeerStreamer binary is supported, but the more agile PeerViewer soultions should be possible to integrate via the Docker system.

CommonTasker The work on CommonTasker will proceed in several directions. A first thread of work will relate to the development of the server part of the application, which currently is emulated by Firebase functionality. Secondly, the application design will be updated according to the feedback received during the November workshop in the Sarantaporo valley. This means exploring the possibility of teaming up with an ICT company providing smart farming advice and solutions to collect data from farms in the area. For CommonTasker this implies putting emphasis on the crowdsourced data collection and sharing component but also decisions about the data storage facilities (e.g., locally vs. over a cloud solution). Finally, the participatory design process will continue as an integral and important part of the software development process, with more opportunities to interact with the local community during the second year of the project.

Bibliography

[1] 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, pp. 154–163, 06/2011 2011. [Online]. Available: http://dx.doi.org/10.1109/MCOM.2011.5784001

[2] M. Efthymiopoulou, N. Efthymiopoulos, A. Christakidis, S. Denazis, and O. Koufopavlou, "Scalable control of bandwidth resources in P2P live streaming," in *22nd Mediterranean Conference on Control and Automation*, June 2014, pp. 792–797. [Online]. Available: http://dx.doi.org/10.1109/MED.2014.6961470

[3] A. Russo and R. Lo Cigno, "Delay-Aware Push/Pull Protocols for Live Video Streaming in P2P Systems," in *IEEE International Conference on Communications (ICC 2010)*, May 2010. [Online]. Available: http://dx.doi.org/10.1109/ICC.2010.5502543

[4] R. Baig, J. Dowling, P. Escrich, F. Freitag, R. Meseguer, A. Moll, L. Navarro, E. Pietrosemoli, R. Pueyo, V. Vlassov, and M. Zennaro, "Deploying clouds in the guifi community network," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 1020–1025.

[5] B. Braem, R. Baig Viñas, A. L. Kaplan, A. Neumann, I. Vilata i Balaguer, B. Tatum, M. Matson, C. Blondia, C. Barz, H. Rogge, F. Freitag, L. Navarro, J. Bonicioli, S. Papathanasiou, and P. Escrich, "A case for research with and on community networks," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 3, pp. 68–73, Jul. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2500098.2500108

[6] L. Maccari and R. Lo Cigno, "A week in the life of three large wireless community networks," *Ad Hoc Networks, Elsevier*, vol. 24, Part B, pp. 175–190, Jan. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.adhoc.2014.07.016

[7] "The NIST Definition of Cloud Computing". National Institute of Science and Technology, U.S. department of commerce." [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[8] A. Marinos and G. Briscoe, "Community Cloud Computing," *Computing*, vol. 5931, no. December, p. 11, Jul. 2009.

[9] M. Selimi, A. M. Khan, E. Dimogerontakis, F. Freitag, and R. P. Centelles, "Cloud services in the guifi.net community network," *Computer Networks*, vol. 93, Part 2, pp. 373 – 388, 2015, community Networks. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128615003175

[10] M. Selimi, F. Freitag, R. Pueyo, and A. Moll, "Distributed storage and service discovery for heterogeneous community network clouds," in *7th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2014)*. IEEE, Oct. 2014.

[11] M. Selimi, N. Apolonia, F. Olid, F. Freitag, L. Navarro, A. Moll, R. Pueyo, and L. Veiga, "Integration of assisted p2p live streaming service in community network clouds," in *7th IEEE International Conference on Cloud Computing Technology and Science CloudCom 2015))*, Dec. 2015, pp. 113–120. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2012.6427547

[12] M. Selimi, F. Freitag, R. Pueyo Centelles, A. Moll, and L. Veiga, "Trobador: Service discovery for distributed community network micro-clouds," in *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on*, March 2015, pp. 642–649. [Online]. Available: http://dx.doi.org/10.1109/AINA.2015.248

[13] N. Apolónia, R. Sedar, F. Freitag, and L. Navarro, "Leveraging Low-Power Devices for Cloud Services in Community Networks," in *2015 3rd International Conference on Future Internet of Things and Cloud*, Aug 2015, pp. 363–370. [Online]. Available: http://dx.doi.org/10.1109/FiCloud.2015.49

[14] M. Selimi, D. Vega, F. Freitag, and L. Veiga, *Towards Network-Aware Service Placement in Community Network Micro-Clouds*. Cham: Springer International Publishing, 2016, pp. 376–388. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-43659-3_28

[15] D. Vega, R. Baig, L. Cerdà-Alabern, E. Medina, R. Meseguer, and L. Navarro, "A technological overview of the guifi. net community network," *Computer Networks*, vol. 93, pp. 260–278, 2015.

[16] L. Abeni, C. Kiraly, and R. Lo Cigno, "On the Optimal Scheduling of Streaming Applications in Unstructured Meshes," in *8th Int.łIFIP-TC 6 Networking Conference (NETWORKING)*, Berlin, Germany, May 2009, pp. 117–130.

[17] R. Lobb, A. Couto Da Silva, E. Leonardi, M. Mellia, and M. M., "Adaptive Overlay Topology for Mesh-Based P2P-TV Systems," in *ACM NOSSDAV*, Williamsburg, Virginia, USA, June 2009, pp. 31–36.

[18] A. Couto Da Silva, E. Leonardi, M. Mellia, and M. M., "Exploiting Heterogeneity in P2P Video Streaming," *IEEE Trans. on Computers*, vol. 60, no. 5, pp. 667–679, May 2011. [Online]. Available: https://doi.org/10.1109/TC.2010.244

[19] L. Abeni, C. Kiraly, and R. Lo Cigno, "Robust Scheduling of Video Streams in Network-Aware P2P Applications," in *2010 IEEE International Conference on Communications (ICC)*, May 2010, pp. 1–5. [Online]. Available: https://doi.org/10.1109/ICC.2010.5502542

[20] L. Abeni, A. Bakay, M. Biazzini, R. Birke, E. Leonardi, R. Lo Cigno, C. Kiraly, M. Mellia, S. Niccolini, J. Seedorf, T. Szemethy, and G. Tropea, "Network friendly p2p-tv: The napa-wine approach," in *2010 IEEE International Conference on Peer-to-Peer Computing: (P2P)*, 2010, pp. 1–2. [Online]. Available: http://dx.doi.org/10.1109/P2P.2010.5569983

[21] C. Kiraly, L. Abeni, and R. Lo Cigno, "Effects of P2P Streaming on Video Quality," in *2010 IEEE International Conference on Communications (ICC)*, May 2010, pp. 1–5. [Online]. Available: https://doi.org/10.1109/ICC.2010.5502003

[22] G. Ciccarelli and R. Lo Cigno, "Collusion in Peer-to-Peer Systems," *Elsevier Computer Networks*, vol. 55, no. 15, pp. 3517–3532, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2011.06.028

[23] R. Birke, C. Kiraly, E. Leonardi, M. Mellia, M. Meo, and S. Traverso, "Hose rate control for P2P-TV streaming systems," in *2011 IEEE International Conference on Peer-to-Peer Computing: (P2P)*, Kyoto, Japan, August 2011, pp. 202–205.

[24] S. Traverso, L. Abeni, R. Birke, C. Kiraly, E. Leonardi, R. Lo Cigno, and M. Mellia, "Experimental comparison of neighborhood filtering strategies in unstructured P2P-TV systems," in *12-th IEEE International Conference on Peer-to-Peer Computing (P2P'12)*, Sept. 2012, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1109/P2P.2012.6335794

[25] L. Baldesi, L. Maccari, and R. Lo Cigno, "Live P2P streaming in CommunityLab: Experience and Insights," in *13th IEEE Annual Mediterranean Ad Hoc Networking*

Workshop (MED-HOC-NET), Piran, Slovenjia, June 2014, pp. 23–30. [Online]. Available: http://dx.doi.org/10.1109/MedHocNet.2014.6849101

[26] 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, June 2015. [Online]. Available: http://dx.doi.org/10.1109/TNET.2014.2307157

[27] L. Baldesi, L. Maccari, and R. Lo Cigno, "Improving P2P streaming in community-lab through local strategies," in *IEEE 10th Int. Conf. on Wireless and Mobile Computing, Networking and Communications (WiMob)*, Larnaca, Cyprus, Oct. 2014, pp. 33–39.

[28] L. Baldesi, L. Maccari, and R. Lo Cigno, "Improving P2P streaming in Wireless Community Networks," *Computer Networks, Elsevier*, vol. 93, Part 2, p. 389–403, Dec. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2015.09.024

[29] L. Maccari, L. Baldesi, R. Lo Cigno, J. Forconi, and A. Caiazza, "Live Video Streaming for Community Networks, Experimenting with PeerStreamer on the Ninux Community," in *Proceedings of the 2015 Workshop on Do-it-yourself Networking: An Interdisciplinary Approach (DIYNetworking '15)*, May 22, 2015, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1145/2753488.2753491

[30] L. Baldesi, L. Maccari, and R. Lo Cigno, "Optimized Cooperative Streaming in Wireless Mesh Networks," in *15th IFIP Networking Conference (NETWORKING)*, Vienna, Austria, May 2016.

[31] B. Sredojev, D. Samardzija, and D. Posarac, "Webrtc technology overview and signaling solution design and implementation," in *Proc. 38th Int Information and Communication Technology, Electronics and Microelectronics (MIPRO) Convention*, May 2015, pp. 1006–1009. [Online]. Available: http://dx.doi.org/10.1109/MIPRO.2015.7160422

[32] W. W. W. Consortium, "HTML5," https://www.w3.org/TR/2014/REC-html5-20141028/, 2014.

[33] Panayotis Antoniadis – Editor, "Multi-Disciplinary Methodology for Applications Design for CNs, including Design Guidelines and Adoption Facilitation (v1)," netCommon Project, Deliverable 3.1, Dec 2016. [Online]. Available: http://netcommons.eu/?q=content/multi-disciplinary-methodology-applications-design-cns-including-design-guidelines-and

[34] Panagiota Micholia: Editor, "Incentives for Participation and Active Collaboration in CNs (v1)," netCommon Project, Deliverable 2.3, Dec 2016. [Online]. Available: http://netcommons.eu/?q=content/incentives-participation-cns-v1

The netCommons project

December 31, 2016

netCommons-D3.2-1.0

Horizon 2020