System requirements and software architecture

Deliverable D2.1

Date: 30 June 2013

Version: 1.0
This page is left in blank intentionally.
Abstract

This deliverable presents the system requirements for the CLOMMUNITY Cloud software system, an evaluation of existing and proposed services for inclusion in the system, and an initial high-level architecture for the system. The document describes the requirements, as of M06, for the components of the CLOMMUNITY platform: the hardware, the Cloud management platform, the network services, and the application services for the platform. We evaluate existing systems as candidate solutions for each of these different layers, indicating their suitability or unsuitability for inclusion in the CLOMMUNITY platform. We conclude with our proposed architecture containing the systems and services we expect to be included in the CLOMMUNITY Cloud software system.
Executive Summary

This document presents the initial system requirements and software architecture for CLOMMUNITY defined during the first 6 months of the project. In order to design and develop new software and services for a community Cloud, we analysed and studied the actual status of the available Cloud systems, community networks, and software systems.

In the first place, the need for Cloud management software is stated, taking into account the status of existing Cloud management platforms, with special attention to the prototypes developed in the CONFINE project, on whose testbed the community Cloud will be deployed. Based upon these requirements and our experience with community networks and their existing software and services, we present a proposed set of network services. Moreover, we explain the need for hardware and software on the nodes to support the development and deployment of a complete community Cloud system. We also propose hardware for nodes to be used by community network users as part of the community Cloud infrastructure.

In the development of the requirements, research on new hardware, software, and services was needed. We identified and evaluated many existing software systems for their potential inclusion in the community Cloud platform. In particular, we tried to assess their fit to the needs of the project, and their suitability to be components of community Clouds. The work done related to this evaluation is a large assessment of many different systems, from networking to application layers, with the intention of integrating them into the community Cloud software system.

Bringing together the experience in the community networks, the existing software, the existing services and the development of new services led to the design of an initial software architecture to support Cloud computing for community networks. We present this proposed architecture, and then conclude the document.
# TABLE OF CONTENTS

1. Introduction ........................................................................................................................................... 9
   1.1. CONTENTS OF THE DELIVERABLE......................................................................................... 9
   1.2. RELATIONSHIP TO OTHER COMMUNITY DELIVERABLES.................................................. 9

2. Overall Requirements Of The Community Cloud Software System................................................. 10
   2.1. CLOUD MANAGEMENT PLATFORM ....................................................................................... 10
       2.1.1. CONFINE.......................................................................................................................... 10
       2.1.2. RELATION TO OPENNEBULA, OPENSTACK AND CLOUDSTACK.............................. 11
   2.2. HARDWARE TO SUPPORT THE COMMUNITY CLOUD HOSTS........................................... 12
       2.2.1. CONFINE NODE DEPLOYMENT (TESTBED)................................................................. 12
   2.3. NETWORK SERVICES...................................................................................................................... 12
       2.3.1. LOCATION OF SERVICES............................................................................................... 13
       2.3.2. AUTO-MANAGED DOMAIN NAME SERVICE............................................................... 13
       2.3.3. CLOUD FEDERATION....................................................................................................... 13
       2.3.4. AUTHENTICATION AND FEDERATION......................................................................... 13
       2.3.5. DISTRIBUTED STORAGE................................................................................................. 14
   2.3.6. NAT TRAVERSAL.................................................................................................................... 14
   2.3.7. ADAPTIVE HTTP PROXY..................................................................................................... 14

3. Evaluation Of Existing Software............................................................................................................. 15
   3.1. CLOUD MANAGEMENT PLATFORMS......................................................................................... 15
       3.1.1. OPENSTACK...................................................................................................................... 16
       3.1.2. CLOUDSTACK.................................................................................................................. 20
       3.1.3. OPENNEBULA.................................................................................................................. 21
   3.1.4. EVALUATION OF DIFFERENT CLOUD PLATFORMS.......................................................... 32
   3.2. COMMUNITY SOFTWARE DISTRIBUTION.................................................................................. 33
   3.3. SERVICE DISCOVERY - AVAHI................................................................................................. 35
       3.3.1. ONE LAPTOP PER CHILD (OLPC).................................................................................. 38
       3.3.2. COMMUNITY NETWORK LOGICAL TOPOLOGY............................................................ 38
       3.3.3. SERVICES AND COMMUNITY....................................................................................... 39
   3.4. DISTRIBUTED FILESYSTEM - TAHOE-LAFS............................................................................ 43
       3.4.1. GENERAL USAGE.............................................................................................................. 45
       3.4.2. TAHOE-LAFS PROJECT AND COMMUNITY..................................................................... 46
   3.5. DISTRIBUTED STORAGE - CATS................................................................................................. 47
   3.6. AUTONOMIC ELASTICITY MANAGER FOR CLOUD-BASED SERVICES - ELASTMAN............ 49
   3.7. P2P MEDIA STREAMING - PEERSTREAMER.......................................................................... 50
   3.8. P2P VIDEO-ON-DEMAND - GVOD.......................................................................................... 52
       3.8.1. TREE-BASED P2P VoD ..................................................................................................... 53
       3.8.2. MESH-BASED P2P VoD .................................................................................................... 53
   3.9. SEARCH SERVICE......................................................................................................................... 55
       3.9.1. INDEX ENTRIES................................................................................................................ 55
       3.9.2. SEARCH SERVICE ARCHITECTURE................................................................................ 56
   3.10. NAT TYPE IDENTIFICATION AND NAT TRAVERSAL........................................................... 57
D2.1 System requirements and software architecture

FIGURES

Figure 1: Community Box........................................................................................................................................12
Figure 2: OpenStack Components....................................................................................................................18
Figure 3: OpenStack And End Users................................................................................................................18
Figure 4: OpenStack Nested Organization Of A Zone........................................................................................19
Figure 5: OpenNebula Architecture For Testing.................................................................................................23
Figure 6: ONE: 'onehost List' Command.............................................................................................................24
Figure 7: ONE: 'onevm List' Command.................................................................................................................24
Figure 8: ONE: 'onevm Show 7' Command............................................................................................................25
Figure 9: ONE: LXC Create Hosts With LXC Drivers............................................................................................26
Figure 10: ONE: Add The Image To The Datastore............................................................................................27
Figure 11: ONE: Create A Template....................................................................................................................27
Figure 12: ONE: Create A Template #2.................................................................................................................28
Figure 13: OpenNebula CONFINE Architecture..................................................................................................28
Figure 14: Community Cloud Devices................................................................................................................39
Figure 15: Community Ecosystem With Connections And Devices.........................................................................42
Figure 16: Tahoe-LAFS Architecture....................................................................................................................44
Figure 17: Tahoe-LAFS: Community Proposal.....................................................................................................47
Figure 18: ElastMan Feedforward Control............................................................................................................49
Figure 19: ElastMan Feedback Control................................................................................................................49
Figure 20: P2P Video-on-Demand Application....................................................................................................54
Figure 21: CLOMMUNITY-Supported Search Architecture................................................................................57
Figure 22: Decentralized Search Architecture.....................................................................................................57
Figure 23: CLOMMUNITY NAT Model................................................................................................................58
Figure 24: A Sample Add-on Icon On Firefox......................................................................................................63
Figure 25: Communication Between Add-on Scripts And Content Scripts..........................................................63
Figure 26: Distributed Software Architecture.....................................................................................................67
D2.1 System requirements and software architecture

TABLES

Table 1: OpenStack Core Services Folsom ................................................................. 16
Table 2: Similarity Of OpenStack Services With Services Of Other Cloud Platforms .......... 17
Table 3: OpenStack Hypervisors Support ................................................................. 17
Table 4: Features In OpenNebula 4.0 Release ......................................................... 22
Table 5: Current Status Of The Testing/development ............................................. 23
Table 6: Evaluation Of Different Cloud Platforms ............................................... 33
Table 7: Distributed Software Architecture ......................................................... 67
1. Introduction

This section presents a summary of the (i) contents of the deliverable, and (ii) relationship to other CLOMMUNITY deliverables.

1.1. CONTENTS OF THE DELIVERABLE

This document presents the services needed at different levels for a community Cloud and relates these services with the provision of applications usable by as many end users as possible. The research proposes a suite of software and hardware for the nodes to support the community Cloud. We propose specific hardware to be used for community Cloud nodes on which we have started to test our system and which is also envisioned for the hosts of the users at home. The community Cloud provides community network end users with services and applications and facilitates access to Cloud infrastructure. The community Cloud will run within the Guifi.net community network and use some services already provided in the network. Based on the research and development already done in the project on the software system, an initial software architecture is defined.

1.2. RELATIONSHIP TO OTHER CLOMMUNITY DELIVERABLES

The D2.1 provides input to WP4 and it is related in part with D3.1, and also presented at M06. With regards to the DoW, the relation of D2.1’s content to other future deliverables is as follows: “The ultimate result of this task will be the definition of a software architecture for the integrated community Cloud software system. That architecture will be first reported in D2.2 (M12, milestone m2.3.1) and finally reported in D2.3 (M24, milestone m2.3.2)”. The defined architecture from T2.1 will be reported in the deliverables D2.2 and D2.3. The initial software architecture is defined in D2.1.
2. Overall requirements of the community Cloud software system

The content of this section is published in part on the wiki of the project [1]. The CLOMMUNITY wiki is a tool to support the dissemination of intermediate progress and results of our work within the community. In this section, we present the overall requirements of the community Cloud software system that are based on the analysis and evaluation of the existing software and services, desired software, research done, future research and prototypes.

In order to enable and provide a community Cloud, i.e., Infrastructure as a Service (IaaS) and Software as a Services (SaaS), in a community network, CLOMMUNITY needs to include a Cloud management platform to manage and provision Cloud resources to applications and services running in a community network; and a number of network-aware services (outlined in the next sections), including service location and discovery, storage, authentication, Cloud federation, auto-managed DNS, adaptive HTTP proxy and NAT traversal, that are needed to build community Clouds, Cloud federations, as well as higher level Cloud-based services and applications in a community network.

General requirements for Cloud resources management and network services defined and discussed in this section include typical software-engineering requirements, such as acceptable availability and performance, fault-tolerance, as well as requirements typical to Peer-to-Peer (P2P) systems that operate under churn (node joins, leaves, and failures).

2.1. CLOUD MANAGEMENT PLATFORM

We consider a Cloud management platform in terms of a layer over the network and services. The Cloud management platform provides an interface for the underlying network that allows the deployment of other services over the abstraction layer. There are a number of open source Cloud management platforms, such as OpenStack and OpenNebula that define the state of the art in the field of Cloud platforms. There are a number of research projects aimed at development of Cloud management systems, such as CONFINE and CONTRAIL. In this document, we provide an overview of some of the existing Cloud management platforms and analyse whether any of them, namely CONFINE, OpenStack, OpenNebula and CloudStack, can be adapted for CLOMMUNITY.

2.1.1. CONFINE

The CONFINE project [38] provides a testbed for network experimentation in community networks. The testbed consists of nodes with CONFINE-specific software and a web-based user and management interface called Controller to manage the resources. The document [2] defines some terminology adopted from PlanetLab [3] with regards to slices and slivers used in the project as: “A slice is an orthogonal fraction of the testbed, or in other words, a set of nodes on which
applications are assigned a fraction of resources by means of a kind of virtual machine called sliver”. We can say that a slice groups slivers; it is a set of nodes. A sliver is a resource fraction. In the referred document a sliver is said to be a kind of virtual machine.

The nodes of the CONFINE project can be managed using a web-based interface called Controller to manage the software and resources of the nodes. The web-based interface allows the configuration of the network interfaces of the node and of the software installed on the nodes. The web-interface supports mainly the management of nodes, slices and slivers as follows.

- For the node management, the Controller allows initial installation of the base software in a node to allow the installation of another layer of software to deploy the experiments.
- For the management of slices, the Controller allows the creation, management, control and debug the status and configuration of slices.
- For the management of Slivers, the Controller allows the creation, management, control and debug the status and configuration of slivers.

The web-based interface also allows the configuration of:

- The underlying Tinc network between nodes that allows to form a Layer 2 network between them.
- The keys used in the authentication of some parts of the system.

The nodes of the testbed can principally be any kind of hardware or software to allow the installation and deployment of the node base software inside them. In practical terms, however, the hardware is restricted by the particular conditions of community networks.

The base layer of the CONFINE project software is installed in the nodes. Inside the installed base operating system layer another software is installed. Its configuration can vary from one experiment to another and it relies on the underlying base to communicate and cooperate with other nodes of the testbed. This software layer refers to the slivers. The slivers are logically grouped in slices definitions. A slice defines and contains one or more slivers.

The researcher in CONFINE see the slivers as usual (traditional) operative systems with its usual network interfaces, memory and hard disk space. They do not see the underlying configuration or complexity, which is managed using the Controller web-based interface.

2.1.2. Relation to OpenNebula, OpenStack and CloudStack

CONFINE is the base for our experiments and deployment, since CLOMMUNITY uses the testbed provided by CONFINE. Apart from CONFINE, we present in other parts of this document the OpenNebula, OpenStack and CloudStack solutions for managing Cloud resources.
2.2. HARDWARE TO SUPPORT THE COMMUNITY CLOUD HOSTS

We use hardware from the CONFINE project to support the research with initial tests and deployment.

2.2.1. CONFINE NODE DEPLOYMENT (TESTBED)

The testbed used in CLOMMUNITY initially is within the same testbed offered by the CONFINE project. CONFINE's research devices, used in CLOMMUNITY as “community boxes”, will host services and applications of the community Cloud. Currently, as part of the CONFINE testbed, they are managed by the CONFINE Controller.

The community box is a hardware and software system dedicated to contain part of the Cloud software. We use CONFINE community networks friendly hardware for the Cloud hosts. It allows the installation of a basic operating system and the creation of virtual machines. The CONFINE testbed does not use virtualization in the classic conception, but it uses a lighter concept related to virtualization, i.e., Linux Containers (LXC). The reason is that for using LXC we do not need a high performance machine to host classic virtualized environments.

At the moment, as community boxes we use the CONFINE hardware, e.g., an amd64 or x86 low-end processor, 2 or 4 GB RAM, one network card, and SSD disks (Figure 1). These devices are fanless, thus they are silent. They are also widely compatible, low-cost and have low power consumption.

![Community Box](image)

Figure 1: Community Box

2.3. NETWORK SERVICES

The project aims to provide network-aware services as stated in the DoW. In practical terms and applied to Guifi.net, the network awareness is supported by an application currently under development that helps nodes to choose the appropriate network configuration to use. With that, the nodes connect and use services using the network-awareness concept. In this section, we expose the needed services. Community networks are already using some of the services, which are going to be part of CLOMMUNITY. Here, we summarise the network-aware services to be provided in CLOMMUNITY, and present some of them in more detail below.
2.3.1. Location of services

In the CLOMMUNITY software, we include a service, called Avahi, to locate services inside the community network. Avahi is an implementation of the Zeroconf specification. It is used to publish and locate network services. We studied it and its functionality to locate and publish services in the project software.

2.3.2. Auto-managed Domain Name Service

Inside community networks, users manage their own Domain Name Service (DNS) for their software and services, and they make the service publicly available to the community network. For example, the network user who has a node, can install and configure a local DNS server with public access. This allows the user services to be present inside the community network and to be reachable by others to query the user’s DNS service.

2.3.3. Cloud Federation

In the context of community Cloud computing, many factors lead to the consideration of federations of micro Clouds as the main organization of the overall system. For example: (i) existing of micro Clouds inside different administrative boundaries, (ii) diversity of resources, and (iii) geographically distribution of micro Clouds and better usage of the locality. The first requirement for cross-Cloud federation is to find some protocol in order to share resources. Clouds inside the same administrative zone of the community may share their resources as a whole, while Clouds of different zones may use other approached, e.g., a market-based model, to share their resources. Moreover, in order to enable inter-operation, a unified interface must be used to abstract the usage of the different underlying Cloud management middleware. Cross-Cloud identities and authentication is also another issue that should be taken into account in order to have a way to securely exchange resources between Clouds and verify the identity of the parties.

2.3.4. Authentication and Federation

Inside the community network, to access some services the user needs to authenticate. In particular in Guifi.net, the LDAP authentication standard is used to authenticate the users in the community network and in the related global resources associated with the community network.

The user can be authenticated on different services and different hosts, sharing the same authentication data between them. Some services of community networks are federated using LDAP. Different groups have developed different access models for community network resources. Guifi.net has a federated LDAP authentication service. The following are some examples of federated LDAP authentication usages:

• Validating users inside Guifi.net against the proxy service to get Internet access.
• Authenticating users in Guifi.net web site.
• Authenticating users to access the Guifi.net Wiki.

2.3.5. DISTRIBUTED STORAGE
We need a distributed storage system for the project. We expose some approaches and show the relations with the CLOMMUNITY software and the software distribution being used. See Section 3.5 for more details.

2.3.6. NAT TRAVERSAL
Many existing distributed algorithms for constructing overlay networks are not practical over the community networks like Guifi.net because of the presence of Network Address Translation (NAT) devices and firewalls. Hence, we need NAT traversal protocols to communicate with the nodes reside behind NAT. We present the details of the NAT traversal techniques in Section 3.9.

2.3.7. ADAPTIVE HTTP PROXY
The community network Guifi.net provides a large set of proxies for accessing the Internet. These proxies are provided by community users on a voluntary basis and are advertised on the community web site. A browser plugin exists that helps the users to select a proxy that is later used by the client browsers. Details of this plugin is covered in Section 3.10.
3. Evaluation of existing software

In this chapter, we describe the results of our work on identifying and evaluating existing software services for inclusion in the community Cloud system. We have identified and evaluated (i) open-source third-party software that can be used in CLOMMUNITY, and (ii) the software earlier developed and contributed to the project by the CLOMMUNITY partners that can be adapted for deployment and provided in a community Cloud. The community Cloud is expected to provide an Infrastructure as a Service (IaaS), network services to support the platform, as well as Cloud-assisted applications for end users. The software presented and discussed in this section include:

- Open-source Cloud platforms, namely, OpenStack, CloudStack and OpenNebula;
- Guifi-Community-Distro, the software distribution with Cloud services for the Guifi.net community network;
- The services that support network-awareness, namely,
  - The Ahavi application that allows programs to publish and discover services and hosts running on a local network;
  - A distributed search service;
  - The Guifi.net Firefox plugin that provides a locality-awareness service for Guifi.net users to choose proxy;
  - NAT type identification and NAT traversal;
- Storage services, namely,
  - The Tahoe-LAFS distributed file system;
  - The CATS distributed key-value storage;
  - The ElastMan, autonomic Elasticity Manager for Cloud-based key-value stores;
- Content distribution services, namely,
  - The PeerStreamer, an open source peer-to-peer (P2P) live streaming application;
  - The Gvod, an open source P2P video-on-demand service;

3.1. CLOUD MANAGEMENT PLATFORMS

This section presents our review of three state-of-the-art open-source Cloud platforms, namely OpenStack, CloudStack and OpenNebula that we have considered as candidates to be adapted in CLOMMUNITY for some of the Cloud infrastructure services, such as Cloud management, compute and storage services.
3.1.1. **OpenStack**

OpenStack is a global collaboration of developers and Cloud computing technologists producing the open standard cloud operating system for both public and private clouds. OpenStack is provided under the Apache 2.0 license. OpenStack identifies services with code-names, as for example the Network service has the “Quantum” code-name. Table 1 represents the core services of OpenStack (the Folsom release), and Table 2 shows the similarity of OpenStack services with services of other Cloud platforms.

<table>
<thead>
<tr>
<th>Service (code-name)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dashboard (Horizon)</td>
<td>- Provides a modular web-based user interface for all the OpenStack services.</td>
</tr>
<tr>
<td>Identity (Keystone)</td>
<td>- Provides authentication and authorization for all the OpenStack services.</td>
</tr>
<tr>
<td></td>
<td>- Provides a service catalogue of services within a particular OpenStack cloud.</td>
</tr>
<tr>
<td>Compute (Nova)</td>
<td>- Provision and manage large networks of VMs;</td>
</tr>
<tr>
<td></td>
<td>- Running instances, managing networks, and controlling access;</td>
</tr>
<tr>
<td></td>
<td>- Hardware and hypervisor agnostic.</td>
</tr>
<tr>
<td>Image Service (Glance)</td>
<td>- Provides discovery, registration, and delivery services for virtual disk images.</td>
</tr>
<tr>
<td></td>
<td>- Images stored in a variety of back-end stores, including OpenStack Object Storage.</td>
</tr>
<tr>
<td>Object Storage (Swift)</td>
<td>- Creating redundant, scalable data storage using clusters of standard servers.</td>
</tr>
<tr>
<td></td>
<td>- Store multiple petabytes of accessible data.</td>
</tr>
<tr>
<td></td>
<td>- Long-term storage system for large amounts of static data that can be retrieved, leveraged, or updated.</td>
</tr>
<tr>
<td></td>
<td>- Distributed architecture with no central point of failure.</td>
</tr>
<tr>
<td>Block Storage (Cinder)</td>
<td>- Provides persistent block storage to guest VMs.</td>
</tr>
<tr>
<td>Network (Quantum)</td>
<td>- Provides &quot;network connectivity as a service&quot; between interface devices managed by other OpenStack services (most likely Nova).</td>
</tr>
<tr>
<td></td>
<td>- The service works by allowing users to create their own networks and then attach interfaces to them.</td>
</tr>
<tr>
<td></td>
<td>- Quantum has a pluggable architecture to support many popular networking vendors and technologies.</td>
</tr>
</tbody>
</table>

OpenStack aims at supporting all available hypervisors. Table 3 shows currently supported hypervisors. OpenStack consists of loosely coupled components, e.g., scheduler, network controller, compute worker, image store, that can be distributed (Figures 2 and 3). Components communicate with each other through a shared queue, e.g., AMQP/RabbitMQ, and a database. Communications are asynchronous.
Table 2: Similarity of OpenStack services with services of other Cloud platforms

<table>
<thead>
<tr>
<th>OpenStack Service</th>
<th>CloudStack</th>
<th>Amazon Web Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dashboard (Horizon)</td>
<td>Management Server</td>
<td></td>
</tr>
<tr>
<td>Identity (Keystone)</td>
<td>Infrastructure</td>
<td></td>
</tr>
<tr>
<td>Compute (Nova)</td>
<td>Infrastructure</td>
<td>EC2: Elastic Compute Cloud</td>
</tr>
<tr>
<td>Image Service (Glance)</td>
<td>Infrastructure</td>
<td>AMI: Amazon Machine Images</td>
</tr>
<tr>
<td>Object Storage (Swift)</td>
<td>Infrastructure</td>
<td>S3: Simple Storage Service</td>
</tr>
<tr>
<td>Block Storage (Cinder)</td>
<td>Infrastructure</td>
<td>EBS: Elastic Block Store</td>
</tr>
<tr>
<td>Network (Quantum)</td>
<td>Infrastructure</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: OpenStack Hypervisors support

<table>
<thead>
<tr>
<th>Hypervisor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XenServer/XCP</td>
<td>Commercial XenServer and open-source Xen Cloud Platform (XCP)</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine. A full virtualization solution for Linux on x86 hardware containing virtualization extensions.</td>
</tr>
<tr>
<td>QEMU</td>
<td>A generic and open source machine emulator and virtualizer. For hardware without virtualization extensions.</td>
</tr>
<tr>
<td>LXC</td>
<td>Linux Containers, a lightweight virtual system mechanism.</td>
</tr>
<tr>
<td>ESXi/VC</td>
<td>From VMWare.</td>
</tr>
<tr>
<td>Hyper-V</td>
<td>From Microsoft.</td>
</tr>
<tr>
<td>Bare Metal</td>
<td>Use hardware directly without virtualization.</td>
</tr>
</tbody>
</table>

As Figure 4 shows, the infrastructure is envisioned as having the following hierarchical disposition:

- Zone (The biggest system vision, ex data center level)
  - Secondary Storage (OpenStack Swift available to the entire zone)
  - Pod (Collection of clusters)
    - Cluster (Collection of workstations)
      - Workstations (With hardware virtualization support)
      - Primary storage (iSCSI or NFS)
D2.1 System requirements and software architecture

Figure 2: OpenStack components

Figure 3: OpenStack and End Users
We expect OpenStack to run in community networks with limited speed and bandwidth. Possible limitations include:

- OpenStack uses heartbeats to check that components are alive. This needs to be increased to match community network properties.
- Timeouts (e.g., when creating VMs) need to be increased as well.
- Bandwidth intensive tasks such as copying an image (~250MB) or VM migration will take very long time compared with a datacenter with gigabit networking.

We have evaluated OpenStack with the following components: Nova, Glance, Keystone, Horizon, and Swift. We have evaluated OpenStack with both KVM running on hardware with virtualized support, and Linux Containers (LXC) running on hardware without virtualization support (in a virtual machine). LXC does not support advanced features such as: resize, pause, suspend, attach volumes, live migration, snapshots, etc [9].

- OpenStack with KVM: OpenStack (Folsom release) with KVM is deployed and used for research in two clusters at KTH and SICS:
  - KTH cluster (4 HP ProLiant DL380 servers). Each with 2 x Intel Xeon X5660 (24 cores in total) and 44 GB of RAM & 2 TB Storage.
  - SICS cluster (11 Dell PowerEdge servers). Each with 2 x Intel Xeon X5660 (24 cores in total) and 40 GB of RAM & 2 TB of Storage.
D2.1 System requirements and software architecture

• Our evaluation of OpenStack (Folsom release) with KVM has shown that it is stable and easy to use for end-users, such as developers and providers of Cloud-based services.

• OpenStack with LXC: OpenStack (Grizzly release) with LXC deployed in a virtual machine with 1 vcpu and 2GB memory (to simulate weak hardware without virtualization support). The limitations of this evaluation include:
  ◦ Image should be in rootfs raw format.
  ◦ No support yet for VNC console to the virtual machine.
  ◦ Not every cloud image works out of the box, because of cloud-init scripts that try to initialize elements not present in the container.
  ◦ The kernel of the rootfs should be the same version of the host running kernel.

3.1.2. CloudStack

Apache CloudStack is open source software designed to deploy and manage large networks of virtual machines, as a highly available, highly scalable Infrastructure as a Service (IaaS) Cloud computing platform. CloudStack is a virtual infrastructure management software created by Citrix, and it was later donated to Apache Software Foundation (ASF). Until very recently the project was still in incubation phase. This is the phase in which the ASF considers a project as potentially useful but still not mature. During this phase the project has assimilated other ASF related projects and grown rapidly, but the exit from the incubation phase is to be considered still as only administrative. In any case it does not reflect or reassure about the development life cycle of the project.

CloudStack is designed around two main components: (i) the Management Server (MS), and (ii) Infrastructure Support (IS). The MS is the single point of configuration of the entire Cloud infrastructure, all the information regarding configurations of the Cloud is managed by the MS. This component also manages user credentials and template availability. In order to scale the infrastructure the MS can be configured to have multiple synchronized load/balanced instances and the database (MySQL) can be configured to run in replication mode.

The runtime needed for the MS to actually match configuration of the machines to the server is the IS. The infrastructure supports only the following hardware hypervisors:

• Citrix XenServer
• VMWare vSphere
• KVM (Ubuntu/Centos)
• Xen (Ubuntu/Centos)
•

The storage model of the Cloud infrastructure needs to have dedicated hardware support in order for storage to be accessible at top performance and high availability. In the case of the secondary storage (the storage used by the manager to manage templates and images) the module serving such storage is not developed by the maintainers of the project. The documentation of the project
advices the use of Swift, the object store from another VIM project OpenStack, as a secondary storage.

The hard-wired network is not modifiable even if routers can be reconfigured by OpenFlow, the user has to map the hardware network to the configurations of such network on the MS and the administrators or users of the infrastructure can modify virtual networks between the guest VMs. The network model is bound to the physical network and each hierarchy element can be configured to have many different subnets. The virtual network between the VMs can be configured based on VTAGS or service level agreements. Users and their permissions are managed by centralized management server. CloudStack supports authentication via LDAP and also creation of different project and mappings from users to projects.

Technology used by CloudStack has a big potential for management and cross-Cloud federation extensions, but the centralized nature of management is a very big limiting factor to use it on a community cloud where bandwidth and availability are not guaranteed. Another limitation of CloudStack is the close relationship between physical and virtual networking that could need additional work, and may not even be plausible for community networks where the topology of the network is not fixed and can change frequently with or without prior notice.

Moreover, storage could be a problem, since secondary storage needs to be always available to the subnets and all the infrastructure in a zone has to share the same secondary storage. Also primary storage has to be separate from the host machines and needs dedicated hardware, which may not always be possible to have on a community cloud. As a final remark CloudStack shows a big potential for cross-cloud federation, but has still limitations it needs to surpass in order for it to become more mature and match the features of the other VIMs.

3.1.3. **OpenNebula**

OpenNebula is an European open-source cloud computing toolkit for managing heterogeneous distributed data centre infrastructures. The OpenNebula toolkit manages a data center’s virtual infrastructure to build private, public and hybrid Infrastructure as a Service (IaaS) Clouds. OpenNebula orchestrates storage, network, virtualization, monitoring, and security technologies to deploy multi-tier services, e.g., compute clusters, as virtual machines on distributed infrastructures, combining both data center resources and remote Cloud resources, according to allocation policies. Table 4 shows the features in OpenNebula 4 release.
Table 4: Features in OpenNebula 4.0 release

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dashboard (Sunstone)</td>
<td>Provides a user-friendly and minimal web-based user interface for managing Open Nebula services.</td>
</tr>
</tbody>
</table>
| Scheduler          | - match making scheduler (*mm_sched*) that implements the *Rank Scheduling Policy*  
                      - Pre-defined placement policies: target, heuristic and implementation.  
                      - Hardware and hypervisor agnostic.                                        |
| Storage (Datastore) | - Shipped with 5 different datastore types: system (hold images for running Vms), file-system (store disk images in a file form), iSCSI/LVM (images in a block device form), VMWare (handles the vmdk format) and vmfs (VMFS format for VMWare hypervisor).  
                      - Transfer manager with 4 different ways to distribute datastore images to the hosts: shared file-system, ssh, iscsi, vmfs and qcow.  
                      - All the drivers can be easily customized.                                        |
| Network            | - When a new Virtual Machine is launched, OpenNebula will connect its network interfaces (defined in the NIC section of the template) to the bridge or physical device specified in the Virtual Network definition. This will allow the VM to have access to different networks, public or private.  
                      - Five Virtual Network Manager drivers: firewall rules, 802.1Q, ebtables, ovswitch and VMWare |
| Security           | - Users and Groups Subsystem for authentication and authorization of requests with complete functionality for user management: create, delete, show...  
                      - Pluggable authentication and authorization based on passwords, SSH RSA keypairs, X509 certificates, LDAP or Active Directory |

OpenNebula aims at supporting most used hypervisors. Currently supported Hypervisors are:

- **Xen**: powerful, efficient and secure feature set for virtualization of x86, IA64, PowerPC and other CPU architectures. It delivers both paravirtualization and full virtualization.

- **KVM**: Kernel-based Virtual Machine. A full virtualization solution for Linux on x86 hardware containing virtualization extensions.

- **VMWare**: The VMWare Drivers enable the management of an OpenNebula Cloud based on VMWare ESX and/or VMWare Server hypervisors. It uses libvirt to invoke the Virtual Infrastructure SOAP API exposed by the VMWare hypervisors, and it entails a much better integration and performance than the Java based drivers traditionally found in the OpenNebula distribution.

Linux Containers (LXC) is not natively supported in ONE yet, even though OpenNebula mentioned that they are interested in supporting it there is no official announcement about it. To test LXC
drivers we have started with a version that was developed by China Mobile team for OpenNebula 3.2.1, and then for OpenNebula 4.

### 3.1.3.1. **Open Nebula 3.2.1 and LXC**

We have focussed on two things: testing OpenNebula 3.2.1 and LXC drivers and trying to port these drivers to OpenNebula 3.8. Now that OpenNebula has released its 4.0 version we plan to try to reuse these drivers to the last version. The current status of the testing/development can be seen in this table:

<table>
<thead>
<tr>
<th></th>
<th>Host Monitoring (adapted for each host OS)</th>
<th>LXC Driver Working</th>
<th>Tested operations</th>
<th>Next Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE 3.2.1 &amp; Ubuntu 12.04 host</td>
<td>●</td>
<td>●</td>
<td>Create / Shutdown</td>
<td>Test more operations and try to connect to a LXC. Check Virtual Network cfg.</td>
</tr>
<tr>
<td>ONE 3.2.1 &amp; confine node with OpenWrt</td>
<td>●</td>
<td>●</td>
<td>-</td>
<td>Currently testing libvirt in OpenWrt. Basic features are working.</td>
</tr>
<tr>
<td>ONE 3.9 &amp; Ubuntu 12.04 host</td>
<td>●</td>
<td>●</td>
<td>-</td>
<td>Focusing in ONE 4.0 beta recently launched.</td>
</tr>
</tbody>
</table>

Our target is to make LXC drivers working with ONE 3.9 (or current beta version of 4.0), but firstly we wanted to check if the existing “drivers” developed by China Mobile were working properly. We have done a short guide explaining this procedure [12]. Our goal is to make it work using a CONFINE node as a host but libvirt portability is becoming more difficult than expected and we are on the way to port it. We have done some ports but only with basic functionalities working. Having seen that libvirt compatibility with Ubuntu is trivial we proceeded to try the drivers with an Ubuntu host instead of OpenWrt in a core 2 duo machine. We have used the architecture showed in Figure 5.
Our goal is to launch virtual machine instances with LXC as hypervisor on the nodes; however OpenNebula does not support LXC as standard. To be able to use OpenNebula to deploy LXC domains in CONFINE nodes we should use specific LXC drivers. These drivers have two main features:

- **IM_MAD**: a series of remote scripts that are able to monitor the remote hosts. These drivers have been adapted to work properly with OpenWrt.
- **VMM_MAD**: a series of remote scripts to manage LXC domains. LXC drivers use virsh commands to monitoring hosts and operating virtual machines.

Currently OneLXC supports some simple functions:

- monitoring host information, for example, cpu and memory
- deploy/delete/monitoring(poll) LXC domains and their information

After having configured the drivers and compiled OpenNebula with them we have been able to do some tests.

Open Nebula 3.2.1 does not support custom drivers (ONE > 3.8 does) when launching instances from Sunstone (ONE dashboard), so we should create the virtual machines using ONE CLI. To create a virtual machine using LXC drivers we should first add the node with such the drivers using the following command:

```
$onehost create 10.139.40.29 im_lxc vmm_lxc tm_ssh dummy
```

(*tm_ssh indicates that the transfer method is going to be ssh and the last parameter is the use of a dummy virtual network)

Figure 6 shows the result.

![Figure 6: ONE: 'onehost list' command](image)

Now that we have added the host with LXC drivers we can create a new VM. To do this we execute onevm create command. We have previously defined a template called lxc_template with CPU/mem parameters and image directory.

```
$onevm create lxc_template
```

![Figure 7: ONE: 'onevm list' command](image)
In Figure 7, we see that our virtual machine is in “running” status meaning that it has worked properly. If we want to see more information of our lxc instance, we can use the command as shown in Figure 8.

```
$ onevm show 7
```

**VIRTUAL MACHINE 7 INFORMATION**

- **ID:** 7
- **NAME:** Supertest2
- **USER:** oneadmin
- **GROUP:** oneadmin
- **STATE:** ACTIVE
- **LCM STATE:** RUNNING
- **START TIME:** 02/28 13:15:17
- **END TIME:** -
- **DEPLOY ID:** lxc-7

**VIRTUAL MACHINE MONITORING**

- **NET_RX:** 0
- **NET_TX:** 0
- **USED CPU:** 0
- **USED MEMORY:** 0

**PERMISSIONS**

- **OWNER:** um-
- **GROUP:** --
- **OTHER:** --

**VIRTUAL MACHINE TEMPLATE**

```
CPU=1
DISK=[
  DISK ID=0,
  SOURCE=/var/lib/lxc/debian32-rootfs.tgz ]
MEMORY=1024
NAME=Supertest2
TEMPLATE_ID=2
VCPU=2
VMID=7
```

**VIRTUAL MACHINE HISTORY**

<table>
<thead>
<tr>
<th>SEQ</th>
<th>HOSTNAME</th>
<th>REASON</th>
<th>START</th>
<th>TIME</th>
<th>PTIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10.139.40.29</td>
<td>none</td>
<td>02/28 13:15:24</td>
<td>34 19:28:56</td>
<td>08 00:00:10</td>
</tr>
</tbody>
</table>

Different from kvm and xen image files, the lxc domain's image actually is a directory called “rootfs”. For the sake of transferring it easily, we compress it, copy it to the target host and decompress it again in the destination. Note the file permission of /usr/bin/sudo of the lxc domain must be 4755. Otherwise, it can not use root privilege to execute command in the lxc domain. In this test we have used the same image that CONFINE uses for the Debian-experiment.

In the following some of the bugs and problems are reported:

- There is a synchronous problem between the “cancel” operation and “tm_delete.sh” for OneLXC driver.
D2.1 System requirements and software architecture

- OneLXC driver cannot implement “reboot”, “shutdown”, and “restart” operations because maybe libvirt or LXC does not support it.
- How to generate the lxc “config” file in “rootfs” directory dynamically? Note that each lxc domain has different image file path.

3.1.3.2. OpenNebula 4.0 and LXC

LXC Drivers for OpenNebula 4.0 are still under development [13], but they are based on 3.2.1 working drivers and they follow the same structure and features. As in the previous tests our purpose is to be able to launch virtual machines instances with LXC hypervisor using OpenNebula software.

To launch virtual machines instances we will use the Sunstone dashboard for all the steps: adding a host, uploading an image, creating a template and starting a virtual machine. First we should add a host with LXC drivers activated as in Figure 9.

Figure 9: ONE: LXC create hosts with LXC drivers

Then we should add the image to the datastore (Figure 10), for this experiment we have used the same Debian image for LXC as the one used in the CONFINE project [14].

Version 1.0 – 30/06/2013
Once we have added the image, we can create a template. For this example we will define a template with 512MB of RAM, 1 processor available and the debianbt32 image we have previously uploaded (Figures 11 and 12).
Now we can launch a virtual machine instance using LXC as hypervisor. As we write this evaluation we are still debugging some issues in the drivers that do not let us start an instance correctly.

3.1.3.3. Installing and configuring OpenNebula in a CONFINE network.

In the rest of this section we go through the steps to get OpenNebula working with LXC as hypervisor and CONFINE research devices (RD) as hosts. Figure 13 represents the OpenNebula CONFINE architecture. As the figure shows, the OpenNebula front end runs Ubuntu 12.04 64b and OpenNebula 3.8.0, the VCT controller is installed in a host, and the RDs are loaded and configured with the last openwrt-confine-dist image.

The steps to setup the service are as follows [10]:

1. Set up both research devices and initialize them using vct controller.
D2.1 System requirements and software architecture

1. Following most of the steps as seen in the CONFINE torrent tutorial
2. Install OpenNebula software in Frontend
   1. Follow OpenNebula installation guide
3. Configure Research Devices to be used as OpenNebula hosts
   1. Creating oneadmin user/group and configuring SSH passwordless.
   2. mkswap for OpenWrt must be installed from confine dist repositories.
4. Create LXC drivers to manage hypervisors in hosts (vmm, virtual network, transfer manager & information driver)
   1. Edit oned.conf file.
      1. Add new lxc vmm section (Listing 1)
      2. Add new lxc im section (Listing 2)
   2. Create /vmm_exec/vmm_exec_lxc.conf (default cfg file)
   3. Create folder and script files for vmm driver
   4. Create folder and script files for im driver (not necessary but recommended)
   5. Create new host
   6. Create new virtual machine (load image, template, and create instance)
5. Mixing OpenNebula and CONFINE slice/slivers.
   1. Cloud Community brainstorming
      1. Users could provide machines properly configured to work as hypervisors.
      2. Use confine network as service network to connect front end open nebula installation with VM hosts.
      3. Keep 1 sliver free in each RD to dynamically launch images for clod purposes using open nebula software.
      4. integrate some vct scripts with open nebula commands
6. TODO
   1. Users in OpenNebula
   2. Virsh drivers
   3. Testing*
In the rest of this section we explain how to prepare a CONFINE node to work as an OpenNebula host. The official requirements of the nodes to be used as Open Nebula hosts are the following:

- SSH server running: Already installed in CONFINE nodes (Dropbear SSH client).
- Hypervisor working properly configured: LXC
- ruby >= 1.8.7: Already installed in CONFINE nodes.

After providing all the requirements to work with OpenNebula, we can configure the nodes through the following steps.

1. The first step required by OpenNebula hosts is to have a user called oneadmin with password oneadmin as well as a group named oneadmin. Confine image does not have
D2.1 System requirements and software architecture

useradd command so we will have to add the user manually. To do it, please login to CONFINE node and follow these steps:

- Edit /etc/passwd file and add the following line:
  
  oneadmin:x:1001:1001:Oneadmin User:/var/lib/one:/bin/ash *(Make sure that group id is the same as in the frontend.)*

- Edit /etc/groups and add the following line:
  
  oneadmin:x:1001:oneadmin

- Edit /etc/shadow and add the following line:
  
  oneadmin:abcdefgeh

- Change oneadmin passwd:
  
  $passwd oneadmin  -> (new passwd = “oneadmin”)

- Create home directory for oneadmin user:
  
  - mkdir /var/lib/one
  - chown oneadmin /var/lib/one
  - chgrp oneadmin /var/lib/one

- Create .ssh directory
  
  - mkdir /var/lib/one/.ssh
  - chown oneadmin /var/lib/one/.ssh
  - chmod 700 /var/lib/one/.ssh

- Create remotes directory
  
  - mkdir /var/tmp/one
  - chown oneadmin /var/tmp/one

2. The second step is to prepare the passwordless SSH connection between the frontend and each host. We should remember that when we launch Open Nebula in frontend we always do it as oneadmin user. In the frontend we should do the following:

- Generate SSH key
  
  $ssh-keygen -t rsa

- Copy the key to the host
  
  $scp ~/.ssh/id_rsa.pub root@hostip:/var/lib/one/.ssh/id_rsa
  $scp ~/.ssh/id_rsa.pub root@hostip:/var/lib/one/.ssh/authorized_keys
Now we should be able to connect from the frontend to the host via SSH without password:

```bash
$ssh oneadmin@hostip
```

3. Install in the host the `swap-utils_2.21.2-1_x86.ipk` file which is at the CONFINE repository.
   - We have to add to the file `/etc/opkg.conf`:
     ```
src/gz confine_repo http://repo.confine-project.eu/packages/x86/packages/
```

   others:
   - Install “sudo”
   - `sudoers` -> `oneadmin ALL = NOPASSWD: /bin/tar`
   - `vmm_exec_lxc.conf` -> `EMULATOR = /usr/lib/libvirt_lxc`

### 3.1.4. Evaluation of different Cloud platforms

Table 6 shows compares different Cloud platforms. Only OpenStack, OpenNebula, CloudStack have a developer community. The CONFINE Controller is maintained by the CONFINE project over the next 2.5 years, but has currently no solid developer community that will maintain it. P2PCloud is currently not more than a incomplete research prototype. It has no developer community. The centralized nature of OpenStack, OpenNebula, CloudStack would mainly fit if the community cloud resources are located at community data centers.

The design of OpenStack, OpenNebula, CloudStack would hardly allow decentralized management and distributed cloud resources. Openstack and OpenNebula have a large number of real users. It can be expected that the platforms are fairly stable. Extension of OpenStack and OpenNebula to community networks will need to adapt to the design of OpenStack and OpenNebula.

An approach could be to develop the envisioned crowd-funded centralized community clouds based on OpenStack and/or OpenNebula, while CloudStack is very recent to be clear about its future. This could correspond to the vision of many local cloud centers spread over the community network, managed locally by a group, using an existing management platform of their choice. This approach would provide the community network with a centralized cloud computing infrastructure at short term that could be offered to users.
### 3.2. COMMUNITY SOFTWARE DISTRIBUTION

In this section, we describe our community software distribution, Guifi-Community-Distro, in relation to the already existing Guifi-Distro, and explain how Guifi.net users use the distro to configure their systems when connected to the Guifi.net network for day to day use. The Guifi-Community-Distro software distribution packs the software applications needed to provide the required Cloud services to the Guifi.net community network [17].

Guifi.net is an open, free and neutral telecommunications network that is built through a P2P agreement among its users. Guifi.net is owned by all who join. It is a collaborative project horizontally managed and composed by individuals, organizations, enterprises, education institutions and universities and government offices. Guifi.net consists of more than 20,000 nodes and thousands of Km of links. The users can benefit from services such as network auto-configuration to configure, offer and access remote services. A community network is an emerging model for the future Internet across Europe and beyond where communities of citizens build, operate and own open IP-based networks.

---

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Openstack</th>
<th>OpenNebula</th>
<th>Cloudstack</th>
<th>ConfineController</th>
<th>P2PCloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized location of cloud resources</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>×</td>
</tr>
<tr>
<td>De-centralized location of cloud resources</td>
<td>Limits not clear</td>
<td>Limits not clear</td>
<td>Limits not clear</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Heterogenous hardware</td>
<td>not clear how it scales to small devices</td>
<td>not clear how it scales to small devices</td>
<td>not clear how it scales to small devices</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Management</td>
<td>Same for all cloud resources</td>
<td>Same for all cloud resources</td>
<td>Same for all cloud resources</td>
<td>Same for all cloud resources</td>
<td>Possibly customizable</td>
</tr>
<tr>
<td>Developer community</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Management of slice network on L3</td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
The hardware we use for testing are the community boxes based upon the hardware used in the CONFINE project. The main operating system of the box is OpenWrt and it supports running other operating systems, such as Debian GNU/Linux.

This Guifi.net software distribution is running at nodes connected to the Guifi.net network. We will adapt and extend it to operate inside the community boxes. We will run a Debian system inside the community box and we will have our own network interfaces that can be configured. We need one network interface which connects to the testbed network.

A software repository of the software distribution is where the nodes connect to install and update software. The main page for information about the repositories is at the wiki.guifi.net website that includes information to add the keys for the signed packages and the lines to add for adding the repositories to the system. The information about the services that are included in the actual Guifi.net distribution is at the Guifi.net blog.

As the blog says, the actual software distribution allows to configure:

- Graphics server: Allows users to draw node statistics.
- Federated proxy: Allows users to configure access to the Internet.
- DNS Server: Allows users to configure the DNS server.

The Guifi.net web is an important component for applications. For example to access the federated proxies, the user has to be registered at the web site. All this services are included in the form of Debian packages, with a Debian repository. The software is distributed adding sources.list files to a standard Debian software distribution, and with that we can install the Guifi.net software.

Guifi-Community-Distro is a software distribution based on Debian/GNU Linux and includes packages from the actual Guifi.net software distribution, which is based on the same Debian/GNU Linux distribution. Some packages already exist on the standard distribution, and others do not. We can add specific functionality and integrate the new software creating our own software repositories with standardised Debian package formats for easy installation and upgrade.

We mainly need support for:

- Guifi.net Firefox add-on
- Service Location: Avahi
- Distributed storage filesystem: Tahoe-LAFS
- Video Streaming
- Scripts and applications as glue to fit all together
- UI applications

The software will be fetched from the various sources added in the form of standard software repositories. One of the advantages of the software distributions is that the user can fine tune

1[http://guifi.net/node/29320](http://guifi.net/node/29320)
some configurations and include new versions while accounting for the updates of the previous configurations. We can customise the server configuration using the central software repository.

We can begin to use the software inside a local network for the initial stages of our experiments. We will configure:

- A global software distribution repository, as the usual Debian repositories.
- A central customised software repository, with the developed applications in the project and customised configurations.

Finally we can integrate the results in the deployment in the community networks.

### 3.3. SERVICE DISCOVERY - AVAHI

Avahi is a free zero configuration (zeroconf) implementation, including a system for multicast DNS/DNS-SD service discovery. This application allows programs to publish and discover services and hosts running on a local network with no specific configuration. For example, a user can plug its computer into a network and Avahi automatically finds printers to print to, files to look at and people to talk to, as well as advertising the network services running on the machine [16].

The growth of mesh networks makes available possibilities to the users where they can discover, access, add and manage new network services. The usual method of accessing and sharing services uses a centralised service in a fixed location, where the client uses invariant data to access services in that fixed location, but with mesh networks the services are moving around the network, using different locations (roaming) to offer services.

We describe here the results of our work on auto-configuring local networks and to discover and share services. The first need is to look for the possibility when a computer is plugged in a network to get the computer auto-configured, to offer services to the network and to get an automatic list of available services in the network. All that with limited intervention by the user. Then we want to have the possibility of expanding the offering of the services beyond the local network and beyond mcast limitations. We are also interested in the possibility of discovering and accessing remote services in other networks as easy as possible. We are going to expose some techniques, implementations, examples and some ideas of possible use cases.

Before explaining Avahi, we need to define the following terms:

- **Zeroconf networking**: it is a set of techniques that allows devices to connect and automatically configure an IP network; with that we can also share and access to those services automatically. There is an IETF working group called Zeroconf Working Group. Zeroconf allows no initial configuration for a local network, where nodes subscribe to a multicast address: 224.0.0.251 for IPv4 and ff02::fb for IPv6 link-local addressing. The subscribed nodes register (locally) their services and when a query is received they respond offering a list of services. Zero Configuration Networking provides:
  - **Assignment of IP addresses**:
    - IPv4: 169.254.0.0/16 (automatic, it’s used when no DHCP is available).
D2.1 System requirements and software architecture

○ IPv6: fe80::/10
  • Automatic assignation and name resolution.
  • Automatic location of network services.
  • The host part of the address is dynamically generated by the host. It's randomly generated in the case of IPv4.
  • Multicast DNS (mDNS): its standard is described in RFC-6762 [39]. mDNS allows to use DNS services without a centralised DNS server. It has resiliency, as opposed to the usage of a centralised DNS server. mDNS allows to query and announce services to the network using DNS packets.
  • DNS Service Discovery (DNS-SD): There is a RFC draft at IETF site about DNS-SD [40]. As described in the document, it works using DNS queries to discover offered services in a given domain. It uses unicast queries, as opposed to multicast queries used by mDNS. When a user queries the server looking for a specific service at that specific domain (allows remote networks services discovery), the user receives a list of services using standard DNS queries. DNS-SD is compatible with, but not dependent on, Multicast DNS.

Avahi implements DNS-SD and mDNS specifications for zeroconf networking. It uses D-Bus to communicate between user applications and a system daemon. Avahi is also a system to automatically discover services in a local network. It is based on mDNS/DNS-SD protocol suite. Avahi is widely used in various operating systems and it is installed by default as a standard in GNU/Linux operating systems. For example Avahi is used in the meshed networks of the OLPC project. Avahi has a graphical user interface called “Avahi Zeroconf Browser” that can be run with the command avahi-discover.

The scalability of Avahi is limited to approximately 1000 nodes\(^2\). However, DNS-SD can be used to scale Avahi Services to more nodes.

Two known problems of Avahi are:

1. Possible big inter-network problem (network packets saturation): if we have various interconnected networks and they form a big network (WAN), and if we propagate the client queries (mcast) directly from one network to another network, the networks can be saturated of mcast packets.

2. More than one Avahi reflector in the same network: it is recommended not to put more than one reflector in the same network, because they can do ping pong of reflected packets.

Possible solutions to the above problems are the followings:

1. Centralised solution: this solution avoids to forward mcast packets to long distances, but with the need of a centralised DNS to reach long distances.

\(^2\)http://blog.gmane.org/gmane.network.zeroconf.workers/month=20060301

Version 1.0 – 30/06/2013
Page 36
D2.1 System requirements and software architecture

- avahi wide-area: The Avahi enable-wide-area configuration parameter enables wide area DNS-SD, aka DNS-SD over unicast DNS. By default this option is set to yes.

2. Decentralised solutions: these solutions are based on the forward of the mcast packets.

- Iptables: If a router is blocking mcast packets, we can define iptables rules to pass the mcast packets from one network to another.
- avahi reflectors: The enable-reflector parameter of avahi-daemon reflects incoming mDNS requests to all local network interfaces.
- mDNS repeater: mDNS repeater is a service which allow us to forward Multicast DNS (mDNS) packets between two different networks.

3. Solutions to Avahi in WAN from Battlemesh v6: On Friday, 19th of April at Battlemesh v6, a lightning talk about Avahi was given. We were looking for a solution to avoid the use of a centralised DNS when using Avahi in WAN. We needed the support and help from the people at the Battlemesh. The response was great, and from many participants as you can see in the next sections. First of all thank you to all people involved in helping for the development or implementation of a distributed DNS service.

- Spain DHT: In Spain we received the information about people working with DHT and Avahi. A document that proposes the use of DHT (OpenDHT) is: “Z2Z: Discovering Zeroconf Services Beyond Local Link”, and people working with possibly scaling Avahi to our interest, 20.000 nodes or more, presented in the paper “Self-Organization in Community Mesh Networks: The Berlin RoofNet”.
- FreiFunk DHT: An email was received about using DHT from the FreiFunk network. They work on a distributed DNS service based on DHT. As they say in the message, it has not been deployed, but it seems to work.
- Italy: Ninux and OLSR: The OLSR mDNS extension for service discovery Ninux is using a solution based on a plugin for OLSR, they wrote “The OLSR mDNS extension for service discovery” in their network.
- USA: They use an app to register services with a text file. They showed a web interface to give access to the network services. Related to the Commotion project.
- Germany: Anycast gateways. They say that it is very interesting someone being working on that idea. The project name is: “Distributed DNS Cache”, yet in the first stages working in a written document about the idea. The idea is a mix between ideas from Guifi.net and the idea of using Anycast DNS. We talked about both ideas and founded with more partners from Battlemesh, the project “Distributed DNS Cache”. The stage of the project is in writing the first draft of the idea being developed.
- Experimental research: Actually other options are being researched to scale Avahi to WAN. We knew some ideas at the Battlemesh v6. The ideas are yet on initial stages of research and we will follow them.
3.3.1. **One Laptop Per Child (OLPC)**

One Laptop Per Child (OLPC) is a project for developing countries to develop a cheap and robust computer by less than 100$ to be used by children. It’s carried by the Foundation with the same name with the mission of: “OLPC’s mission is to empower the world's poorest children through education.”

The OLPC computers network is configured automatically: each computer is connected to the other computers, they form a mesh network and they access to the services of the other computers of the mesh and each one offer services and share Internet connection. As said, the project uses Avahi.

We can take as a base the OLPC XO-1 project computer to explain hardware level operation of mesh networks. It uses the protocol 802.11s. It forms a mesh with the other computers, where the intervention of a system administrator is not necessary. The IPs are configured automatically, the services are discovered automatically and the Internet connection too; when a node has Internet connection it shares the connection between the other nodes of the mesh.

The nodes participate in a MANET network, a self-configuring infrastructureless wireless ad-hoc network composed by mobile devices. But the protocol implemented in the hardware is proprietary code. Although there are free software alternatives such as B.A.T.M.A.N. or OLSR run at the data-link layer (Layer 2), they do not (as in 2011) run on the OLPC wifi hardware.

3.3.2. **Community network logical topology**

For Avahi, the community Cloud forms a Layer 2 fully connected network topology. As a P2P network, it is plain and has no hierarchies. Figure 14 shows the vision that the devices are free and independent inside the Cloud and can be anywhere and be of any kind. The devices include managed community boxes, used directly by the user. The Cloud services are transparent to the user.
In Avahi, each device is connected to each other directly as in a switch, it is done at the OSI Layer 2. The services are accessed directly from one device to another and there are no NAT limitations. With the usage of Avahi inside the cloud we can without intervention of any intermediary:

- Obtain automatically an IP address without the need to configure in the network a DHCP server.
- Share and discover directly services between the network devices.
- Access to nodes using names instead of IPs without the need of a centralised DNS server.

3.3.3. SERVICES AND COMMUNITY

Based on the prior description, now we make some proposals illustrated with some Figures (Figure 15 and Figure 16). Inside the Community cloud there are services, and they're local. A local service can:

- Stay only locally accessible and be unpublished, we say that the service is unpublished.
- Be globally accessible an be published, we say that the service is published.

And we need to state first some assumptions. When we talk about the local part of Avahi, we use mDNS queries and all is automatic. In the global part we have to use globally reachable DNS servers if we are offering services which we want them to be possible to be accessed from global locations and DNS-SD queries to those DNS servers if we are clients. When we talk about remote queries to discover local services offered globally, the DNS server has to be globally available.
When we talk about offering services globally, we suppose that the cloud nodes can be globally accessed.

In Figure 15, we show a general diagram about services and the connection to them. First we talk about offering community services globally using Avahi and DNS, then we describe in the next sections the use cases showed in Figure 16 to participate in the community Cloud or to only access given services. In the following there are some example of community services using Avahi:

- Offering global services: In this case Avahi is configured to register the given services in a DNS server. If the DNS is globally reachable, when a DNS-SD query arrives to the DNS server, a list of services is returned. With that, services can be globally queried by other clients using Avahi. Those services are published.

- User1 connects and participates in Community: In this case User1 (Figure 16), which is at the Guifi.net network, participates directly in community. User1 can offer services to the community Cloud and uses the given services given by the cloud accessing them locally. All already mentioned possibilities of Avahi in local networks are available to the new node. The user connects to Community using a VPN Layer 2 connection. User1 uses the community DNS to register her/his offered services if User1 wants that they can be queried from outside. User1 runs local services (unpublished) in the beginning, and some of them can be globally accessible (published) later.

- User2 accesses to Community services: In this case User2 (Figure 16), which is also at Guifi.net network, does not use a Layer 2 connection and can only access and use the globally published community services. In this case User2 configures Avahi to query the available services offered by the community Cloud from a location outside the Cloud. User2 accesses only to globally reachable (published) services.

With the use of Tinc, the Layer 2 VPN used in the project, we can add devices to the community Cloud offering services that are behind a NAT. The services are registered locally in the community Cloud DNS and can be accessed from outside the community Cloud. All kind of services a user can have at home behind a NAT for example can be accessed using the community Cloud. The essential tools are Tinc Layer 2 VPN, and Avahi.

We are actually using this idea to build software on-top of the Community-Lab testbed for the project. For example a user can install Avahi and to offer services using his own externally reachable DNS server. The services can be discovered with a simple DNS query. This case can be applied to Guifi.net users which can offer services which can be discovered globally. Services can be discovered remotely by doing queries to the globally available DNS servers that the user at Guifi.net has installed. If a user wants to know which services are offered by a domain, the user only has to do a query to that domain and she/he receives a list of the available services. A group of nodes with Avahi can auto-configure their network devices and offer globally their services.

Another example is a user behind a NAT, who wants to register her/his services with the community Cloud. She/he can use Tinc to connect her/his mobile device (for example) to the community Cloud and with this, other users can query for a list of global services offered by the user from the community Cloud using an Avahi client including the services given by the offering user. The services offered by the client connected to the Cloud, can be discovered remotely by another client that uses Avahi and queries the Cloud for services.
To summarised this section, we can use Avahi in a local network for node auto-configuration, automatic service share and discovery without intermediaries. If we have a network with around 1000 nodes, we can use Avahi directly without centralised DNS services. If we have a network of more than 1000 nodes it’s convenient to use too DNS-SD to reach remote networks, and avoid mcast saturation problem, it uses standard DNS queries.

We can use Avahi to register local services (unpublished) and those services can be offered globally (published). Any client from any location can be part of the Community cloud and offer services globally, despite the user being behind a NAT. The state of the art in researching distributed DNS is open and actively worked nowadays.
Proposal:

- The Community cloud is formed by Community boxes in the beginning. The cloud provides services. Some more devices and services can be added. Globally published services (published) are registered in globally accessible DNS servers.

- Inside the Community cloud, there are local services, some of them are globally published services (published) and others are only local services (unpublished). A local service (unpublished) can be published and became globally accessible (published) too.

- User1 can connect to the cloud directly using Tinc Layer 2 VPN. With that User1 has local access to the cloud and User1 can publish new services to the cloud, those services can be only local (unpublished) or can be, despite being local, globally published (published) too by User1 from the Community cloud.

- User2 queries from global access to the global side of the Community cloud looking for globally published (published) services and gets a list of globally published (published) services from the Community cloud, then User2 can access to the globally published services (published) included in the received list.

Figure 15: Community ecosystem with connections and devices
D2.1 System requirements and software architecture

3.4. DISTRIBUTED FILESYSTEM - TAHOE-LAFS

Tahoe-LAFS is a distributed file system [18]. We can see it as a RAID. The files can be mutable or immutable and can be given permissions. Tahoe is a Free and open Cloud storage system, and it is the first decentralised storage system with provider-independent security.

The project uses the Principle of Least Authority (POLA) principle that is operations can be done with the least possible privileges. The performance is not a priority in the project. The priorities are the security with encryption, user's privacy and data availability. The used Tahoe version is v1.9.2. The minor required version is some 1.0.0-compatible release. The Python requirements are: Python > v2.4.4. Python 3 does not work. The project has a commercial and Alpha service at Amazon S3 Cloud. Tahoe works inside OpenWrt. The project can be considered mature.

There are other alternatives to Tahoe-LAFS. OpenStack has Swift, too low level for an usual user and has not NAS, but you can use GlusterFS. GlusterFS is a distributed file system but does not encrypts by default. This is a very reduced set of popular alternatives as example, the major drawback is that they have not been designed with security and privacy in mind or cannot be used easily with FUSE.

Figure 16 shows the Taho-LAFS architecture, which is taken from the Tahoe-LAFS official project main web page. We can identify four components in this architecture (from right to left in Figure 16): client, gateway, introducer and storage nodes. Usually client and gateway are in the same host.

- **Client**: The client uses the introducer to know to which servers to connect. The connection between clients and storage servers is a direct connection. The data is transferred directly. The client side can be divided in two parts:
  - **Client of storage nodes**: Connects to the storage nodes.
  - **Client of the gateway**: Connects to the gateway. Can connect using a CLI, SFTP, FTP, FUSE (sshfs) or an application using the Web REST API.

- **Gateway**: The main responsibilities of the gateway are to connect to storage servers (storage client side), split and encrypt the data and give APIs to clients. A client/server node that provides web-API service is called a gateway. The gateway usually is a local service, but can be a remote service.

- **Introducer**: The main responsibility of the introducer is to act as a simple publish-subscribe hub. The servers connect to the introducer and announce their presence and the client nodes connects to the introducer and get a list of all servers. The introducer doesn't transfer data between clients and storage servers. Which is done directly between both sides. Each client, then, connects to every server and a "bi-clique" topology is created. A general definition of clique can be adapted to our topology: Clique: Group of nodes which each one know each other. Clients and servers depends on the introducer. It's position is fixed, but more than one introducer can exist.

- **Storage nodes**: The data is stored in Storage Nodes.
To talk about data distribution we can say in Tahoe jargon: 3-of-10, we split our data in 3 parts and we use 10 storage servers, the data is (with a pseudo-random function) distributed across the 10 storage servers keeping 1/3 of the data on each server, or in other words: the amount of data stored on each storage server is 1/3 of the original data size.

The data is kept plain at the client side and can be "seen" clear at the Gateway side. The data stored in the storage servers can't be "seen" plain or changed, can only be "seen" plain or changed in the client side. The Gateway encrypts and keeps the data distributed across the various storage servers. The storage server does not knows about the files stored and never can reconstruct a file by itself.

A user can host his own gateway and rely only on external storage servers. If the user installs locally her/his own gateway, it adds complexity, but on the other side, doing this, the user gets more freedom and the service is more decentralised. Using an external gateway allows to the gateway operator to see the raw data. But the user hasn't to worry about nothing more than using a simple CLI client or FUSE or others as described before. A user can share a file if she/he shares the URI of that file. There is no other way to know or look for a file. It's recommended that data between gateway and client goes through SSH.
D2.1 System requirements and software architecture

3.4.1. General Usage

As a general and basic usage we can install Tahoe-LAFS in Debian for example (it’s on repositories). We can do basic operations as create a Tahoe-LAFS directory, put and get files. We can use the Public Test Grid (pubgrid, introducer and storage servers from volunteers); the gateway doesn't need to be installed because the gateway is at the client side, the introducer and storage servers don’t need to be installed or configured, they own to volunteers of the Public Test Grid. For Public Testing, we only have to install the client side (gateway included, transparent installation, it’s on localhost 127.0.0.1) and configure access to a predefined (Public) introducer.

We assume a GNU/Linux environment, commands were tested on Debian Wheezy. The only command to execute as root is the installation command for Tahoe-LAFS, the other commands are executed as normal user. The first step is to install as root the client side: apt-get install Tahoe-LAFS, then we have to configure the client side as normal user to use the example introducer of the Public Test Grid, editing ~/.tahoe/tahoe.cfg (Listing 3).

```
[client]
introducer.furl = pb://hckqqn4vq5ggzuukfztpuu4wykwefa6d@publictestgrid.dnsd.info:50213/introducer
```

Listing 3: Tahoe-LAFS introducer.furl

Then start the client side service typing Tahoe start as normal user. As clients, Tahoe gives us some options to access files and directories:

- Basic command-line interface (CLI): Tahoe provides its own command-line interface (CLI). We can test the basic command-line interface typing some example commands:
  - “tahoe create-alias tahoe”: it's used to create a new directory.
  - “tahoe cp myfile.txt tahoe:myfile-uploaded.txt”: (it’s important to type tahoe followed by : to refer to the linked directory in the GRID) used to copy a file (upload) to tahoe.
  - “tahoe ls tahoe:”: it's used to list the contents of the tahoe directory. We get the contents of the directory, that only contains the file name of the previous uploaded file myfile-uploaded.txt.
  - “tahoe cp tahoe:myfile-uploaded.txt myfile-downloaded.txt”: it's used to copy a file from Tahoe to the local file system.

- FUSE (sshfs): We can mount a Tahoe directory typing as normal user: sshfs 127.0.0.1:tahoe tahoe-mount. Then we can work with that directory as it was local, it's in the user’s file system. NOTE: The user needs to be in the fuse users group.

- Other client-side possibilities: Apart from using the CLI and FUSE, the client can use:
D2.1 System requirements and software architecture

- **WAPI**: It's the Restful API. Can be used directly (web interface) or we can program apps to use this Restful API.
- **FTP**: Usual FTP client.
- **SFTP**: Usual SFTP session.

### 3.4.2. **Tahoe-LAFS Project and Community**

**CLOMMUNITY** uses the CONFINE testbed, based on it, we can use the nodes for implementing services as Tahoe-LAFS. **Tahoe-LAFS** provides to the project a distributed file system with encryption and privacy with very simple usage at the client side. The privacy is assured, because the storage server operator can not see or change the stored data nor know about a file name or contents. That only can be known using a URI.

If we want to use Tahoe-LAFS in CLOMMUNITY, first we need to setup our own storage servers and a gateway. The users can store securely files and works internally as a RAID, distributing the data across the various storage servers, but the user only knows a directory mounted via sshfs in the simplest use case. Clients, introducer and servers can be inside the slivers of the testbed. And we need a gateway too, that can be on the client side. Storage servers can be added or removed at any time.

The simplest use case requires a client (with local gateway), a introducer and storage servers (2 or more, we want test data split). The client can be behind NAT, introducer and storage servers have to accept incoming connections, from clients and servers, and clients respectively. Perhaps, the introducer can be placed as an added service to the place where the confine-controller software is installed. Tahoe clients can be placed inside the slivers, perhaps behind NAT if IPv4.

About the gateway, if it is local to the client, can be placed inside the sliver. If it's shared, can be placed too in the same place as the confine-controller software. Remember that placing the gateway outside the client allows to share URIs, but the data can be seen “clear” (plain text) by the gateway operators. If the client wants to share local files, external access to gateway have to be provided by the user (no NAT, or port redirect needed or other options). The proposal for including **Tahoe-LAFS** is to use the CONFINE's testbed as you can see in Figure 17.

The introducer defines a **Tahoe** GRID. Clients and servers are registered in the introducer. Then clients connect to servers and exchange data. The service is a decentralised service, which only relies on the clients and the storage servers. We can install a gateway on the client side to improve the independence of the entire system. Then the relationship is direct between clients (with a gateway) and storage servers. The introducer only participates as a mere hub to get which servers are up and receive the queries of the clients (+gateway) to know to which servers they can connect directly.

It could be possible to include the introducer at the client side (client + introducer + gateway), Then the relationship is direct between the client side and the storage servers, for that we would need to allow connections (to open ports) on the client side to get the connections from the storage servers to register into the introducer at the client side. The **Tahoe-LAFS** runs entirely at the user space, we don't need root access to perform configuration or running services. It gives the
user independence and autonomy to run a GRID by a group of users without relying on third parties. It’s run and owned by community users.

The Tahoe-LAFS project could be used in the project as a distributed file system for storage with encryption by default, it keeps the privacy and it’s easy of use with FUSE, a file system in the user space, using sshfs. It works as a distributed network RAID storage in the background. As all new things, doing tests are required and previous configuration is needed in storage servers, introducer, (perhaps) external gateway to the client, and install the client(s). Clients, gateways (optional), introducer and storage servers could be placed inside CONFINE slivers.

Figure 17: Tahoe-LAFS: Community proposal

3.5. DISTRIBUTED STORAGE - CATS

Although Tahoe-LAFS will provide a file-system service over our community Cloud, there is still a general need for a highly available storage service for applications that run on the Cloud, that is, Storage as a Service (SaaS). Swift in OpenStack could be considered a SaaS, although it is designed to store Big Data, that is, up to petabytes of data. Amazon’s comparable Big Data service is S3. Typically, however, Cloud computing platforms also support a low-latency, highly available, scalable storage service for applications, such as Dynamo at Amazon [21]. Dynamo is a key-value store and is a key service used by thousands of applications on Amazon Web Services. We will adapt and integrate our own scalable storage service CATS, developed at both SICS and KTH, into the COMMUNITY platform [25, 26].
D2.1 System requirements and software architecture

CATS replicates data and distributes it over a large number of machines in a distributed system. CATS provides a simple API for accessing data in terms of a few basic operations, i.e., put(key, value) and value = get(key), and hence it is referred to as a key-value store. The number of replicas accessed by put and get operations determines the level of data consistency provided by the system. To achieve strong data consistency, whereby clients have the illusion of a single storage server, put and get operations need to access overlapping quorums of replicas [19]. Typically, the more servers an operation needs to wait for, the higher its latency will be [20].

Early designs of key-value stores, such as Dynamo and others [21, 22], were targeted at applications that did not require strong data consistency, and driven by the need for low latency and availability, they chose to provide only eventual consistency for put and get operations. Eventual consistency means that for a given key, data values may diverge at different replicas, e.g., as a result of operations accessing less than a quorum of replicas or due to network partitions. Eventually, when the application detects conflicting replicas, it needs to reconcile the conflict. In general however, conflict detection and resolution increases application complexity, both syntactically, by cluttering its logic with extra code paths, and semantically, by requiring programmers to devise reconciliation logic for all potential conflicts.

The strongest level of consistency for put and get operations, called atomic consistency or linearisability, and informally it guarantees that for every key, a get returns the value of the last completed put or the value of a concurrent put, and once a get returns a value, no subsequent get can return an older, stale value. Thus, in spite of failures and concurrency, put and get operations appear to occur in the same sequential order at all clients and every get always returns the value of the most recent put.

When scalable systems grow to a really large number of servers, their management effort increases significantly. Therefore, self-organization and self-healing are commendable properties of modern scalable data stores [23]. CATS relies on consistent hashing [24] for automatically managing data storage and replication responsibilities when servers join and leave the system, or when they fail. Moreover, with consistent hashing all servers are symmetric. No master server means there is no scalability bottleneck and no single point of failure. This makes CATS particularly relevant for Community Networks, as it is entirely self-managing.

Scaling to a very large number of servers also increases the likelihood of network partitions and inaccurate failure suspicions caused by network congestion or by the failure or misconfiguration of network equipment. For the class of critical applications mentioned above, it is imperative to maintain consistency during adverse network conditions, even at the expense of service availability. CATS introduces consistent quorums as an approach to guarantee linearisability in a decentralized, self-organizing, dynamic system spontaneously reconfigured by consistent hashing, and prones to inaccurate failure suspicions and network partitions. With the consistent quorums, CATS provides a scalable distributed key-value store where every data item is an atomic register with linearisable put and get operations and a dynamically reconfigurable replication group.
The increasing spread of elastic Cloud services, together with the need to minimize resource consumption while meeting required QoS requirements, motivates the need of an elasticity controller. The controller automatically resizes an elastic service in response to changes in workload, in order to meet Service Level Objectives (SLOs) at reduced resource consumption. Auto-scaling of the service allows to avoid overprovisioning of resources while meeting SLOs.

The research group at KTH has designed and implemented an initial prototype of ElastMan [41, 42], an elasticity manager for Cloud-based services, such as elastic key-value stores. ElastMan automatically resizes an elastic service in response to changes in workload. To achieve this, ElastMan combines feedforward and feedback control. The feedforward controller of ElastMan monitors the workload and uses a logistic regression model of the service to predict whether the workload will cause the service to violate the SLOs, and acts accordingly. The feedforward controller (Figure 18) is used to quickly respond to sudden large changes (spikes) in the workload. The feedback controller (Figure 19) monitors the service performance and reacts based on the amount of deviation from the desired performance specified in the SLO. The feedback controller is used to correct errors in the model used by the feedforward controller and to handle gradual (e.g., diurnal) changes in workload.

KTH has evaluated the initial prototype of ElastMan using it with the Voldemort key-value store [43] running in a Cloud environment based on OpenStack. Our evaluation has shown that ElastMan can handle gradual workload and quickly respond to rapid workload spikes [41, 42]. The evaluation results show the feasibility and effectiveness of our approach to automation of Cloud service elasticity.

As efficient resource consumption is an important issue in community Clouds, we intend to enhance and adapt the initial prototype of ElastMan and to use it for auto-scaling of elastic services in community Clouds in order to improve the end-user experience with the service at a reduced resource consumption.

---

3 https://elastman.svn.ict.kth.se/elastman/public/ElastMan/
3.7. P2P MEDIA STREAMING - PEERSTREAMER

It is useful for a community Cloud to have a service that lets the users do P2P streaming. We look for an open source project to enable streaming in community Clouds. We discuss a software called PeerStreamer that is a lightweight and customizable P2P Media Streaming framework [15].

PeerStreamer is an open source P2P Media Streaming framework written in C, developed as part of the NAPA-WINE European research project. It includes a streaming engine for the efficient distribution of media streams, a source application for the creation of channels and a player applications to visualize the streams. To the best of our knowledge, PeerStreamer is unique among other similar applications for at least three reasons:

- open source development,
- fast and efficient code,
- network awareness and friendliness.

Its main features include:

- digests almost any media format,
- configurable chunking engine,
- configurable topology module,
- streaming engine with several streaming protocols (push/pull/negotiation based),
- configurable chunk/peer scheduler,
- ALTO (Application-Layer Traffic Optimization) support.

The first test in the CONFINE testbed was the compatibility of PeerStreamer with the Debian images that are currently used in the CONFINE. Before doing these tests, we had to install xfce and xorg in the slivers to be able to launch visual application via SSH. To install PeerStream, we have two options:

1. Installing .deb package in a sliver with Debian: If we try to run the standard player [$peerstreamer] we get the following error:

   SDL_OpenAudio: Text has zero width
   ERROR, COULD NOT INITIALIZE CODECS
   Terminated

   This error seems to happen because Debian does not detect the sound card. In our case, it is a Audio device: Intel Corporation N10/ICH 7 Family High Definition Audio Controller. If we
just want to stream a video from a sliver, we can do it with the following command indicating which interface we want the streamer to use:

$ ./streamer-ml-monl-chunkstream-static -f /demo_video.avi -I pub0

In this case, we want to stream a demo avi file with the public interface of the sliver. If we try this, we get the following error:

Net-helper init : NAT traversal timeout while creating socket
Net-helper init : Retrying without STUN
My network ID is: 10.139.40.61:6666-10.139.40.61:6666
chunkstream connection error: Socket operation on non-socket

2. Compiling the package directly in a sliver: We have seen that the problems we get with the compiled package of PeerStreamer are not related with this software so at the moment we have not tried to compile the package manually.

Following are some of the problems we had:

- sound card driver not found if we want to launch the player in a sliver
- STUN not working
- Connection problem: Socket operation on non-socket

PeerStreamer seems a good software to stream from a sliver. However, we are not able to launch a player in the sliver, but we can start a source of a video using private interfaces. If we try to use public interfaces, we get some network errors that we were not able to solve. They seem related to our software distribution and not related to PeerStreamer. At the moment of this writing we are pending on interactions with the PeerStreamer developers.

Alternative solutions to PeerStreamer include:

  GoalBit is an open source P2PTV platform, capable of distributing high-bandwidth live video content. Its main component, the GoalBit media player, is free software based on the VLC media player and is, therefore, released under the GNU General Public License. It mainly provides the capability of streaming in the P2P GoalBit protocol, a protocol based on BitTorrent.

- **Tribler - Python** [http://www.tribler.org](http://www.tribler.org)
  Tribler is an open source peer-to-peer client with various features for watching videos online. Tribler is based on the BitTorrent protocol and uses an overlay network for content searching, which makes the program operate independent of external websites. Some
D2.1 System requirements and software architecture

Tribler features include: video-only searching, experimental video streaming, and an integrated video player.


### 3.8. P2P VIDEO-ON-DEMAND - GVOD

One of the main challenges in building a video service for community networks is overcoming the bandwidth variability, high packet loss rates, and high network packet latency. The two options for delivering video on community networks are video-on-demand (VoD) and video streaming. A VoD service allows users to watch a video, from a library of available videos, at any point in time. Video streaming, on the other hand, is primarily the live streaming of television. As such, users expect minimal delays in video streaming playback, compared to traditional television.

Currently, video-on-demand (VoD) is a more tractable service than video streaming for community networks, as playback can be delayed until a point when the VoD application is confident that video playback will be able to continue uninterrupted for the duration of the video. Video streaming users, on the other hand, generally assume that a television-like experience — immediate playback with limited lag (playback latency) from the media streaming source. As such, our first research and development efforts at a video service for community networks will concentrate on VoD.

VoD over the Internet is getting more popular every day. The conventional solution to provide this service is the client-server model, which allocates servers and network resources to each client request. However, providing a scalable and robust client-server model, such as Youtube, with more than one billion hits per day, is very expensive. There are few companies, who can afford to provide such an expensive service at large scale.

We will adopt an alternative approach using application level multicast, which utilizes overlay networks to distribute videos to a large number of users. Our service will be built as a Peer-to-Peer (P2P) overlay network in which each node (user) simultaneously functions as both a client and a server. In this model, nodes that have all or part of the requested media can forward it to the requesting nodes. Since each node contributes its own resources, the capacity of the whole system grows when the number of nodes increases. Hence, P2P overlays can provide VoD services at large scale, but with a relatively lower cost for the service provider than that of the client-server model, as network traffic and data storage/processing costs are pushed out to peer nodes.

In VoD service, although a large number of users may be watching the same video, they are asynchronous to each other and different users are watching different portions of the same video at any given moment. In general two main topologies are used for data dissemination in P2P VoD systems: tree-based and mesh-based topologies. In the following, we present these topologies that have been used in the existing VoD systems.
3.8.1. **Tree-based P2P VoD**

In the tree-based topologies data blocks are pushed over a tree-shaped overlay with a media source as the root of the tree. The low latency of data delivery is the main advantage of this approach. Disadvantages, however, include the fragility of the tree structure upon the failure of interior nodes and the fact that all the traffic is only forwarded by them. P2Cast [31] is an example of such systems. In this system users are grouped into sessions based on their arrival time. The users that arrive close in time and within the threshold constitute a session, and build an application-level multicast tree. The source, then, streams the entire video over the tree.

3.8.2. **Mesh-based P2P VoD**

Rajaee et al. have shown in [33] that mesh overlays have consistently better performance than tree-based approaches for scenarios where there is churn, i.e., continuous join and failure of nodes, and packet loss. The mesh structure is highly resilient to node failures, but it is subject to unpredictable latencies due to the frequent exchange of notifications and requests. In this model, neighboring nodes in the mesh overlay exchange their data availability with each other, and the nodes use this information to schedule and pull the required blocks. There are different policies to schedule and pull blocks from other nodes: nodes can pull the missing block with the closest playback time first, or download the rarest blocks first, as in BitTorrent, or a hybrid of these two policies that uses in Give-to-Get [32].

At SICS, we have been developing a P2P VoD application, called Gvod [34], using the Kompics java framework [35]. The system has been developed for the open Internet, and as such there are several assumptions it makes that do not hold for wireless community networks – such as not addressing network locality in the preference function for selecting neighbours for downloading content. We also need to integrate our P2P VoD application with the NAT traversal service being developed from Section 3.9.

Figure 20 shows the Gvod overlay. The system operates as follows. Nodes join the system by requesting the addresses of a small subset of nodes currently active in the system from a bootstrap server, that runs at a well-known address. The joining node then attempts to connect to the returned node addresses to build a neighbourhood consisting of a small number of nodes in the system (called neighbours). Nodes can only exchange video content and control messages with their neighbours.

The neighbourhood relation over the nodes defines the topology of the overlay network. As such, nodes adapt the overlay topology by constantly updating their neighbourhood using a preferential neighbour selection policy. Nodes periodically exchange their set of neighbour addresses with their neighbours and use the preferential neighbour selection policy to determine which nodes to keep in their neighbourhood and which nodes to remove from their neighbourhood, while ensuring that the neighbourhood size is close to, but does not exceed, its maximum allowed size.

The video file is divided into a number of chunks (typically 2MB), which themselves are composed of a number of pieces (typically 16KB). Pieces consist of a number of subpieces (typically 1KB) used for data transfer. A piece is the minimum unit of playable video content, while a sub-piece is the minimum unit of network transfer. Nodes are assumed to have enough available local storage
capacity to store the entire video file. This approach contrasts with BitTorrent, where there the smallest subpiece of a file is a chunk, which is typically at least 512KB. Smaller piece sizes, however, enable faster playback of video, as well as helping to mitigate slow peers, as pieces from slow peers can be re-requested from alternative peers without resulting in excessive wasted bandwidth.

The main research challenges we have in adapting Gvod to wireless community networks include:

- integration in an easy-to-install-use system, e.g., Firefox plugin;
- providing NAT traversal support to ensure that all nodes can contribute to providing the VoD service;
- providing network locality awareness in building our VoD overlay – prefer to connect to nodes that in terms of network metrics (latency, higher bandwidth), than nodes that are close in terms of download position in the video (as is done currently in Gvod);
- providing library functionality for clients storing many videos, with higher priority given to the video currently being watched, while still contributing upload resources to other videos stored locally at clients;
- providing a search service for discovering videos available in the network.

![Diagram of P2P Video-on-Demand application](image)

**Figure 20: P2P Video-on-Demand application**

At SICS, we have developed NAT traversal services, and we have started integrating them with Gvod. We are also working on providing awareness of network zones, similar to Guifi.net network
zones, as a metric for determining the proximity of nodes in the system. We have also started building library functionality. The next section discusses the requirements for our search service.

### 3.9. SEARCH SERVICE

The video-on-demand service is not practical without an ancillary search service that enables users to discover content available in the network. Our main requirements on the search service are that it should:

- be reliable and provide results with low latency;
- be general enough to be used by many different services, not just video-on-demand;
- provide consistent responses to queries;
- be as robust as possible to network partitions;
- not over-consume the limited resources in the CLOMMUNITY Cloud.

The index will be user-generated and user maintained. Users will be able to add new index entries, for example, when he/she makes a new video available in the system. We see our main research challenges as:

- building a reliable, low-latency search service using unreliable nodes and an unreliable network;
- ensuring that the service works for nodes behind NATs;
- ensuring the integrity of the search index at all times, given that users are responsible for updating the index.

#### 3.9.1. INDEX ENTRIES

The following is an example of the structure of a search index entry that could be used by our system:

- **IndexId** (required, not searchable)
- **URL** (required, not searchable)
- **Social network ID of the person who published the index entry** (required, not searchable)
- **Name of file** (required, searchable)
- **File size** (not required, searchable)
- **Date uploaded** (not required, searchable)
- **Language** (not required, searchable)
- **Category of the Content** (video, books, music, games, etc) (required, searchable)
D2.1 System requirements and software architecture

- Free text description of the file, up to a limit of 5000 characters (required, searchable)
- InfoHash of the searchable contents (required, not searchable)
- Availability status for the content (required, searchable)

We should strive to make index entries immutable, as this simplifies protocols for replicating the index among nodes. However, in the above example for an index entry, the availability status for the content would be mutable. We could treat that data as separate from the index entry, and as data that could be discovered at runtime using the service responsible for the content - e.g., the video-on-demand service.

3.9.2. Search Service Architecture

Architecturally, there are two different approaches we could follow to build our search service. We could host the search service and the entire index on COMMUNITY Cloud nodes or we could distributed the search index across potentially all nodes in the wireless community network.

- COMMUNITY-supported search service: A COMMUNITY-supported search service (Figure 21) would require us to use resources at COMMUNITY nodes to provide the search service, to store the search index and to replicate the search index among all nodes in the COMMUNITY cloud. Clients would locally build a model of the expected response time for some of the COMMUNITY nodes, and greedily route search requests to the node with the lowest expected latency. The Cloud-supported search service could be implemented as an asynchronously replicated search service, as high latency links and unreliable connections prevent the use of either strongly consistent replication algorithms or quorum-based replication algorithms. Search queries would have eventual consistency semantics, meaning that, eventually, all index updates will reach all COMMUNITY nodes. However, for a small amount of time, users may not be able to find content until the index entry referring to the content has been replicated to the Clommunity node to which it has sent its query.

- Fully decentralized search service: We will also investigate the idea of building a fully decentralized search service, based on a partitioned index (Figure 22). Nodes would store partitions of the index locally, probably in a local instance of Apache Lucene. Nodes within the same partition would gossip updates to the index with each other. This service would again provide eventual consistency semantics for the search service. Nodes would add an index entry by contacting a leader node within their partition, who would then be responsible for ensuring the consistency of the index entry and that it is safely replicated to enough nodes. When a node searches for an index entry, it needs to send the query to at least one node in all partitions. In practice, nodes will send requests to more than one node per partition and use the result from the first node to respond from each partition. The query will return results to the user either when a node from each partition has returned results or when a timer expires (say 5 seconds). We can constrain users to search within categories of content, significantly reducing the system overhead of a search query, as illustrated below.

We will make our decision about whether to build a COMMUNITY-supported search service or a decentralized search service based on the success of our NAT traversal service. If the NAT traversal service is
D2.1 System requirements and software architecture

reliable and does not introduce excessive control traffic, we will adopt the decentralized architecture. Otherwise, we will build a CLOMMUNITY-supported search service.

![Figure 21: CLOMMUNITY-Supported Search Architecture](image)

Although our Layer-2 VPN, Tinc, enables Cloud services to bypass Network Address Translation (NAT) devices, our project will also deliver Cloud-assisted applications that require end-user devices contribution at the application layer. Cloud-assisted applications, such as VoD (see Section 3.8) will require NAT traversal support for their correct functioning. In general, existing distributed algorithms for constructing overlay networks are not practical over community networks because of the presence of NAT and firewalls. For example, we assume that in our P2P VoD application (Gvod) any pair of nodes can communicate directly with each other, whereas, in reality and in community networks, e.g., Guifi.net, nodes behind NATs (private nodes) do not support direct connectivity with nodes outside their private network. Hence, we need to use NAT traversal.

![Figure 22: Decentralized Search Architecture](image)

3.10. NAT TYPE IDENTIFICATION AND NAT TRAVERSAL
D2.1 System requirements and software architecture

protocols to communicate with private nodes, except in the case where the source and destination nodes both resides behind the same NAT.

Two popular NAT traversal protocols are relaying and hole-punching. Both techniques require a third-party host with a public IP to operate, that is, a node that can be connected to directly by any node, whether behind a NAT or not. As we see in Figure 23, in the Guifi.net, the wireless routers could all be considered to be public nodes, as they have an IP address that is in the network. However, end user nodes are typically behind NAT devices that are connected to the Guifi.net routers. The CLOMMUNITY Cloud will provide services for (i) routing traffic to private nodes, which are behind NATs, and (ii) session establishment (hole-punching or relaying to private nodes).

![Figure 23: CLOMMUNITY NAT Model](image)

Relaying is preferable when the connection setup time should be short (typically less than one second) and small amounts of data will be sent over the connection. On the other hand, hole-punching is preferable when large amounts of traffic will be sent between two nodes (as in CATS and Gvod), and when slow connection setup times are not a problem. However, using hole-punching is not always possible and depends on the NAT devices settings. In order to determine which hole-punching mechanism to use, it is important to know the NAT characteristics of the
source and destination nodes. Hence, a model of NAT behaviour is necessary to enable private nodes to identify what type of NAT they reside behind. Classical Session Traversal Utilities for NAT (STUN) [4] cannot be used to determine all three policies: mapping, allocation, and (iii) filtering [5]. Classical STUN can only accurately determine filtering policy, thus, we use a modified version of STUN protocol described in [3] to determine all three policies.

When determining a node’s NAT type, the main observable behaviour of a NAT is that it maps an IP address/port pair at a private node to a public port on a public interface of the NAT. IP packets sent from the address/port at the private node to a destination outside the NAT are translated by the NAT replacing the packet’s private IP address and port number with the public IP and mapped port on the NAT.

The earliest model of NAT behaviour was STUN that grouped NATs into four groups: full cone, restricted cone, partial cone and symmetric [4]. However, this model is quite crude, and its NAT traversal solutions ignore the fact that when two nodes both reside behind NATs, it is the combination of NAT types that determines the NAT traversal algorithm that should be used. In [5], a richer classification of NAT types is presented, that decomposes a NAT’s behaviour into three main policies:

• Port mapping: This policy decides when to create a new mapping (NAT rule) from a private port to a public port.
• Port assignment: This policy decides which port should be assigned whenever a new mapping rule is created.
• Port filtering: The port filtering policy decides whether incoming packets to a public port mapped on the NAT are forwarded to the mapped port on the private node.

Considering these policies, there are two general techniques that are used to communicate with private nodes: (i) hole punching and (ii) relaying. Hole punching can be used to establish direct connections that traverse the private node’s NAT, and relaying can be used to send a message to a private node via a third party relay node that already has an established connection with the private node. In general, hole punching is preferable when large amounts of traffic will be sent between the two nodes and when slow connection setup times are not a problem. Relaying is preferable when the connection setup time should be short (less than one second) and small amounts of data will be sent over the connection.

• Hole-punching: it enables two nodes to establish a direct connection over intermediary NATs with the help of a third party rendezvous server [7, 8]. Connection reversal is the simplest form of hole punching, which is when a public node attempts to connect to a private node, it contacts the rendezvous server, that, in turn, requests the private node to establish a connection with the public node. Hole punching, however, more commonly refers to how mapping rules are created on NATs for a connection that is not yet established, but soon will be. Simple Hole Punching (SHP) [5] is a NAT traversal algorithm, where both nodes reside behind NATs and both nodes attempt to send packets to mapped ports on their respective NATs with the goal of creating NAT mappings on both sides to allow traffic to flow directly between the two nodes. SHP is feasible when (i) the filtering
D2.1 System requirements and software architecture

policy is Endpoint-Independent (EI) or (ii) the mapping policy is EI or (iii) the mapping policy is stronger than EI and the filtering policy is weaker than Port-Dependent (PD) [5]. The EI policy means that the NAT reuses the same mapping rule for address/port pairs from the same private node, and the PD mentions that the NAT filters all incoming traffic on the public port, except those packets that come from an external node with IP address X and port P that has previously received at least one packet from this public port. Port prediction using contiguity that uses port scanning is another NAT traversal algorithm that can be used when the port assignment policy is Port-Contiguity (PC), where for two consecutive ports mapped on the NAT, e.g., u and v, the NAT binds \( u = v + x \), for some \( x = 1, 2, \ldots \). Similarly, when the NAT maps the port number at the private node to the same port number on the public interface of the NAT, called, Port-Preservation (PP), prediction using port preservation (PRP) can be used [5].

- Relaying: Relaying can be used either where hole punching techniques do not succeed or where hole punching takes too long to complete. In relaying, a third party relay server that has a public IP address keeps an open connection with the private node, and other nodes communicate with the private node.

We are developing three main services on the CLOMMUNITY Cloud as the following:

1. a NAT type identification service,
2. a NAT relaying service, and
3. a NAT hole-punching service.

These services are distributed that will be deployed on CLOMMUNITY nodes, and can be used by Cloud-assisted applications.

3.11. NETWORK-AWARE HTTP PROXY PLUGIN

Internet access is of course the most fundamental service that a community network needs to provide. If the network does not provide general purpose border gateways that can relay arbitrary traffic, the solution is to provide application layer proxy services. The most important of these proxies is the one used for HTTP traffic. These will give the community access to most Internet services. We will in the future refer to the HTTP proxies simply as the proxies, although one might have proxies for other protocols.

The community network Guifi.net provides today a large set of proxies. These are provided by community users on a voluntary basis and are advertised on the community web site. A browser plugin exists that helps the users to select a proxy that is later used by the client browsers. The selection process is done manually and based for instance on the geographical location of the proxies.

The current solution works but has its drawbacks. First of all the selection is done manually and often based on geographical information. Choosing a geographically close proxy hopefully means that this proxy is also network-wise close but there are of course no guarantees. Another
D2.1 System requirements and software architecture

deficiency is that load balancing between proxies needs to be done manually, if at all, by the end users. A end-user would probably only switch proxy if it is unavailable or if performance drops drastically.

In a network that does provide the nodes with public IP addresses, or provide a Network Address Translation gateway (NAT) to the Internet, the need for application layer proxies are not needed from a functional point, but might serve a purpose as caches to improve services. The benefits of introducing caches are of course highly dependent on the network infrastructure, cache sizes and traffic patterns. General purpose web proxies do not dramatically improve performance for the service as a whole; too much content is dynamic content that should not be cached. However, one can improve a limited set of services to the point that is worth the added complexity, especially when low capacity network links.

A proxy service should be completely automatic. The user should not have to manually change the proxy setting of the browser, or at least not have to manually change them as the situation changes. If the network, or the load of the proxy change, the users should migrate to an alternative proxy that can maintain the quality of the service. New proxies added to the network should immediately be offered to the end-users. A more advanced solution could be network aware and also take dynamic load into account.

If proxies are also used for caching the overhead should of course be justified by either increased performance for the end user or, possibly a reduction of the traffic in the network that indirectly improves the performance for the users. The primary focus for proxy services should be community network were general purpose Internet access is either not available or only used by a limited set of users. The proposed solutions should however not be constrained to these scenarios.

We have implemented a Firefox plugin at SICS based on the existing Guifi.net proxy plugin by adding two new features: (i) showing the availability of the proxies, and (ii) presenting a geographically-sorted list of proxies based on the location of the users. In the rest of this section we explain briefly the structure of this plugin, which we call it guifi-addon. We also implemented a server, called guifi-server that collects the information from the installed plugins on end-users desktops.

The guifi-addon provides the following features:

- it shows a Guifi.net icon on the add-on bar,
- upon clicking the icon, it sends a proxy-list request to the guifi-server, and shows the received message as a sorted list of Guifi.net proxies, such that the proxies in the same zone-id of this host are located on the top of the list, and the rest of them at the bottom of the list,
- it also shows the availability of the proxies in the list by assigning colors to them, e.g., green to available proxies, and red to unavailable proxies,
- makes a connection to a selected proxy that enables a connection to the Internet,
- it also checks periodically the availability of the proxies in its zone and reports it to the guifi-server.
D2.1 System requirements and software architecture

When the guifi-server starts, it goes through the following steps:

- it fetches the list of proxies from the Guifi.net site\(^4\),
- it queries zone-ids of the proxies from the CLOMMUNITY mysql database,
- it updates the availability of the proxies based on the reports from the plugins,
- it waits for requests from clients,
- upon receiving a request, it queries the zone-id of the client from the CLOMMUNITY mysql database,
- it sorts the proxy list based on the zone-id of the requestor, and returns the sorted list to the client.

We have used the add-on SDK to implement the guifi-addon. The add-on SDK is an environment that allows users to implement Firefox add-ons easily. To install this environment, the following steps should be done:

- download the tarball package\(^5\),
- extract the tarball, and change directory to it,
- run source bin/activate that changes the environment to the add-on sdk environment.

There are three main commands in this environment:

1. cfx init: this command should be run in an empty directory. It initiates the directory by creating the required files and directories. The main directories are lib and data, and the important files are lib/main.js and package.json. The main body of an add-on is implemented in lib/main.js, and auxiliary files are usually put in data directory. The preferences of an add-on is defined in package.json.

2. cfx run: it executes an add-on, by opening a Firefox browser with an instance of add-on installed on it. When you run this command for the first time, it generates an identifier for the add-on and writes it in the package.json, but does not run the addon. In the first time you should run this command twice to load the addon.

3. cfx xpi: it makes an .xpi addon that can be installed normally on any Firefox browser.

3.11.1. GUIFI.NET ADDON

When the guifi-addon runs on a Firefox browser, it loads the widgets of the addon on the addon panel of Firefox. Figure 24 shows, how the guifi-addon looks like when it loads on a Firefox browser.

\(^{http://guifi.net/guifi.dir/GuifiProxyList.csv}\)
Listing 4 shows the corresponding code of defining a widget. As it shows, the widget is connected to a panel, called comboPanel, and it triggers readProxyList() handler, whenever the icon is clicked. When a user clicks on the Guifi.net icon, the addon sends a request to the guifi-server, and upon receiving a reply, it prints out the message in a multi-line box. From now on we call the code written in lib/main.js and panels, addon script and content script, respectively. The content scripts are usually located in data directory. The addon scripts and content scripts communicate with each other through a port object and two handlers, i.e., “emit” and “on”, that are used to send and receive data, respectively. Figure 25 show how these two script codes (addon script and content script) are connected.

---

https://addons.mozilla.org
When the comboPanel loads in the guifi-addon, it sends a message through the panel port to the content script, and when receives a reply, it triggers the in-line handler. The content of a panel is specified as HTML and you can execute scripts in it. When a user clicks on the Guifi.net icon, the guifi-addon sends a request to the guifi-server, and when receives the proxy list, it shows the list in its panel. The user, then, can choose a proxy from the list.

In addition to make a connection to a proxy server, the guifi-addon also checks the availability of the proxies by sending a GET message to them. If the proxy is available it returns a reply, otherwise a timeout happens. It might also need in the future that we use the guifi-addon to collect statistics about other nodes in the system. We can extend the existing guifi-addon, such that each guifi-addon periodically sends a message to other nodes, and upon receiving the response from that node, measures the latency and bandwidth to that node and reports them to the guifi-server.

### 3.11.2. Guifi.net Server

We used node.js package to implement the guifi-server. Node.js is a server-side software system designed for implementing applications in javascript. Listing 6 shows how to install node.js platform and its mysql module.

The guifi-server (i) first connects to a predefined server and fetches the list of proxies, (ii) then, it connects to mysql database and finds out the zone-id of each proxy, and (iii) upon receiving a request from a client, finds out the zone-id of the client through sending a query to the mysql database, and returns a sorted list of proxies to the client, such that it puts the proxies with the same zone-id of the client zone-id at top, and the rest of the proxies later.
D2.1 System requirements and software architecture

1 amir@rakhsh:~$ sudo apt-get install software-properties-common
2 amir@rakhsh:~$ sudo apt-get install python-software-properties python g++ make
3 amir@rakhsh:~$ sudo add-apt-repository ppa:chris-lea/node.js
4 amir@rakhsh:~$ sudo apt-get update
5 amir@rakhsh:~$ sudo apt-get install nodejs npm
6 amir@rakhsh:~$ sudo npm install mysql@2.0.0-alpha7

Listing 5: Installing node.js package
4. Proposed Software Architecture

As the DoW states, in WP2 a software architecture will be defined, whose initial version will be reported in D2.2. For our current work regarding requirements and software evaluation, however, we needed to define an initial scheme of the software architecture. Based on this step, the results of our work related to the building of services will allow us to obtain a better understanding of the system in a form of a software architecture.

We define a distributed software architecture based on (i) the research on the software services requirements of the project, (ii) the provided community-lab testbed, and (iii) the experience at Guifi.net. This software architecture is being used in the community software distribution and in the end-users' provided services. Later in this section, we describe the point of view of users and administrators.

We want to build a Cloud system to provide distributed Cloud services to community networks. In the following, we present an abstract model of the elements in different layers of the distributed software architecture. From a top-down approach we have the following architectural elements:

- User UI and Applications: These elements provide the end-users with applications and user interfaces to interact in a transparent way with the underlying community services.

- Community Services: These services provided by the community software distribution. Location of services, and distributed storage are examples of these services.

- Community Software Distribution: The software distribution that runs over the underlying layers, e.g., the containers or virtual machines. This provides a way to pack and distribute the project services in a single package.

- The Containers and Virtual Machines: The elements of this layer can be containers, virtual machines or similar elements. They are created inside the Cloud management platform that provides an environment to install the community software distribution.

- Cloud Management Platform: This is the base operating system that executes inside a community box. It allows to create containers and virtual machines.

- Community Box: It connects to the community network and part of the Cloud management platform is installed on it.

- Community Network: This is the network infrastructure, e.g., a community network.

Table 7 and Figure 26 show visually the elements described in this architecture and represent the relation of each layer with the presented software services explained in Chapter 3.
D2.1 System requirements and software architecture

Table 7: Distributed Software Architecture.

<table>
<thead>
<tr>
<th>Architectural element</th>
<th>Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>User UI and Applications (Gvod, Http Proxy, other applications to be defined)</td>
<td></td>
</tr>
<tr>
<td>Community Services (Avahi, Tahoe, CATS, NAT, Search Service, ElastMan, other applications to be defined)</td>
<td>Software</td>
</tr>
<tr>
<td>Community Software Distribution (Guifi-Community-Distro)</td>
<td></td>
</tr>
<tr>
<td>Containers/Virtual Machines (LXC)</td>
<td></td>
</tr>
<tr>
<td>Cloud Management Platform (CONFINE, OpenStack)</td>
<td></td>
</tr>
<tr>
<td>Community Box</td>
<td>Hardware</td>
</tr>
<tr>
<td>Community Network</td>
<td>Network</td>
</tr>
</tbody>
</table>

For the Services inside the software architecture we take into account that all are designed to maintain the data privacy of the user [36-37]. A proposal about the implementation of the
software architecture is available at [16] at the Figure “Community ecosystem with connections and devices”.

4.1. USER PERSPECTIVE

From the user point of view, the system is seen as a Cloud of services with features such as:

- The user can create a node to join the Cloud. The node installs software components of the community Cloud.
- Location of Services: The user can publish new services and locate running services. The user gets a list of published services in the Cloud.
- Distributed Storage: The user has a directory where to put data and the user can share that data with other users. The data is distributed using various storage servers. In a specific use case, that directory is accessed by the user and the other users in different standard ways: HTTP, FTP, and FUSE (with this, the user can see a Cloud directory as a local directory).

4.2. ADMINISTRATOR PERSPECTIVE

From the administrator, developer or the advanced user point of view:

- Hardware and software of the node has to be installed and maintained.
- The node connects to the community network.
- The access the community network, the user has to follow the community network rules.
- The node contains the Cloud management platform with its tools, and containers or virtual machines.
- Part of the community software distribution is installed in containers or virtual machines. The containers and virtual machines can be managed with the provided default tools for each kind of element.
- The community software distribution manages the services and user interfaces offered by the community Cloud. The software distribution can be updated, upgraded and managed.
5. Conclusions

As more and more everyday services migrate to the Cloud, it is important that community networks can provide many of the services expected by Internet users. In this document, we described the requirements for our community Cloud platform, identified and evaluated existing software tools and services, their suitability for inclusion in the platform, and finally, we proposed an architecture for the system with our preferred services. Our goal is to provide a community Cloud platform that provides both the network and platform services required for community Cloud applications, and some example Cloud-assisted applications of interest to community network users, such as video-on-demand (VoD).

The community Cloud platform builds on our prior work in the CONFINE project, in community networks such as Guifi.net, and in developing self-managing distributed services, such as VoD. In general, network and application services defined for the Internet cannot be simply reused on the COMMUNITY project. This is typically because traditional services assume the availability of static network endpoints to bootstrap or provide services. Given this, we have evaluated a number of experimental network services designed for decentralized community networks, such as auto-managed DNS, and we have also proposed a number of new Cloud-assisted services that will be adapted for community networks, such as storage, NAT traversal, and VoD services. Some services will be provided on our community Cloud platform, such as storage services, e.g., TahoeFS and CATS, while others will be provided as Cloud-assisted applications, such as peer-to-peer VoD, which uses search and NAT traversal services that we are developing.

We have also contributed an extensive analysis of existing Cloud management platforms for their inclusion in the community Cloud platform. While systems such as OpenNebula and Cloudstack show promise, it appears that both OpenStack and the CONFINE controller are the Cloud management systems that currently best suit the needs of community networks. OpenStack appears to be a good choice for deployments that require virtual machine migration and where clusters have relatively stable network connections. CONFINE with LXC, however, offers a more decentralized platform that can be deployed on more resource-constrained hardware.

Finally, we have proposed an architecture for our community Cloud platform, containing an initial set of services. We believe that this platform, when developed and deployed on community networks, such as Guifi.Net, will provide both useful end-user services and offer new possibilities in developing Cloud applications for community networks.
6. References


[3] PlanetLab, an open platform for developing, deploying, and accessing planetary-scale services: https://www.planet-lab.org/


[38] http://confine-project.eu, [online: 25 June 2013]


D2.1 System requirements and software architecture


D2.1 System requirements and software architecture

Licence

The CLOMUNITY project, June 2013, CLOMUNITY-201306-D2.1:

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 Unported License