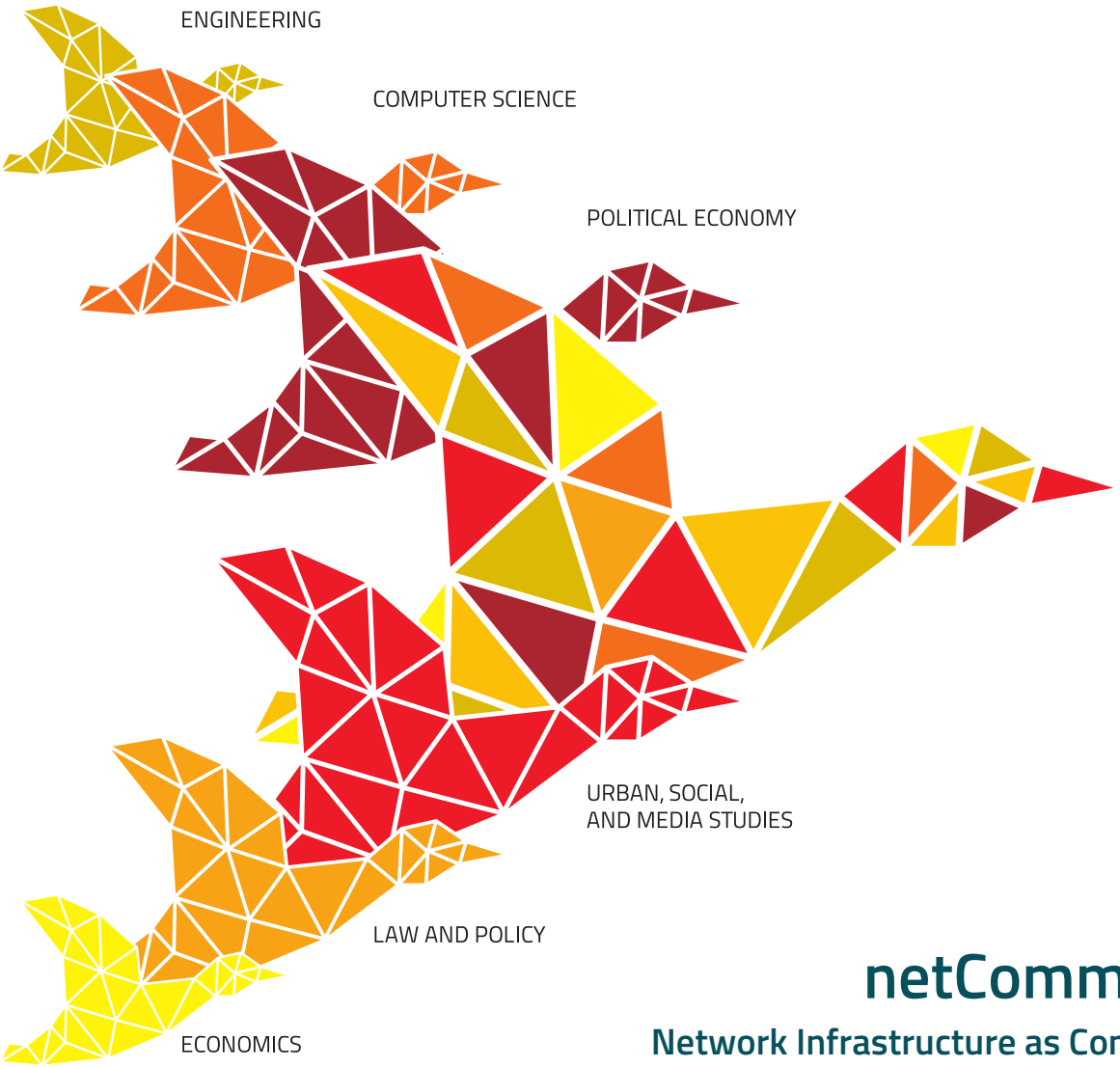


netCommons

Network Infrastructure as Commons

Release of New Open Source Software for All Applications

Deliverable Number D3.4 – Interim release
Version 0.5
August 9, 2017

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

netCommons.eu

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

Editor:	Leonardo Maccari, UniTN
Deliverable nature:	Report (R)
Dissemination level:	Public (PU)
Contractual Delivery Date:	December 31st, 2017
Actual Delivery Date	June 30st, 2017
Number of pages:	30
Keywords:	community networks, dissemination
Authors:	Leonardo Maccari, Luca Baldesi, Nicolò Facchi, Renato Lo Cigno UniTN Felix Freitag, UPC Merkouris Karaliopoulos, Panagiota Micholia, Aris Pilichos, AUEB-RC
Peer review:	—

History of Revisions

Rev.	Date	Author	Description
v0.1	12/6/2017	Leonardo Maccari	First draft
v0.2	15/6/2017	Felix Freitag	Contributions about progress in Cloudy
v0.3	15/6/2017	Leonardo Maccari	First draft with T3.3 progress
v0.4	30/6/2017	M. Karaliopoulos, P. Micholia, A. Pilichos	First draft with T3.4 progress
v0.5	30/6/2017	Renato Lo Cigno	Revision and editing for web-site publication

Executive summary

This is an interim release documenting the activities of T3.2, T3.3, and T3.4, the software development tasks of WP3. It describes the work done in the first half of the second year of netCommons, so it updates the work officially reported in D3.2 [1]. Information reported here and will be the base for the final version of D3.4 to be published at M24.

The three tasks continued in the development of their software, or advanced the research studies that will support new development in the rest of the project. The deliverable is structured as follows:

T3.2 produced two new publications based on the Cloudy platform that improved the monitoring system and put the bases for smart Proxy management (in Chapter 2);

T3.3 integrated PeerStreamer (PS) with the new docker-based Cloudy platform and Serf (in Chapter 3);

T3.4 continued the development of the CommonTasker application, based on the feedback received from the Sarantaporo community (in Chapter 4).

Contents

List of Figures

%let

%listoftables

List of Acronyms

CN	Community Network
PS	PeerStreamer
P2P	Peer-to-Peer
SDP	Session Description Protocol
RTP	Real-time Transport Protocol
VLC	Video-Lan Client
ISP	Internet Service Provider

1 Introduction

The second year of netCommons, for WP3 is the year in which the software needs to reach a state of usability, sufficient to be deployed in some test Community Networks (CNs). The third year will mostly focus on monitoring the use of these applications in the real scenarios, maintaining and fixing the code, and gathering feedback from the users.

Thus, it is important to give an intermediate update on the state of tasks, in order to understand what to expect from the second half of the project, and from the next six months of development. The work done in WP3 differs from task to task. For instance, T3.2, starting from an already robust code-base and running nodes in real CNs, focused on advances and further monitoring of Cloudy application in real networks, in order to design new features. T3.3 pushed on the integration of PS in Cloudy and its ecosystem of communication protocols, in order to make seamless the addition and fruition of new live streams in Cloudy. T3.4 finally developed furthermore the connection with the Sarantaporo community and thus, the customization to a real use case, that is smart farming.

Each of the following chapters describe the state of the work and will be further expanded in the final version of this deliverable at M24.

2 Update on the Cloudy Distribution

This Chapter describes recent work on community clouds as part of the work of WP3 in the netCommons project, specifically task T3.2.

As to the DoW, *"T3.2 will research on Community Clouds, the application of distributed cloud systems to CNs."* It combines the tasks of the evaluation of the community cloud system, in order to assess its position and needs, and the developments of components to enable new possibilities with the platform.

In year 2 in T3.2, in the community cloud platform, we had as starting point the initial support of some Docker-provisioned services in Cloudy, and learned from the experiences with the Cloudy operation and service usage, in order to understand and decide on future developments and orientations. Some specific works are summarized below, addressing service monitoring in Cloudy and the guifi.net proxy service.

Service monitoring: Information on service availability in Cloudy is important to design and enable new features. Therefore, it was considered important to understand better the options for a monitoring system in Cloudy. For this purpose we researched and evaluated a monitoring system leveraging the gossip platform used in Cloudy. This work has led to the following publication:

1. "Gossip-based Service Monitoring Platform for Wireless Edge Cloud Computing", N. Apolónia, Freitag, F., and Navarro, L., Proc. of the *14th IEEE International Conference on Networking, Sensing and Control (ICNSC)*, Calabria, Italy, 2017 [2].

Guifi proxy service: While a community network may offer different applications, Web access is the most popular application in guifi.net. Web proxy nodes connected both to guifi.net and an ISP are often used by community network users as free Internet gateways. Many of these proxies run on simple servers at the homes of the community members. In the Cloudy distribution, this proxy service was integrated at the beginning of the Cloudy development and is part of a set of services for network management offered by Cloudy.

One of the limitations of the current proxy service in guifi.net is that proxies are selected manually by the users. This may have the effect that many users choose and burden "popular" proxies. As a consequence, potential hotspots can reduce the quality of the user experience. In addition, the non-optimal usage of the access links saturates the available bandwidth of these links and the shared bandwidth to the ISP, while the community member who voluntarily shares her connectivity may suffer performance degradation in her Internet access service. This field has an important potential for Cloudy since understanding how to build an automated proxy assignment could lead to the design of a component, which supports this functionality in the Cloudy's proxy service. Several on-going research work items, which we have been conducting in this field, have been published recently, including:

1. "Client-Side Routing-Agnostic Gateway Selection for heterogeneous Wireless Mesh Networks", ME. Dimogerontakis, Neto, J., Meseguer, R., Navarro, L., and Veiga, L., Proc. of the *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Lisbon, Portugal, 2017 [3].

Next steps in T3.2 are expected to include further developments in the Cloudy platform in order to design and develop the components that increase the flexibility of service deployments. We also consider exploring new application domains that could benefit from the services provided by Cloudy devices. Among these domains we can identify the Internet of Things (IoT), edge computing capabilities, and components which may contribute to privacy, security and edge data analytics.

3 Update on PeerStreamer

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

3.1 Motivation

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

We believe that PeerStreamer can contribute to the diffusion of Cloudy, and, at the same time, it can benefit from the the Cloudy platforms in terms of simplified installation and management. Thus, after D3.1, where we focused on the realization of a stripped-down version of PeerStreamer that can be installed directly on network nodes, we now shifted our attention in two areas:

- The better integration of PeerStreamer into Cloudy;
- The initial design of the extension of PeerStreamer as a multi-stream application, that lies the at the base of the work needed to transform PeerStreamer into a Peer-to-Peer (P2P) video conferencing system.

3.2 Integration of PeerStreamer into Cloudy

Cloudy is a complex system that interconnects three main components:

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

Docker is a well known container solution that allows to run processes in separated environments. The services on a Cloudy distribution can be expanded adding containers with new applications that run locally in the user's Cloudy node. Serf is a gossip-based membership protocol, that makes it possible to distribute in a decentralized way information among groups of members. Information can relate to the current composition of the group (addition and removal of members) and custom events that one member generates and wants to distribute to the other nodes. Cloudy uses Serf to distribute the knowledge about a new service in the network, after a new container has been downloaded and run by a node. Finally, Cloudy relies on a running CN to work and assumes that IP addresses are all routable and not NAT-ted.

PeerStreamer (PS, for brevity) is a live P2P streaming platform, already described in the previous deliverable [1]. It had been already integrated by Cloudy before it switched to Docker. PS uses its own gossiping system to distribute the video chunks among the swarm of peers that participate in the streaming session, and thus, apart from being able to run and stop the service, the integration with Cloudy was only partial. After the work to

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

integrate PS in the wireless router [1], in this deliverable we describe our effort to integrate PS in Cloudy, thus, addressing another use-case.

The main sub-tasks of this activity were:

- Made PeerStreamer a library. PeerStreamer became a set of independent components that make the application much easier to be integrated on other platforms. This is a revolution in the PS architecture that was designed to run in a separate application, and forms the first step of the work needed to include multi-stream support in PS, useful also for the future development of PS as a conference system (see Sec. 3.3). For this reason we renamed this system as next-generation PeerStreamer, PS-ng.
- Dockerized PeerStreamer. We realized a custom Docker image for PS, based on the Ubuntu 16.04 image that can be downloaded from the Docker Hub[2]. The image contains PS, which is run in a container and appears among the available services in Cloudy.
- We integrated PeerStreamer with the Serf protocol: We realized a C language interface to make Serf communicate with PeerStreamer, so that new live video streams can be published on Serf (currently experimental). So, not only Cloudy can be used to run PS, but Serf can be used by the instances of PS to notify each other about the presence of new videos. Currently the source of the video is run out of the Cloudy platform, but future developments will integrate it within Cloudy.

3.2.1 PS-ng

Fig. 3.1 reports the configuration we imagine for PS-ng integrated into Cloudy. The home network of a user is provided by his wireless router, that is connected to the CN and to the Cloudy device. The latter connection can be wireless or wired. In some network node (possibly remote) there is a video streaming source that can be received from any node in the network. When a Cloudy node is turned on, it will connect to the Serf swarm and receive the list of the other running Cloudy instances. This is the default behavior of Cloudy as described in [1].

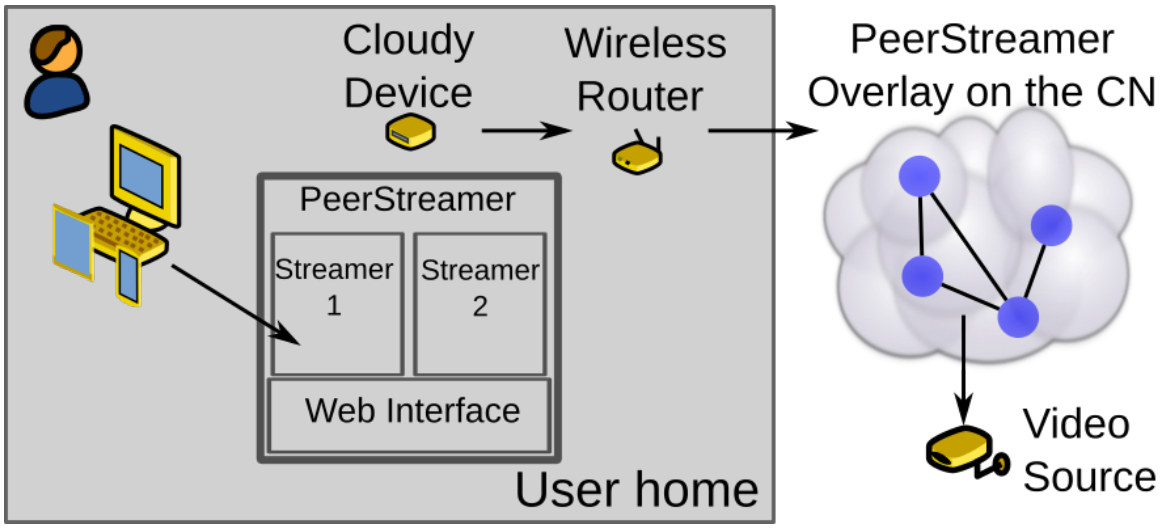


Figure 3.1: The components of the live streaming inside a home network.

The user will connect with a browser to the Cloudy device, and, using the configuration panel of Cloudy, he can download the PS docker image. Among the list of available services PS will appear as Fig. 3.2 shows.

Once PS-ng is started inside Cloudy, it will use again Serf to list the available videos at the moment (possibly more than one). The user will connect through his web browser to the PS web interface directly from the

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

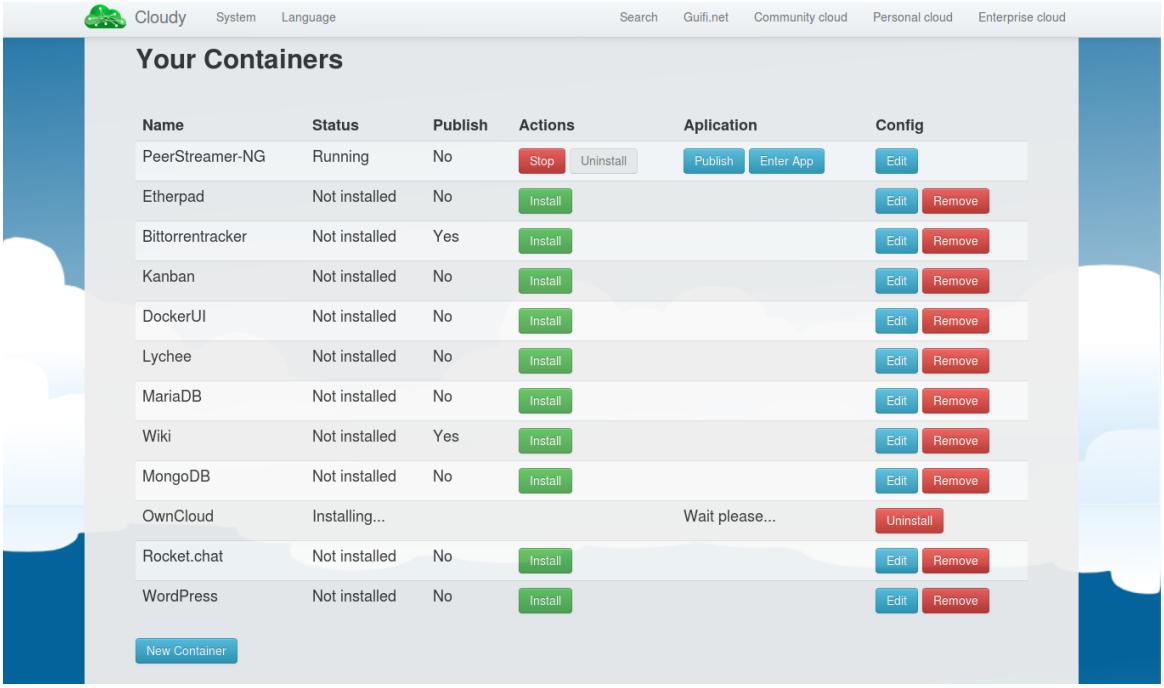


Figure 3.2: The list of available services in the Cloudy interface.

Cloudy interface, and will interact with the PS web page running on Cloudy. He will be able to see the list of currently running videos and choose one to play in his browser, or in his dedicated application (such as the widely used Video LAN Client open source program) as Fig. 3.3 shows.

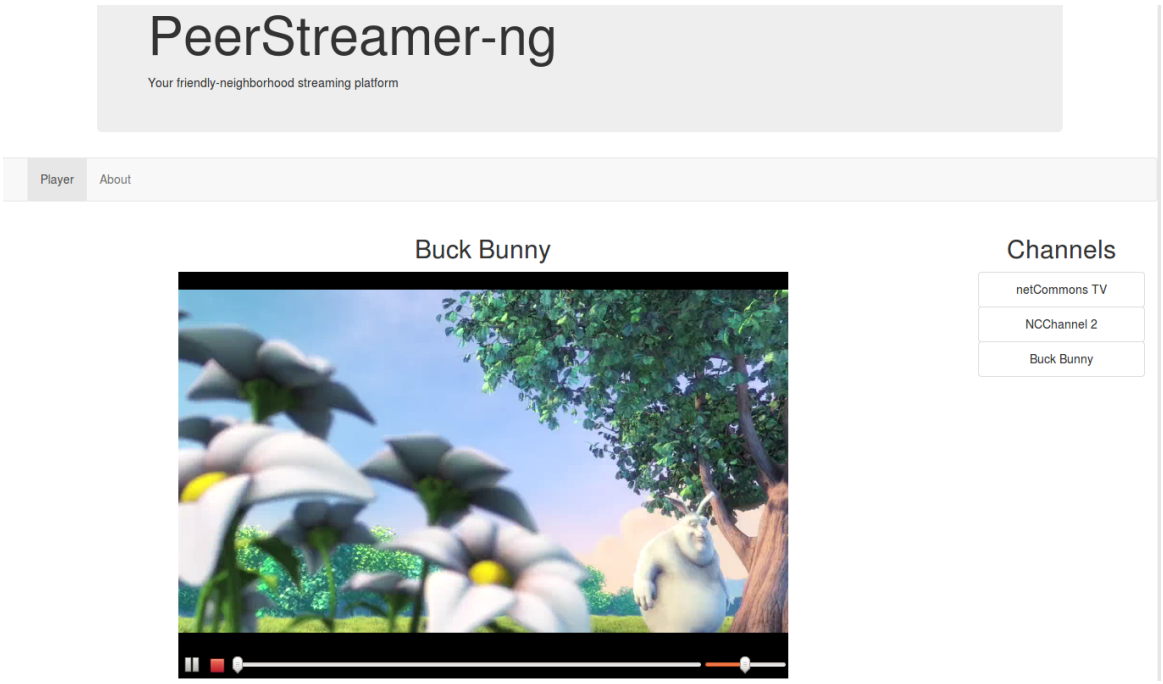


Figure 3.3: The web interface of PS-ng.

In principle, more than one users can access the local Cloudy device, receive the list of channels and see a different video at the same time. This feature is currently under testing.

Firefox plug-ins issues The original design of PS-ng involved the use of Real-time Transport Protocol (RTP) to carry the video chunks encapsulated in the PS messages. This is particularly convenient because several widely used open source programs, such as Video-Lan Client (VLC) support the generation and the playout of video in the RTP format. Any browser, for instance, can be equipped with the VLC plugin that supports in-browser reproduction of RTP videos. During the development of PS-ng all most popular browsers changed this approach, due to security reasons, and today, the procedure of enabling the VLC plugin has become cumbersome. While the HTML5 standard supports embedding RTP directly into HTML, browsers do not support it directly, and today the most popular way of delivering video is by embedding it into a TCP stream inside HTML, or with the WebRTC protocol. Both approaches have their limitations, as documented in [1], and in the future months we will evaluate what is the most effective way of realizing the web-based player. For the current state of things, in case the browser does not support the VLC plugin, we automatically present to the user, on the PS web-interface, the URL that can be used by any recent media player to play RTP, as Fig. 3.4 shows.

link here

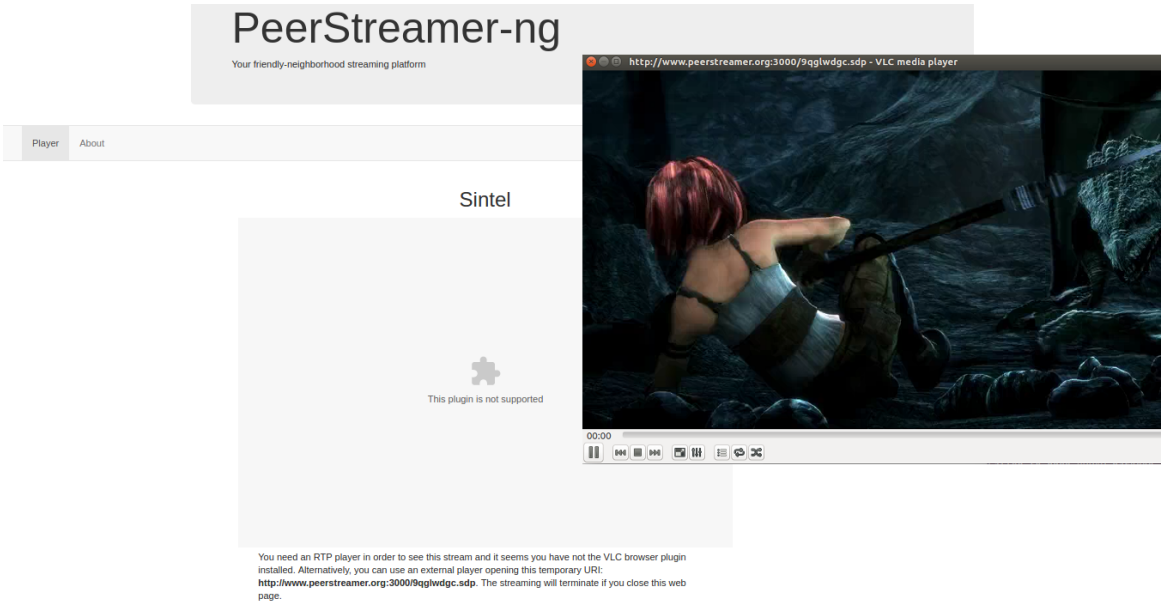


Figure 3.4: The web interface of PS-ng showing the URL to the video descriptor, and the VLC program playing the video (note the URL bar on top of the window frame).

3.2.2 Internals of PS-ng

Fig. 3.5 reports the architecture of PS-ng, integrated into Cloudy broken down into its functions.

PS-ng has been designed with modularity in mind and it leverages the flexibility of its carefully chosen components. The core of the P2P streaming algorithms is based on the GRAPES library [4].

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

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

We chose HTTP for controlling as it is a commonly used solution for providing user-friendly, highly portable GUI and it does not require any installation burden for the users. PS-ng implements a ReST interface upon the

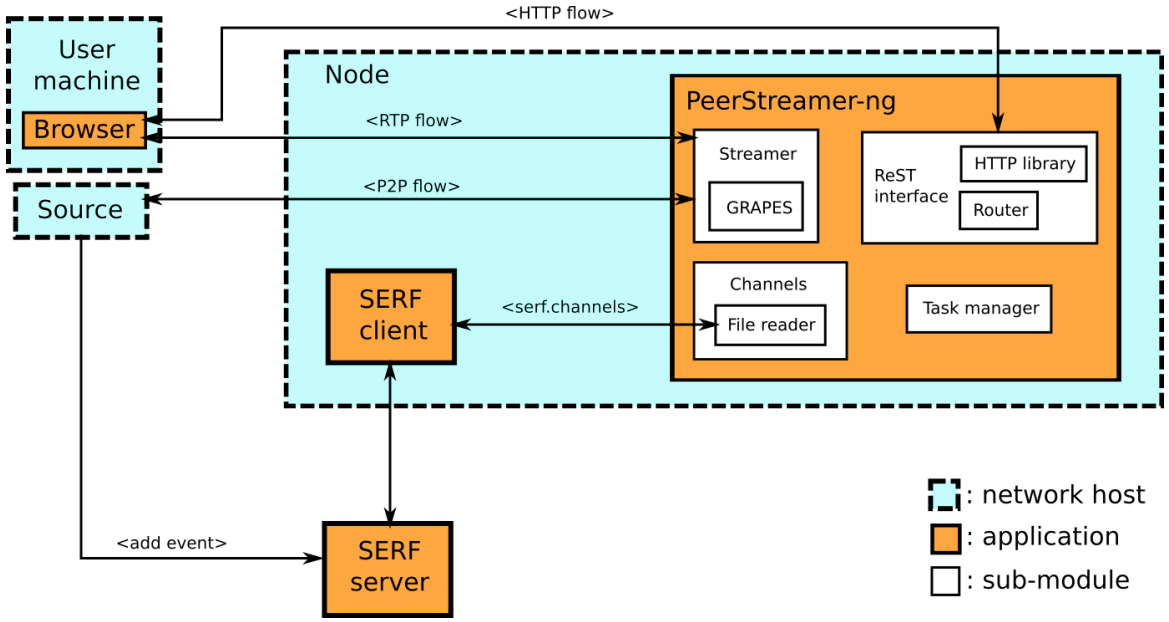


Figure 3.5: The architecture of PS-ng and the integration with Cloudy.

Mongoose HTTP library[3], which has been designed to be lightweight and highly portable, serving the GUI as HTML/Javascript document and exposing the controlling interface:

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

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

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

- handling HTTP requests
- refreshing channel list from the SERF client
- handling the PS P2P communication

3.3 Design of the support for Multi-Stream in PeerStreamer

Another goal of T3.3 is to experiment with PS to create a P2P video-conferencing system. This is a challenging task that involves several sub-tasks, given that PS was initially designed as a P2P-TV system, in which a single video source is streamed in the P2P network. This task will be the main body of work in the second half of the second year of D3.3, in the first half we started to design and add the needed components to PS.

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

The first step was to choose a standard descriptor for video streams. In fact, if PS is to be used with multiple streams, the user will have to collect information about the running streams. Note that this is a different problem than the one already solved using Serf in the Cloudy distribution. In that case, there is one set of peers per stream, meaning that for each source, the group of peers that exchange the video chunk is different (the group of peers is called an *overlay* in the P2P terminology). When a PS instance joins an overlay, it will receive one video and to build the list of available videos we use Serf, as already mentioned. In the case of video conference instead, there is one overlay that delivers more than one video, and inside the overlay PS must be able to distinguish one video stream from the other. Thus, we need to choose a format to describe the video streams and mark each video chunk with an ID per video stream.

We chose Session Description Protocol (SDP), a standardized format for describing video sessions [5] that several video players already support. SDP is human readable, and a typical descriptor is made of a subset of the fields reported in Listing 3.1.

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

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

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

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

Listing 3.1: The available fields in the SDP format.

The SDP description was added to the topology description messages in PS. Once a peer enters a new overlay, it will poll other peers with topology messages. These messages are used to ask and receive a set of IP addresses of available peers in the overlay, in order to build a partial view of the PS overlay topology, sufficient to participate to the video chunk distribution. We added to the topology messages an optional list of the available session identifiers, with a pointer to a known peer that is currently distributing it. Each session identifier represents one video stream and consequently one SDP descriptor. Once the new peer has collected a list of identifiers, it will collect the SDP asking them directly to the corresponding peers. The SDP contains all the needed information

on the quality and encoding of the video, and the information on the source node. Each video chunk will also contain an additional field that is the corresponding identifier.

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

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

At the current state of the work we have implemented step 1 to 4, and we are in the phase of testing it on small networks. In the next months we will modify the PS interface to show the videos of the ongoing streams. This task is one of the most challenging since PS is based on RTP video, as already described, but currently, browser support for UDP-based video streams is poor. Several solutions are under analysis; more detail will be provided in the next version of this deliverable.

3.3.1 PS-ng on the Android platform

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

The current limitation resides in the fact that Android-based browsers do not support the VLC plugin and we need a second app playing the videos, as in the configuration depicted in Fig. 3.4. In our current browser-based design we require the browser to be active during the stream, in order to detect when the user is watching the video and stop streaming when he/she leaves. When the browser is killed (or stopped) the Javascript contained in the web-page stops loading and the video is not streamed anymore. This is not an issue in standard PC, but, since Android phones send the browser to sleep when a new app is launched, we have an additional difficulty in the current set-up to use PS-ng on mobile devices.

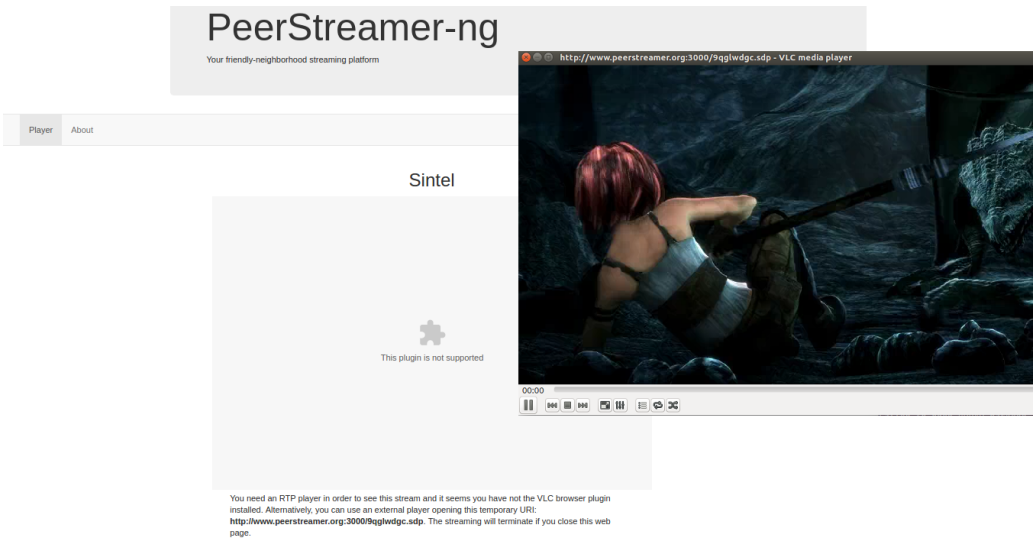


Figure 3.6: An android phone playing a video with PS-ng. Note the necessary split-screen with the browser and the VLC mobile app running at the same time.

Fig. 3.6 depicts a video stream played on an Android device, note that there are two applications running, a browser and the VLC app in parallel. This configuration is not really usable but shows that the building blocks of PS-ng are compatible with the Android mobile environment. As a future work we will evaluate if it is possible to develop a native player for Android.

3.4 New Research results on P2P Video streaming on CN

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

- Leonardo Maccari, Nicolò Facchi, Luca Baldesi, Renato Lo Cigno: "Optimized P2P Streaming for Wireless distributed Networks", accepted for publication in the Elsevier Journal of Pervasive and Mobile Computing.
- Luca Baldesi, Leonardo Maccari, Renato Lo Cigno: "On the Use of Eigenvector Centrality for Cooperative Streaming", accepted for publication in IEEE Communications Letters.

The two proposals are complementary and are under implementation in an experimental branch of the PS-ng software. In this section we briefly summarize their content, that can be found in the URLs in the bibliography [6, 7].

3.4.1 Optimized P2P Streaming for Wireless distributed Networks

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

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

In [6], we proposed a cross-layer optimization scheme to perform *live* video streaming (i.e., with a strict deadline on the arrival delay) in mesh networks. The optimization minimizes the impact of the streaming overlay on the underlay network exploiting information on the topology and the routing of the underlay. The optimization is based on the concept of *centrality*, taking into account the centrality of peers in the underlay graph, the optimized overlay topology greatly improves the efficiency of the video distribution and maintains high performance.

More specifically, the paper key contributions are as follows:

- We formalize an optimization problem that, given the underlay of the P2P network and the peers in the overlay, finds the overlay topology whose cost on the underlay is minimum. We define the cost of the overlay as a combined metric composed by the load and the fairness imposed by the overlay on the underlay;
- We prove that the optimal strategy is NP-complete by reducing it to a quadratic knapsack problem;
- We propose two techniques for relaxing the optimal strategy, which relies on the concept of betweenness centrality of the nodes of the underlay. The relaxed strategies are applicable to any wireless mesh network, as long as they use a link-state routing protocol;
- We further improve the performance of the relaxation strategies by introducing a *neighborhood pruning* heuristic, which exploits characteristics often found in real network scenarios;

- We evaluate the proposed relaxation strategies and pruning heuristics through extensive simulations that compute the best overlay, according to each proposed technique, on synthetic network topologies. Simulation results show that the proposed relaxations and heuristics are reasonably close to the optimal solution and largely outperform random overlay building strategies.

The paper builds on a previous work published in Y1 of netCommons, in which we first formalized the optimization problem and then introduced the relaxation strategy [8]. This paper extends these results with further optimizations to make the proposal even more efficient in overlays with a realistic topology, i.e. comprising many leaf-nodes in the network graph.

3.4.2 On the Use of Eigenvector Centrality for Cooperative Streaming

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

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

The problem we tackled in [7] is twofold: i) define the values of τ_i per each node; ii) define a strategy for the choice of the target neighbor. The constraints we posed are that at steady state every node receives the entire content, and at the same time the overall resources used to achieve this goal are minimal.

In [7] we showed that this can be obtained exploiting the *eigenvector centrality* of the normalized adjacency matrix describing G and that there is a performance improvement in terms of reduced packet loss and delivery delay compared to other solutions currently in use. Finally we highlighted that the eigenvector centrality can be efficiently estimated with distributed algorithms, so that the implementation of our solution in real systems is feasible and can scale up to systems made of hundreds of nodes.

4 Update on CommonTasker

4.1 Reiterating on the application scope

The CommonTasker application was built from the beginning with the aim to serve the needs of the Sarantaporo.gr community. The 1st release of the application featured three discrete components, enabling users to crowdsource tasks, exchange information (*e.g.*, question and answers capabilities), and share equipment in line with the sharing economy paradigm, respectively. This release was presented to a group of local people and other interested actors during a symposium that took place in the Sarantaporo area in November [9]. Based on the feedback we got there and the reactions to different topics presented by the symposium participants [9], a decision was made in the project to focus on the data crowdsourcing functionality of the application and develop it further to enable the collection of data for smart farming applications. Smart farming is an umbrella term for practices that leverage the modern ICT technology for optimizing farming tasks. It can generate a number of benefits for farmers such as reducing the production cost and ameliorating the quality and quantity of products, saving environmental resources and minimizing the risk for the farmers.

One of the actors involved in the symposium was GAIA Epicheirein (hereafter referred to as GAIA), a multi-entity alliance with 1/3 of stocks' value held by around 70 agricultural cooperatives across Greece. GAIA offers a plethora of services for farmers ranging from advisory ones, through remote and face-to-face interaction, to mediating for payments of European subsidies. One of their activities evolves around the provision of smart farming services and their presentation on what is possible with it received great interest from the local community.

4.1.1 GAIA's smart farming services and enabling data

A couple of years ago, GAIA launched the provision of experimental smart farming services with the aim to provide them country-wide by 2019. Their platform is built in the cloud and combines four different types of data in order to extract knowledge through smart farming analytics and feed it back to farmers as best practice advice and recommendations (Fig. 4.1). This feedback comes in the form of graphics and verbal advice from experienced agricultural advisors.

One type of data consists in satellite images made openly available by the European Space Agency (ESA) for research purposes. The other two types of data are collected from the field through a network (GAIA SENSE) of telemetric IoT devices called GAIATrons, with appropriate software systems controlling the data acquisition processes. One type of GAIATron monitors environmental conditions (temperature, humidity, air pressure) and a second one monitors ground conditions (soil acidity, penetration of water, ingredients etc). GAIATrons are built with the goal of low cost maintenance, they use solar energy and are portable.

A final fourth type of data, which is most crucial for these services, has to be provided manually by the farmers through GAIA's web application (Work Calendar). The required information includes reports of daily activities in the farm with adequate detail (e.g., time and duration of irrigation, time of lubrication, chemical composition and brand of fertilizer). This last type of data has emerged as the bottleneck for the intended services. Contrary to the other types of data that become available either independently or automatically, this data requires the active human involvement. Unfortunately, the majority of the agricultural population is not familiar with such practices and does not devote the little time and effort needed to crowdsource this information.

Essentially, the platform embodies the essence of the crowdsourcing concept and faces typical challenges related to it; namely, it leverages small individual data contributions from multiple sources (multimodal data) to

generate collective intelligence that can benefit everyone (including those who do not contribute data). The challenge is, however, to motivate contributions from a critical mass that will render it meaningful.

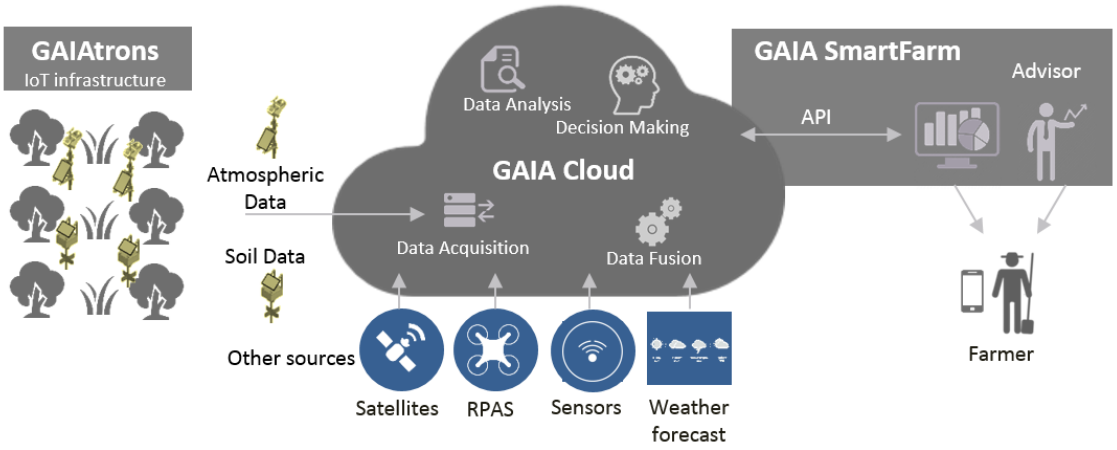


Figure 4.1: GAIA SENSE. GAIA Infrastructure for supporting smart farming services.

Discussions with GAIA during the symposium and in a follow-up meeting in February 2017 between the netCommons AUEB team and Neuropublic (software SME that serves as a major stakeholder of GAIA and as provider of software solutions to it) shaped the very promising idea of enhancing CommonTasker capabilities to link it with the GAIA SENSE platform. This would put the application at the service of the fundamental needs of the local economy and maximize its impact on the community. More specifically, the idea is to enrich the crowdsourcing component of the CommonTasker with a gamification component that will nudge farmers to contribute to the web app. As a second mode of coupling, the CommonTasker app could be used as a means to ease the provision of farmer inputs “on the go” through their mobile device.

4.2 CommonTasker design updates

The 1st release of CommonTasker implemented three main client capabilities: crowdsourcing of tasks, exchange of information, and sharing of equipment. These three options were presented to the user through the main menu in the UI as (”Errands”, ”Questions”, ”Sharing”), respectively. In the 2nd release of CommonTasker, we will combine functionality from the ”Errands” and ”Questions” modules to be able to crowdsource farming data from users and complement it with a gamification mechanism to incentivize the farmers’ inputs.

The functional model of the application is shown in Fig. 4.2. Four entities, all in distinct phsyical locations, are involved:

- the mobile client running on a smart mobile device (Mobile UI);
- the application backend, currently implemented over the Firebase web platform;
- the cloud infrastructure of GAIA storing the user data (GAIA Cloud); and
- the desktop PCs, through which farmers currently interact with the web app of GAIA and enter calendar-type data about their daily activities.

The CommonTasker functionality is split between its frontend (mobile UI) and its backend (currently drawing on Firebase) and includes:

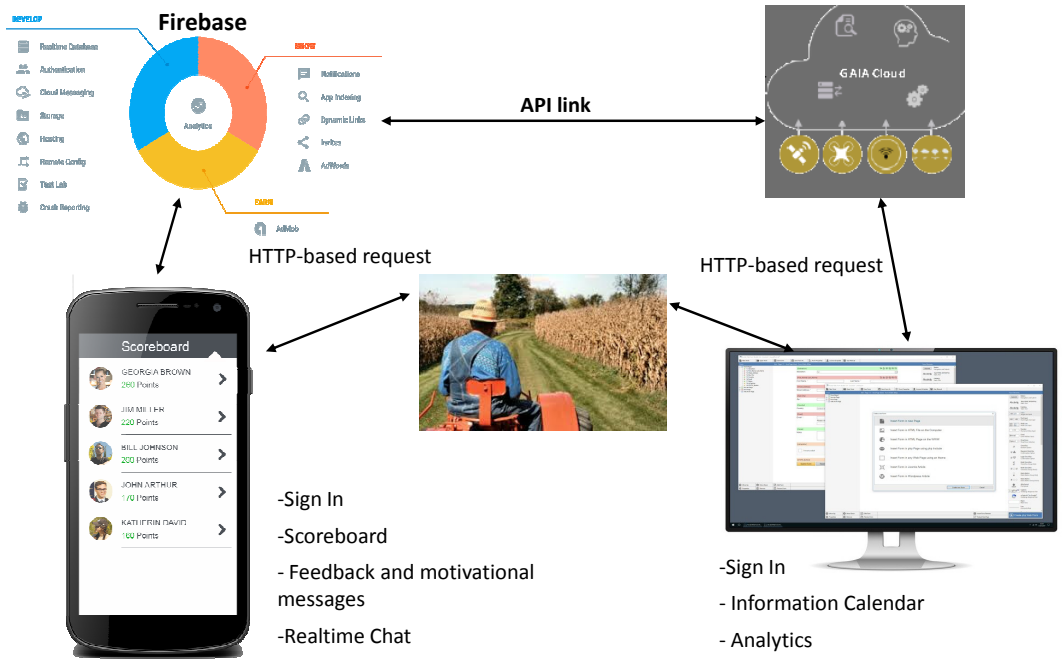


Figure 4.2: Communication between mobile app and web app.

4.2.1 Collecting farmers' inputs.

Farmers can use the mobile app to report the details of their daily work. Initially, the application will allow for text inputs and forward them to the GAIA Cloud. CommonTasker could also be used to collect other data types such as photos, videos, sound etc. However, in this 2nd release, text is the main envisaged data type. An example sequence diagram is shown in Figure 4.3.

4.2.2 Gamification modules.

To foster the farmers' willingness to keep reporting data for smart farming services, CommonTasker will implement gamification mechanisms. The idea is to keep a record on the user's past uploads or filling forms and create certain metrics for the comparison of users. Metrics that could be relevant to this end, include the number, frequency of even the whole time-series of upload events per user, quality indices assessing the relevance and value of data (how detailed are they, how clearly presented, how unique for a given area of coverage/interest).

These data will be made available to the application from the GAIA Cloud through an API that will be defined in the next phase of the task in coordination with GAIA. The data will be processed by the application backend to produce several kinds of statistical information pertaining to individual users or group of users, over different time intervals. These statistics will have a strong comparison (game) flavor, reflecting how well a user fares when compared to other app users. Depending on his/her performance, different types of motivating messages and motives will be presented to him/her.

Gamification utilizes basic elements taken out of game design and integrates them according to the particular context used and the desired gamified experience to be achieved. The design methodology used for the gamification mechanism of CommonTasker focuses on the MDA framework [10]. This framework is widely used for game design and includes the three basic principles of Mechanics, Dynamics and Aesthetics. Mechanics refers to the type of functional components selected for a game. Dynamics explores the user's interactions with the game and other users and Aesthetics values the overall user experience created by the game.

4.2.2.1 CommonTasker Mechanics

CommonTasker's initial design of the gamification mechanism includes a *point system* where users are differentiated utilizing different metric types according to their activity within the application, *scoreboards* that utilize the accumulated points to rank users and *challenges* in the form of small text messages and notifications letting informing the users about reaching potential goals in the scoreboards. All of the aforementioned mechanics can either be individually set or cooperatively. In more detail the functional components include:

- **Per user ranking** will aim to capture the individual interest or intrinsic motivation of a user to compete with other individuals. Messages informing the user for its current ranking or potential place given that he/she performs certain activities are due in this case. Individual information may be seen by other users.
- **Per village ranking** will draw on aggregate metrics of users' performance within each village. It aims to capture the competitiveness of group activities and local interest of users. Individual information is not shared with other users.
- **Per age group** ranking aims at distinguishing the users according to their specific age groups in an effort to identify the age groups that remain inactive and incentivize i.e. providing special messages for triggering their interest. Individual information is not shared with other users.

4.2.2.2 CommonTasker Dynamics:

Through the gamification mechanism the user, in response to the mechanics, creates the daily inputs required to receive points and competes with other users or user groups in the scoreboard. Moreover, the user is expected to exert extra effort when a challenge is available. This means providing the inputs required by the app either individually or cooperatively. In group efforts(per village or per age group ranking), users not only learn to compete with other but they also learn to cooperate with others. Reaching a common challenge can engage users in a more efficient and enjoyable way than targeting individual goals [11].

4.2.2.3 CommonTasker Aesthetics:

The aesthetics of the system focus on the feelings created to the users through their interactions with the mechanics and the dynamics. CommonTasker's gamification mechanism aims to provide a pleasant experience to the users through fun and satisfaction. Fun should be the result of individual and group competition while satisfaction can be produced in two ways. The first way is to capture the goals and challenges of the application. The second way through its basic utility and added value that brings to the users which provide inputs in order to receive agricultural advice from GAIA for their agricultural activities.

4.2.3 Additional CommonTasker functionality

The additional functionalities of CommonTasker include:

- The **API that needs to be defined between the GAIA Cloud and Firebase (mobile App backend)**. This will determine the format of data that will be passed to Firebase to implement the statistics required by the gamification modules.
- **the synchronization of a users' accounts in the GAIA web app and their profiles in CommonTasker**. This requires the flexibility to sign-in users (using multiple providers, like Facebook Login or Google Sign-in) and then have the authentication system recognize the user even if they tried signing in via a different method.When authenticating with an email/password, you can store in Shared Preferences an authentication token which can be used later to restore a session even after the app is killed. Upon creating a new user, the user can be registered and then logged in realtime database. Firebase will then retrieve farmer data from the GAIA Cloud through the API, match with auth token and join with Score database reference (ref. Fig.4.3) 4.4).

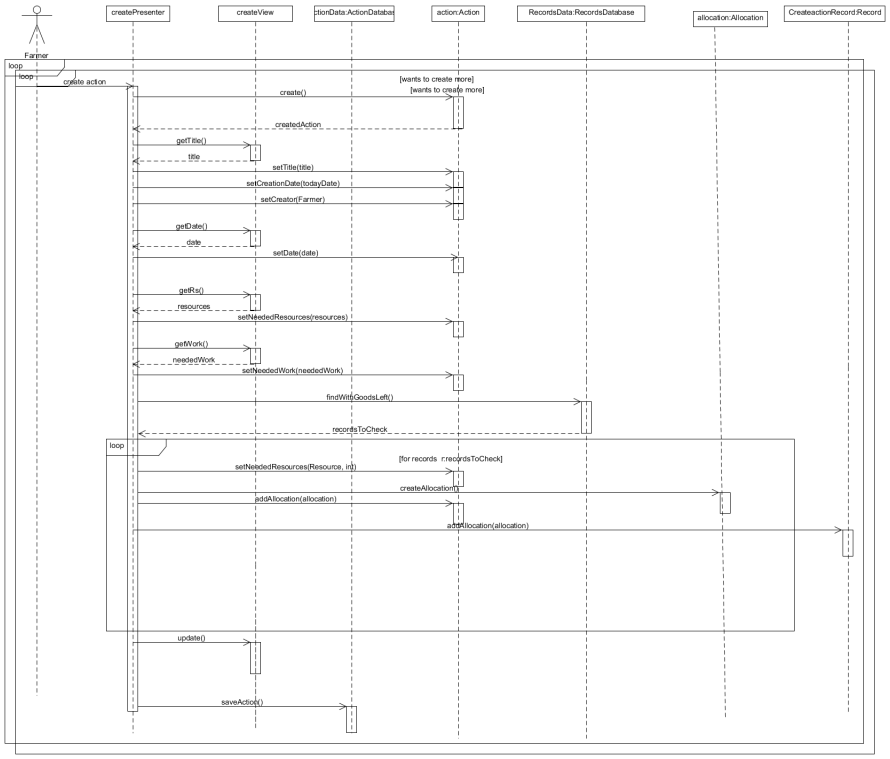


Figure 4.3: Sequence diagram to create an action of their daily work.

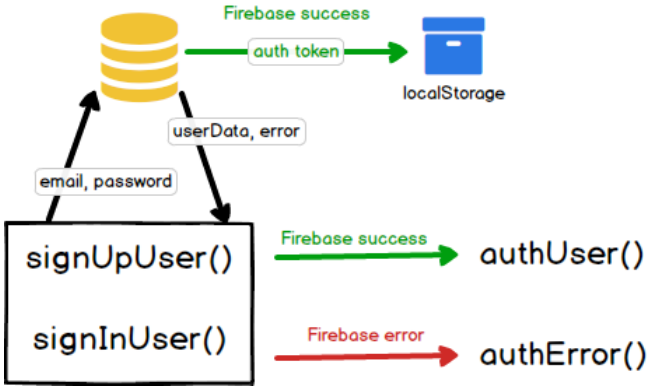


Figure 4.4: Authentication of a user during Sign In/Up processes.

All processes takes JSON format makes a easy way to replicate score ranking with Scoreboard.So you do validate encourage messages feedback system and users have the opportunity evidence time series diagram.

4.3 Software updates

The implementation of the application, has taken a shift in order to provide a qualitative and quantitative understanding of users. This is implemented via evaluating user performance with specific metrics. The new features of the application contain user profiles and scoreboards which are relatively straightforward to accomplish.

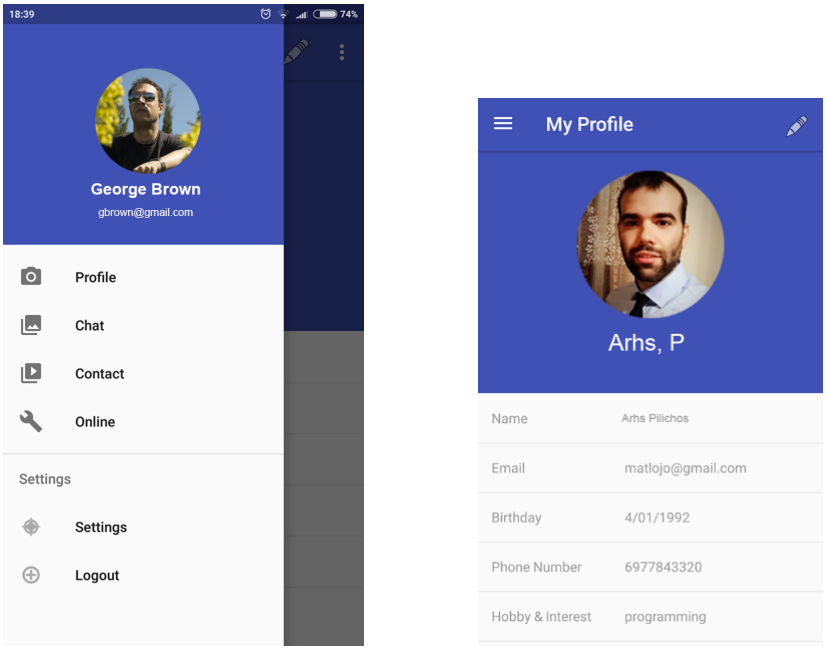


Figure 4.5: User profile pages and menus in the CommonTasker UI.

4.3.1 Client side

The updated client part of CommonTasker consists of the following entities.

- **User profiles:** Users are the main actors of the project. The application retains for each user, information such as their (nick)name, telephone number, age (including checking users for being over 18), their location (useful for per village ranking feature) and the ID used by the application as an identifier of the particular user. Each user has a user's profile layout with UI friendly design and encouragement messages.
- **Scoreboard:** The scoreboard (Fig. 4.6) aims to motivate user engagement and participation and also, drive their behavior towards achieving predefined goals. The scoreboard gives the user quick, anytime access to their personal credit scores and ranking reports, which are successively constructed according to the inputs given by each user. The scoreboard scale has three basics modules: user ranking, village ranking, age group ranking.

4.3.2 Backend server

The backend server is currently operated using Firebase (Fig. 4.7). Firebase is a mobile and web application development platform, made up of complementary features that developers can mix-and-match to fit their needs. CommonTasker utilizes Firebase services to store the received user inputs. Firebase allows for data storage in JSON format and immediate synchronization across all platforms and devices when changes are performed in the data. These desired features have enabled a flexible real-time version of CommonTasker. In fact, the communication with the application response data is delivered within seconds (within 200 ms after the arrival on the asynchronous database). It also, implements authentication using the Firebase Auth service.

After having added Firebase to the android project the integration of Firebase libraries includes the following steps:

1. Adding rules to the build.gradlefile, to include google-services plugin.
2. In app/build.gradle file add: applying plugin: 'com.google.gms.google-services'

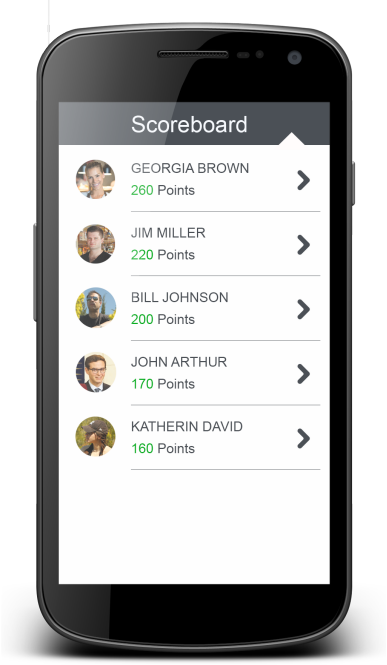


Figure 4.6: Scoreboard

3. Adding dependencies using packages com.google.firebase:firebase-auth:11.0.2 for Authentication, com.google.firebase:firebase-database:11.0.2 for Real-time Database and com.google.firebase:firebase-storage:11.0.2 for Storage.

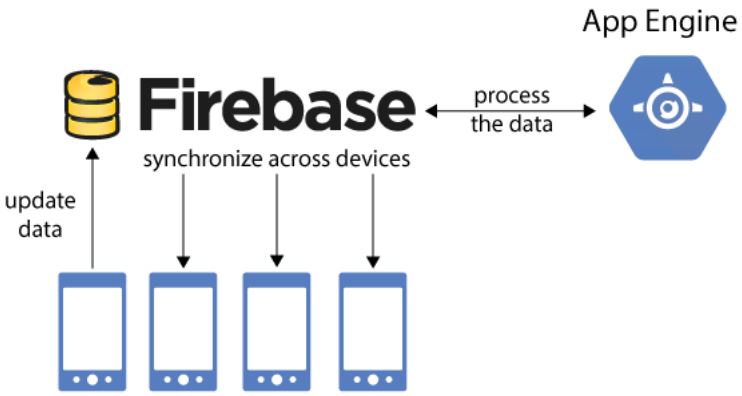


Figure 4.7: Firebase

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

4.3.2.1 Authentication

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

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

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

When users sign in to the app, their sign-in credentials (for example, their username and password) are sent to the authentication server. The server checks the credentials and returns a custom token if they are valid. After a user signs in for the first time, a new user account is created and linked to the credentials—that is, the user name and password, phone number, or auth provider information—the user signed in with. This new account is stored as part of the Firebase project, and can be used to identify a user across every app in your project, regardless of how the user signs in. After having added Firebase and Authentication dependencies to CommonTasker, we can let your user create login id by the following:

- First, add FirebaseAuth object in your onCreate method

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

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

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

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

- You can setup an AuthStateListener to check login status

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

Figure 4.8: User Authentication and creation of login ID.

4.3.2.2 Real-time Database

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

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

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

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

Data can be fetched by adding a listener named ValueEventListener to DatabaseReference. You can fetch data by adding a listener named ValueEventListener to DatabaseReference by following the codes in Figure 4.9 and 4.10.

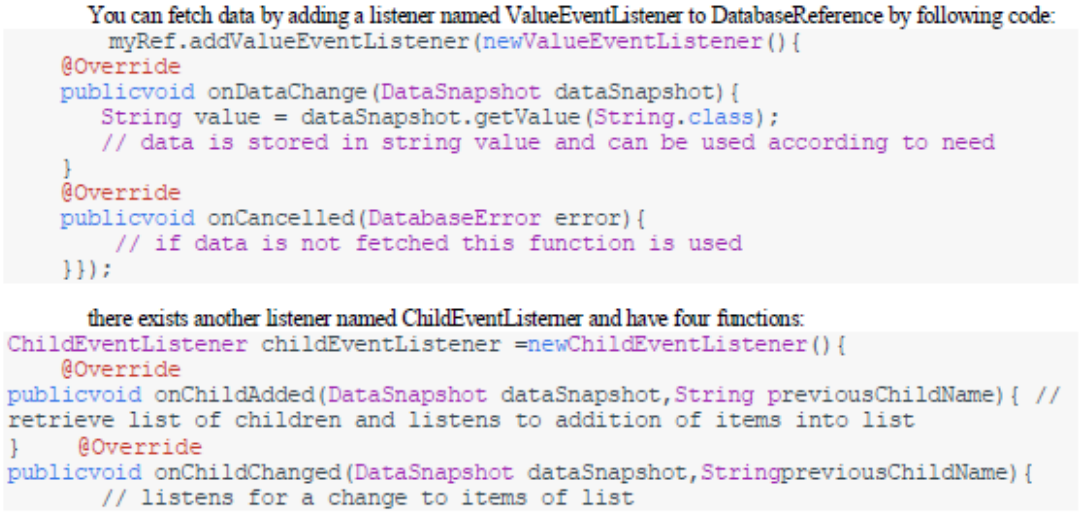


Figure 4.9: Real-time Database - Listeners code.

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

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

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

Figure 4.10: Real-time Database - Child Event code.

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

4.3.2.3 Storage

Firebase Storage provides secure file uploads and downloads for Firebase apps, regardless of network quality. The storage feature enables storing files like images, audio, video etc. Data stored in a highly secured and robust environment and is able to resume from a previous updated point in case a network error occurs. Storage feature is for those who want their apps to store files like images, audio, video etc. Data stored is highly secured and robust means they resume from last point if network error occur. The steps below are followed to use the storage feature in the android application:

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

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

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

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

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

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

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

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

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

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

Figure 4.11: Storage Upload item code.

Bibliography

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

[2] N. Apolónia, F. Freitag, and L. Navarro, "Gossip-based service monitoring platform for wireless edge cloud computing," in *IEEE International Conference on Networking, Sensing and Control (ICNSC)*, 2017.

[3] E. Dimogerontakis, J. Neto, R. Meseguer, L. Navarro, and L. Veiga, "Client-side routing-agnostic gateway selection for heterogeneous wireless mesh networks," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Lisbon, Portugal, 05/2017 2017. http://people.ac.upc.edu/leandro/pubs/proxy-allocation-im2017.pdf

[4] L. Abeni, C. Kiraly, A. Russo, M. Biazzini, and R. Lo Cigno, "Design and Implementation of a Generic Library for P2P Streaming," in *Proceedings of the 2010 ACM Workshop on Advanced Video Streaming Techniques for Peer-to-peer Networks and Social Networking*, ser. AVSTP2P '10. New York, NY, USA: ACM, 2010, pp. 43–48. http://doi.acm.org/10.1145/1877891.1877902

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

[6] L. Maccari, N. Facchi, L. Baldesi, and R. Lo Cigno, "Optimized P2P Streaming for Wireless distributed Networks," *Accepted for Publication in the Elsevier Journal of Pervasive and Mobile Computing*, 2017.

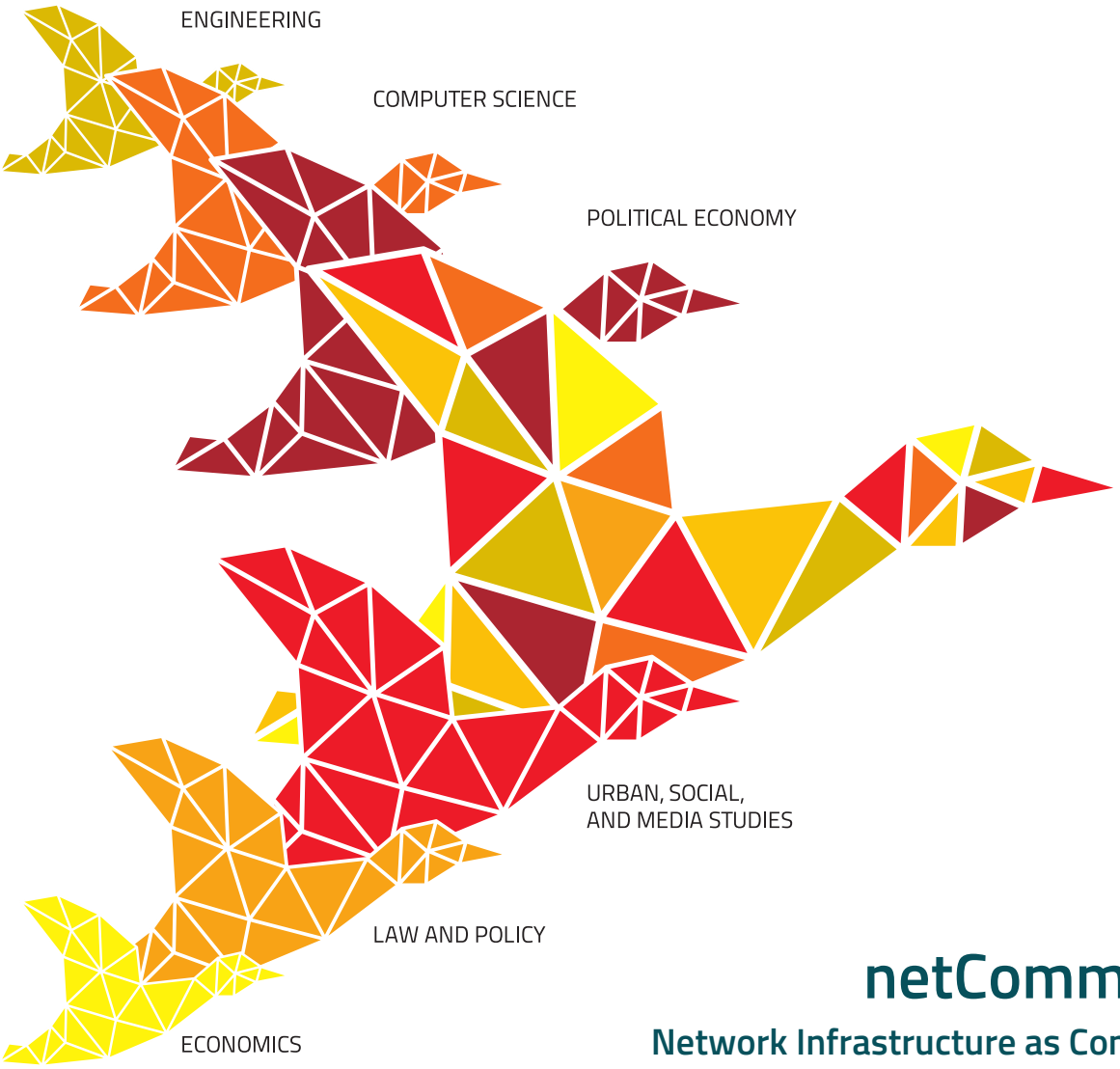
[7] L. Baldesi, L. Maccari, and R. Lo Cigno, "On the Use of Eigenvector Centrality for Cooperative Streaming," *Accepted for publication in IEEE Communications Letters*, 2017.

[8] ——, "Optimized Cooperative Streaming in Wireless Mesh Networks," in *In Proc. of The 15th IFIP Networking Conference (NETWORKING)*, May 2016.

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

[10] R. Hunicke, M. LeBlanc, and R. Zubek, "Mda: A formal approach to game design and game research," in *Proceedings of the AAAI Workshop on Challenges in Game AI*, vol. 4, no. 1, 2004, p. 1722.

[11] S. Lounis, K. Pramatari, and A. Theotokis, "Gamification is all about fun: The role of incentive type and community collaboration," 2014.



netCommons

Network Infrastructure as Commons

Release of New Open Source Software for All Applications

Deliverable Number D3.4 – Interim release
Version 0.5
August 9, 2017

netCommons.eu