**Project Acronym:** CLOMMUNITY  
**Project Full Title:** A Community networking Cloud in a box.  
**Type of contract:** STREP  
**contract No:** 317879  
**Project URL:** [http://clommunity-project.eu/](http://clommunity-project.eu/)

<table>
<thead>
<tr>
<th>Editor:</th>
<th>Jim Dowling (SICS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverable nature:</td>
<td>R, P</td>
</tr>
<tr>
<td>Dissemination level:</td>
<td>Public (PU)</td>
</tr>
<tr>
<td>Contractual Delivery Date:</td>
<td>30 June 2015</td>
</tr>
<tr>
<td>Actual Delivery Date:</td>
<td>30 June 2015</td>
</tr>
<tr>
<td>Suggested Readers:</td>
<td>Project partners</td>
</tr>
<tr>
<td>Number of pages:</td>
<td>119</td>
</tr>
<tr>
<td>Keywords:</td>
<td>WP2, community clouds, software distribution, cloud services</td>
</tr>
<tr>
<td>Authors:</td>
<td>Jim Dowling (SICS), Lars Kroll (KTH, SICS), Alexandru - Adrian Ormenisan (KTH, SICS), Roger Baig (Guifi.net), Pau Escrich (Guifi.net), Miquel Martos (Guifi.net), Agust'ı Moll (Guifi.net), Roger Pueyo (Guifi.net), Amin Khan (UPC), Jorge Florit (UPC), Felix Freitag (UPC), Dani Marti (UPC), Roc Meseguer (UPC), Leandro Navarro (UPC), Mennan Selimi (UPC), Miguel Valero (UPC), Davide Vega (UPC), Vladimir Vlassov (KTH), Hooman Peiro Sajjad (KTH), Paris Carbone (KTH)</td>
</tr>
<tr>
<td>Peer review:</td>
<td>Leandro Navarro (UPC)</td>
</tr>
</tbody>
</table>
Abstract

This document presents the software architecture for the CLOMMUNITY project. The software architecture in CLOMMUNITY is centered on a Linux distribution, called Cloudy, that was developed in the project. Cloudy provides custom support for wireless community networks through decentralized services and a cloud-computing platform. Our work on Cloudy contains two main activities. In the first activity, we evaluated existing software for inclusion in Cloudy. We carried out our evaluations on our testbed, and in many cases, we needed to make modifications to existing software to provide the necessary support for decentralized operation. In our second activity, we developed a number of fully-decentralized platform services and application services that are integrated on nodes running the Cloudy distribution. One of the main lessons we have learnt in building Cloudy is that low-level infrastructural services from existing open-source software platforms need non-trivial modifications and configuration to operate reliably in wireless community networks, and that there is a large benefit to developing our own platform services and applications, as they can easily be customized to the challenging network environment.
Executive Summary

This document describes the software architecture of the CLOMMUNITY project and its realization in the Cloudy Linux distribution. We also define the system requirements that guided our design of both Cloudy and the software services that were developed and tested during the CLOMMUNITY project. *Cloudy* is a Linux distribution containing open-source software services for our cloud platform for wireless community networks. Cloudy also integrates platform services that were developed in the project, including decentralized storage (a key-value store), video-on-demand, and search services. Although our platform services were designed and developed primarily for wireless community networks, they also can be used on the open Internet.

The software we provide in this deliverable spans a number of layers in the stack, from the network to the operating system all the way up to applications. At the bottom layer, is the wireless community network. On top of this, Cloudy builds on the Debian Linux distribution, in which we support infrastructural cloud-services (virtualization, Linux containers) and decentralized network services (Avahi and TincVPN). Layered over the cloud management platform, we have evaluated cloud federation platforms. In addition to these operating system level services, we have developed our own platform services that support cloud-enabled applications, with support for decentralized storage and NAT-traversal. At the highest user-facing layer, we have developed application services that act as demonstrators for the CLOMMUNITY cloud platform, including video-on-demand and a decentralized search service. All of these services are integrated in the Cloudy stack.

This deliverable describes the software components developed, evaluated, and adapted for Cloudy. It contrasts with other activities in CLOMMUNITY, such as the research performed when developing services (such as storage and video-on-demand) in WP3, and the testing and performance measurement of services on our testbed in WP4. This document should be read as a follow-on to D2.2, where we evaluated many other software components that are now included in the CLOMMUNITY software architecture. The latest version of our software packages can be found at: [http://wiki.cloommunity-project.eu/soft:sourcecode](http://wiki.cloommunity-project.eu/soft:sourcecode).
# Contents

1. **Introduction**  
   1.1. CLOMUNITY Architecture .................................................. 12  
   1.2. Relationship to other CLOMUNITY deliverables ....................... 14  
   1.3. Cloudy software, documentation, and web sites .................... 14  

2. **Architecture**  
   2.1. User Perspective .......................................................... 16  
   2.2. System Architecture ..................................................... 16  
   2.3. Cloudy Distribution Architecture .................................... 17  

3. **Infrastructural Services**  
   3.1. Proxmox Virtual Environment .............................................. 19  
   3.1.1. Features ............................................................................ 19  
   3.1.2. Single installation .......................................................... 21  
   3.1.3. Clustering ......................................................................... 21  
   3.2. Debian on Intel Galileo boards ............................................ 25  
   3.3. Federation with OpenStack Clouds ....................................... 25  

4. **Platform Services**  
   4.1. CaracalDB ................................................................. 28  
   4.1.1. Design ............................................................................. 29  
   4.1.2. Architecture .................................................................... 35  
   4.1.3. Deployment Guide .......................................................... 36  
   4.1.4. User Guide ...................................................................... 38  
   4.2. CaracalDB Data Model ......................................................... 39  
   4.2.1. Introduction ..................................................................... 39  
   4.2.2. Concepts ........................................................................... 39  
   4.2.3. Data model operations ..................................................... 40  
   4.2.4. Architecture ..................................................................... 40  
   4.2.5. Deployment Guide .......................................................... 41  
   4.2.6. User guide ........................................................................ 41  
   4.3. Tahoe-LAFS ......................................................................... 47  
   4.3.1. Introduction ..................................................................... 47  
   4.3.2. Tahoe-LAFS architecture in CONFINE testbed .................... 47  
   4.3.3. Tahoe-LAFS integration in Cloudy .................................... 48  
   4.4. NatTraverser ......................................................................... 49  
   4.4.1. NAT-aware address format .............................................. 51  

5. **Application Services** ...................................................... 53  
   5.1. Video-on-Demand (Gvod) ..................................................... 53  
   5.1.1. Existing video on demand (VoD) systems .......................... 53
5.1.2. Gvod .......................... 54
5.1.3. Gvod client features .......... 54
5.1.4. Gvod architecture ............ 55
5.2. Decentralized Search .......... 57
   5.2.1. Decentralized Search Service 58
   5.2.2. Search Architecture .......... 59
   5.2.3. Design of the utility function 60
   5.2.4. Leader Selection ............. 60
   5.2.5. Index Manager Component ...... 61
   5.2.6. Format of an index entry ...... 61
   5.2.7. Adding index entries ........ 62
   5.2.8. Threat Model ................. 63

6. Cloudy: CLOMMUNITY GNU/Linux distribution 66
   6.1. Motivation for a community network (CN) cloud distribution . 66
   6.2. Community network cloud environment ................ 66
   6.3. Distribution requirements .......... 68
   6.4. Implementation ................. 68
      6.4.1. Architecture ............. 69
      6.4.2. Services in Cloudy .......... 70
      6.4.3. Web-based management interface ...... 70
      6.4.4. Plugin-based modular architecture ........ 71
      6.4.5. Distributed Announcement and Discovery of Services ....... 73
      6.4.6. Clouder: Docker with Cloudy container .......... 76
      6.4.7. Build tools ................ 77
   6.5. Installation process .......... 78
      6.5.1. Image based ............. 78
      6.5.2. Virtualized ............... 79
   6.6. Integration of new services ........ 80
      6.6.1. Pastecat ................ 80
      6.6.2. Syncthing ................ 88
      6.6.3. CaracalDB and the REST API ...... 89
      6.6.4. Gvod ................ 91
      6.6.5. Sweep ................ 91

7. Software Development and Quality 92
   7.1. Kompics Testing & Simulations ........ 92
   7.2. Software Services ............. 93
   7.3. Cloudy ................ 94
      7.3.1. Cloudy releases policy ...... 94
      7.3.2. Applications integrated into Cloudy ....... 95
      7.3.3. Third-party components ........ 95

8. Conclusions 96

A. System Requirements 97
   A.1. Cloud Management Platform .... 97
   A.2. Network Services ............. 97

Deliverable D2.3
# List of Figures

1.1. Cloudy GNU/Linux distribution architecture .............................................. 13  
1.2. Cloudy Web site. ..................................................................................... 14  
1.3. Decentrify Web site. ............................................................................... 15  
1.4. Cloudy Demo site. ................................................................................... 15  
2.1. CLOMMUNITY from the users' perspective. .............................................. 16  
2.2. CLOMMUNITY System Architecture. ....................................................... 17  
2.3. Cloudy Distribution Architecture. ............................................................ 17  
2.4. Cloudy Environment. ............................................................................... 18  
3.1. Proxmox VE web management interface. ............................................... 20  
3.2. Proxmox VE single machine diagram. ..................................................... 21  
3.3. Proxmox VE local cluster diagram. ......................................................... 22  
3.4. Proxmox VE cluster distributed nodes map. ........................................... 23  
3.5. A Shared backend between two identity services. .................................... 26  
3.6. Synchronous multi-master replication between the backends of two identity services. .......................................................... 26  
3.7. Flat-Network Topology between Clouds in Guifi-net. .............................. 27  
4.1. CaracalDB logical view. ............................................................................ 30  
4.2. CaracalDB server architecture. ................................................................. 35  
4.3. Data Model architecture. ....................................................................... 41  
4.4. Tahoe-LAFS deployed in the Community-Lab testbed ............................ 48  
4.5. Screenshot of the Cloudy web interface .................................................. 48  
4.6. Partial screenshot of the Cloudy web interface showing the form for joining a Tahoe-LAFS grid with a storage node. ......................... 49  
4.7. Nat-traversal middleware as a set of components .................................... 50  
5.1. AngularJS web-ui with VideoJS as video player ................................. 55  
5.2. Gvod Internal Architecture .................................................................... 56  
5.3. Cloud-assisted Vod and cloud-assisted Nat Traversal using nodes running Gvod and NatTraverser on the CLOMMUNITY cloud ................................. 57  
5.4. Decentralized “large fanout” search. The search index is sharded into Categories, and each Category can be divided up into a number of Partitions. .................. 58  
5.5. Decentralized Search Architecture. ......................................................... 59  
5.6. Modified 2-Phase Commit Protocol. LG is the LeaderGroup, “ids missing by the leader” refers to any index entries the leader may have missed, for example during a network partition. ......................................................... 62  
6.1. Community network cloud environment ................................................ 67  
6.2. Cloudy GNU/Linux distribution architecture ......................................... 69  
6.3. Cloudy management interface system options ....................................... 72  
6.4. Cloudy management interface multilingual support ............................. 73
6.5. Cloud services menu in the Cloudy web interface ........................................ 73
6.6. Guifi services menu in the Cloudy web interface .......................................... 74
6.7. Cloudy geTinconf configuration .................................................................. 75
6.8. Search services menu in the Cloudy web interface ........................................ 76
6.9. Cloudy services discovery (web interface) ..................................................... 76
6.10. Cloudy services discovery (command-line interface) ................................. 77
6.11. Clouder Boot of menu install ................................................................. 77
6.12. Cloudy image boot menu ........................................................................ 79
6.13. Cloudy console after boot ................................................................. 79
6.14. The web interface of Pastecat is light and simple, yet functional. ............... 81
6.15. Syncthing installation process in Cloudy. .............................................. 88
6.16. Syncthing’s web interface showing the shared repositories and the list of Synthing nodes that are exchanging data (this screen capture shows a single demo repository). 89
6.17. CaracalDB integration in Cloudy. ........................................................... 90
6.18. CaracalDB REST API integration in Cloudy. ......................................... 90
6.19. GV oD integration in Cloudy. ............................................................. 90
6.20. Sweep integration in Cloudy. .............................................................. 91

7.1. Example component architecture of Kompics in simulation (Source [1]) .... 93

C.1. Eucalyptus platform architecture .................................................. 103
C.2. Components of Eucalyptus architecture ........................................... 103
C.3. Eucalyptus platform federated with Amazon Web services .................... 104
C.4. Overview of Scalr cloud management ............................................... 105
C.5. Scalr architecture and the different components .................................. 106
C.6. ECmanaged Dashboard ........................................................................ 107
C.7. Add additional cloud providers to ECmanaged. ................................... 107
C.8. Select from given templates. ........................................................... 107
C.9. Add customized template ............................................................... 108
List of Tables

6.1. Contents of Cloudy, the CLOMMUNITY GNU/Linux distribution . . . . . . . . 71

C.1. Some of the important projects in the Cloud federation domain . . . . . . . . 111
C.2. Supported Cloud computing platforms by the Cloud federation projects . . . . 111
C.3. The Cloud federation requirements addressed by the projects . . . . . . . . 111
1. Introduction

CLOMMUNITY is designing and developing a platform to enable cloud-computing on wireless community networks. To this end, CLOMMUNITY is developing a new distribution of Linux containing customized support for decentralization, called Cloudy, along with a number of decentralized platform services and decentralized applications for Cloudy. Decentralization is a key requirement for wireless community networks. In this deliverable, we present our work on:

- the infrastructural services we have evaluated for inclusion in Cloudy, including federation solutions,
- the decentralized platform and applications services developed for wireless community networks,
- and the integration of these services in the Cloudy distribution.

1.1. CLOMMUNITY Architecture

CLOMMUNITY is built on the Cloudy Linux distribution, illustrated in figure 1.1. Cloudy consists of a number of new services for Linux, designed to help build cloud-computing services for decentralized, community networks. Cloudy’s main components can be considered a layered stack with services residing both inside the kernel and higher up and user-level.

All of the software included in our platform is open-source. Many of the components have been developed during the course of this project, and many are open-source software services that we have customized for the Cloudy distribution. The services currently included in the architecture are based on the requirements from D2.1, our research from WP3, our experiments on the software on the provided community-lab testbed, and our experience on using the software in production at Guifi.net. An earlier version of our software architecture is defined in D2.2.

The software developed for the CLOMMUNITY architecture can be grouped into one of three levels: software relating to the construction and management of the Cloudy distribution, platform services to enable and facilitate application development, and some (software-as-a-service) applications to demonstrate the functionality of the CLOMMUNITY platform.

As a Linux distribution, Cloudy lies at the lowest layer of the stack, providing custom decentralized network services for service discovery and service announcement. Service discovery and announcement are crucial building blocks for enabling distributed services to be orchestrated to provide platform and application services. Cloudy has adapted the TincVPN service to provide Layer 2 connectivity between CLOMMUNITY boxes. Layer 2 connectivity is needed between cloud boxes, as they may reside on different administrative domains and even be located behind firewalls. Cloudy also includes a customized version of Avahi to provide decentralized service discovery at Layer 2, again needed to discover other services that will be used to provide higher-level services. Cloudy also includes a number tools to ease the building and packaging of our Debian-based distribution, as well as a user-interface to discover and use platform services available in a CLOMMUNITY network. For inclusion in Cloudy, we have also evaluated and deployed platforms in our testbed for resource allocation based on virtualization (OpenStack, OpenNebula, Proxmox VE and Eucalyptus).
1. Introduction

1.1. CLOMMUNITY Architecture

At the platform level, we have tested and evaluated a secure, decentralized file-system, TahoeFS, and we have also built two platform services to support application development on CLOMMUNITY. The first platform service is a NAT-traversal service that provides a fully-decentralized service that enables network connectivity between nodes, even if one or both of the nodes reside behind a Network Address Translation gateway (NAT) gateway. The second service is a decentralized key-value store, with a data model that closely resembles the Java Persistence API (JPA) model, familiar to many Java programmers.

Finally, we developed a number of decentralized applications to demonstrate the potential for the CLOMMUNITY platform. The most challenging application to build is our peer-to-peer (P2P) VoD system, Gvod, that uses the CLOMMUNITY cloud to help improve quality-of-service and provide bootstrapping for the system. Gvod uses the NAT-traversal service to enable connectivity between nodes residing behind NATs. In parallel, we have been developing a fully decentralized search service, based on Apache Lucene. This is a generic search service, where application instances act as both clients and servers for the search service. That is, clients collectively manage the search index, as well as being the users of the search index. The index is stored, partitioned, at all participating nodes in the system and can be searched in parallel. We have also implemented a network-aware plugin for Firefox that helps Guifi.net users improve their experience of browsing the Internet, by suggesting the current HTTP proxy in the system that will have the “best” performance. We model “best” as
a combination of reasonably low latency access and reasonably high bandwidth to the Internet. Our network-aware plugin was described in D2.1 and is not further discussed in this deliverable.

1.2. Relationship to other CLOMMUNITY deliverables

Like any living software platform, CLOMMUNITY and Cloudy are constantly undergoing evolution and refinement. This deliverable D2.3 builds upon earlier versions of the software architecture introduced in D2.2, which, in turn, is built on requirements from D2.1. The work of WP2 has been formed by experimental results from WP4, T4.1 which studied the cases for community clouds and provided feedback from experiments on platform services and applications. Research on scalable service overlays and the network service and architecture, including T3.1 and T3.4, has informed the development of the platform services and applications defined in this deliverable. The software systems developed in WP2 have been tested and evaluated in WP4, providing feedback to improve the software. WP4 contributed by making the community cloud testbed operational (T4.2) and allowing to evaluate the suitability and performance of different services and applications (T4.3).

1.3. Cloudy software, documentation, and web sites

The software developed in CLOMMUNITY is available as open-source, and the different packages and their repositories are listed in section B.

Much of the documentation in this deliverable is also published on the wiki of the project [2]. The CLOMMUNITY wiki is a tool to support the dissemination of intermediate progress and results of our work within the community.

The Cloudy web site\(^1\) has been created to describes different aspects of the Cloudy distribution (Figure 1.2).

\(^1\)http://cloudy.community

![Figure 1.2: Cloudy Web site.](http://cloudy.community)
The Decentrify web site\textsuperscript{2} integrates specifically to Sweep, CarcalDB and GVod tools (Figure 1.3).

![Decentrify Web site](http://www.decentrify.io/)

Figure 1.3: Decentrify Web site.

The Cloudy demo instance\textsuperscript{3} offers access to try out Cloudy at any time with the user:password guest:guest (Figure 1.4).

![Cloudy Demo site](http://demo.cloudy.community)

Figure 1.4: Cloudy Demo site.

\textsuperscript{2}http://www.decentrify.io/

\textsuperscript{3}http://demo.cloudy.community
2. Architecture

This chapter introduces the overall system architecture of the CLOMMUNITY project. This architecture is the result of research, evaluation, and experience with the Clommunity platform over the course of the project. The chapter also discusses the different layers of infrastructure and platform, over services to applications and how they are inter-connected. Further it categorises software into layers.

2.1. User Perspective

![Figure 2.1: CLOMMUNITY from the users’ perspective.]

From the user point of view, the system is seen as a collection of services that can be discovered via a Service Browser web interface, as exemplified in figure 2.1. On the one hand a user can chose to use services that are already provided. Those include video-on-demand provided by GVoD which is described in section 5.1, distributed search provided by Sweep, see section 5.2, and distributed file system provided by Tahoe-LAFS (section 4.3).

On the other hand a user can use the Cloudy web interface (see section 6.4.3) to provide services on their CLOMMUNITY box for others. This could for example be CaracalDB instances (section 4.1) or copies of the Sweep index.

2.2. System Architecture

The overall system architecture of CLOMMUNITY can be seen in figure 2.2. It is divided into three major layers: 1) Infrastructure and Platform Services (orange/yellow), 2) Clommunity Services (green) 3) Application Services (blue). The very bottom of the stack is provided by hardware like the
physical Clommunity network, the Clommunity boxes or servers that run OpenStack and are part of CONFINE, for example. On top of all of that runs the CLOMMUNITY distribution Cloudy, which is described in detail in chapter 6. In order to allow for service discovery via Avahi, Cloudy instances are connected via a level 2 over level 3 network provided by Tinc. Other services, like Sweep, GVoD, Tahoe-LAFS, and so on, can then be selected and run as Docker containers.

2.3. Cloudy Distribution Architecture

---

Figure 2.2: CLOMMUNITY System Architecture.

---

Figure 2.3: Cloudy Distribution Architecture.
At the core of CLMMUNITY is the Cloudy Unix distribution which is described in detail in chapter 6. Cloudy can be run on bare metal or on virtualized hardware, as depicted in figure 2.3. Instances are connected via the virtual L2 over L3 network which provides the overlay to interconnect all the servers in a micro cloud. On top of the overlay network service announcement and discovery processes are built, that respectively publish local information to the cloud and receive remote data from its neighbour peers.

The system can be administered via either the web interface or the console. The first one provides to end users a tool for that, covering the most basic options (installation, self-configuration and control of the services).

The console is the traditional approach to the system management, providing all the available options, left for advanced users. As shown in figure 2.4 Cloudy can be run in a variety of environments, from low end computers like low-power PCs or ARM based boards, via commodity hardware, all the way to high performance systems in data centers. This gives Cloudy deployments the flexibility necessary to deal with the variety of conditions that could be found in a community network.
3. Infrastructural Services

This chapter introduces a number of cloud-management (or infrastructural) services that were deployed on our target hardware and network environment. The Cloud management platform can be considered a layer over the network and computational resources. The Cloud management platform provides an interface for the underlying network that allows the deployment of other services over the abstraction layer. In D2.1, we evaluated a number of open source Cloud management platforms: OpenNebula, OpenStack, CloudStack, and CONFINE. In this chapter, we continue our work on evaluating existing cloud management platforms, discussing first Proxmox.

3.1. Proxmox Virtual Environment

Proxmox VE is a complete open source virtualization management solution for servers. It is based on KVM virtualization and container-based virtualization (OpenVZ), and manages virtual machines, storage, virtualized networks, and HA Clustering.

3.1.1. Features

• **Virtualization technologies.** Proxmox VE, as commented above, uses basically KVM and container-based virtualization technologies.

  - **Kernel-based Virtual Machine (KVM).** KVM\(^1\) is an Open source hypervisor providing a full virtualization solution for Linux, added as a kernel module, on x86 hardware containing virtualization extensions (Intel VT or AMD-V).

    With KVM it is possible to run multiple virtual machines to provide nimbly and easily different OS images (Linux, Unix, Windows, etc.) to the cloud infrastructure.

  - **Container-based virtualization.** OpenVZ\(^2\) is container-based virtualization for Linux. OpenVZ allows to create multiple secure, isolated Linux containers on a single physical server enabling better server utilization and ensuring that applications do not conflict.

• **Live Migration.** It moves the running virtual machines and containers from one physical host to another without any downtime.\(^3\)

• **Bridged Networking.** Proxmox VE uses a bridged networking model. All VMs can share one bridge as if virtual network cables from each guest were all plugged into the same switch. For connecting VMs to the outside world, bridges are attached to physical network cards assigned a TCP/IP configuration. For further flexibility, VLANs (IEEE 802.1q) and network bonding/aggregation are possible.

• **Central Management.** Proxmox VE has a Web-GUI that gives an overview of all the KVM guests and Linux containers and of the whole cluster (figure 3.1).

---

\(^1\)http://www.linux-kvm.org/

\(^2\)http://openvz.org/

\(^3\)CT Live Migration Video Tutorial: http://youtu.be/AkimEEHArqg
3.1. Proxmox Virtual Environment

It uses the unique Proxmox Cluster file system (pmxcfs)\(^4\), which is a database-driven file system for storing configuration files. This enables to store the configuration of thousands of virtual machines, and by using corosync\(^5\), these files are replicated in real time on all cluster nodes.

- **Administration.** Using role based user and permission management, it is possible to define granular access for all objects, like VMs, storage, nodes, etc. Proxmox VE supports multiple authentication sources like Microsoft Active Directory, LDAP, Linux PAM standard authentication or the built-in Proxmox VE authentication server.

- **Backup and Restore.** A backup tool (vzdump) is integrated that creates consistent snapshots of running OpenVZ VEs and KVM guests, by creating an archive of the VM or CT data and also including the VM/CT configuration files.

- **Flexible Storage.** The Proxmox VE has a flexible storage model in which the VMs images can be stored on one or several local storages or on shared storage; supporting all storage technologies available for Debian Linux, like LVM Group, iSCSI target, NFS Share, etc.

- **Proxmox VE High Availability Cluster.** Proxmox VE HA Cluster enables the definition of high available virtual servers. If a virtual machine or container (VM or CT) is configured as HA and the physical host fails, the VM is automatically restarted on one of the remaining Proxmox VE Cluster nodes.

![Proxmox VE web management interface.

\(^4\)http://pve.proxmox.com/wiki/Proxmox_Cluster_file_system_(pmxcfs)

\(^5\)http://corosync.github.io/corosync/
3.1.2. Single installation

We tested Proxmox VE first on a single machine to take advantage of the resources for using them more efficiently. This system provided us a support for a fast deployment of virtual machines and made much easier to manage the different OS installed on them instead of being installed in different physical machines.

This machine with Proxmox VE was used for deploying some services both to the community and to help researchers.

One of the first VMs deployed was a Debian 7 OS with quite low resources available (see figure 3.2), to help the development of a Firefox add-on for the Guifi.net proxies. Also it was decided to give a Proxy service to the Guifi community by installing a new VM with more resources, especially in regard to storage with several GB to save the logs.

![Figure 3.2: Proxmox VE single machine diagram.](image)

We have been testing VM functionalities and features during several months using KVM technologies.

3.1.3. Clustering

For extending the clouds, we took advantage of the Proxmox VE clustering system\(^6\).

Given the success with a single Proxmox VE installation, we decided to create a cluster by adding several machines. In order to have an easy and more efficient management, we set them up in a Proxmox cluster, to manage the hosts through the Proxmox VE web management interface (figure 3.3).

The Cluster uses a database-driven file system for storing configuration files and it is replicated in real time on all nodes by the corosync software\(^7\). It allows to manage any node through the web management interface of any of the nodes in the cluster; i.e. a unified management with a distributed configuration.

---

\(^6\)http://pve.proxmox.com/wiki/Proxmox_VE_2.0_Cluster

\(^7\)http://corosync.github.io/corosync/
3.1. Proxmox Virtual Environment

3.1.3.1. Local

First, we installed Proxmox VE in another machine with the same specifications, and we configured a Proxmox VE cluster between the two machines.

To allow a simple cluster with two machines, a special parameters must be added in the config file (/etc/pve/cluster.conf) to avoid some issues\(^8\). It should be done by creating a new file and applying the changes through the web interface in order to apply the changes in the ProxmoxVE cluster filesystem\(^9\).

Apart from identifying the two nodes cluster, we set the expected votes to 1 because the clustering system used by Proxmox VE expects by default one vote per node, and if one node goes down, the cluster may be get blocked and/or not working.

**Listing 3.1:** Changes in config to create a two-node cluster.

```xml
<cmans two_node="1" expected_votes="1"/>
```

Progressively, we have been adding more machines to support the deployment of different experiments and services. Now we have seven machines in this cluster for that purposes (see listing 3.2).

**Listing 3.2:** UPC cluster configuration file.

```xml
<?xml version="1.0"?>
<cluster name="cloud-upc" config_version="8">
  <cmans keyfile="/var/lib/pve-cluster/corosync.authkey"/>
  <clusternodes>
    <clusternode name="cloud2" votes="1" nodeid="1"/>
    <clusternode name="cloud6" votes="1" nodeid="2"/>
    <clusternode name="cloud-nas" votes="1" nodeid="3"/>
    <clusternode name="cloud8" votes="1" nodeid="4"/>
    <clusternode name="cloud5" votes="1" nodeid="5"/>
    <clusternode name="cloud7" votes="1" nodeid="6"/>
  </clusternodes>
</cluster>
```

---

\(^8\)http://pve.proxmox.com/wiki/Proxmox_VE_2.0_Cluster#Two_nodes_cluster_and_quorum_issues
\(^9\)http://pve.proxmox.com/wiki/Fencing#General_HowTo_for_editing_the_cluster.conf
3. Infrastructural Services

3.1. Proxmox Virtual Environment

We have the local Proxmox VE cluster in production for more than half a year. From a system administration perspective and functionality, our conclusions about using Proxmox VE are very positive. We think that it is a VM managing platform which could be applied for the management of community clouds, where an easy setup and usage of a cloud management platform is needed.

3.1.3.2. Distributed

Apart from providing services and support in a local cloud, we have set up a testbed with Proxmox VE in different machines distributed around the Guifi network (see figure 3.4).

The objective of this cloud (distributed cluster) is to provide a testbed similar to the real scenario, where the cloud nodes are spread all over the community network and not only connected in a local area network.

The main use of this testbed is for deploying and testing the Cloudy distribution (previously known as Guifi Community Distro/GCODIS), explained in chapter 6.

In spite of that all machines are connected to the Guifi network, we set up a Tinc VPN\(^\text{10}\) tunnel to do a backbone connection between them.

![Figure 3.4: Proxmox VE cluster distributed nodes map.](http://www.tinc-vpn.org)
Proxmox VE uses Corosync Cluster Engine\textsuperscript{11} to perform and to manage the clusters, which is configured by default to use IP multicast to communicate between nodes. As commented above, in this scenario we set the cluster through a Tinc VPN, but we had issues with synchronization between nodes, which might be related to the multicast over the VPN. For that reason, we set temporary the cluster to unicast\textsuperscript{12} to allow the communication between nodes and avoid sync issues. To make this change (see listing 3.3), we have to follow the instructions to edit the cluster\textsuperscript{9}.

Listing 3.3: Configuration for cluster with unicast.

\begin{verbatim}
<cluster name="clomm-cloud" config_version="12">
  <cman keyfile="/var/lib/pve-cluster/corosync.authkey" expected_votes="1" transport="udpu"/>
  <clusternodes>
    <clusternode name="cloud-hangar" votes="1" nodeid="1"/>
    <clusternode name="cloud-taradell" votes="1" nodeid="2"/>
    <clusternode name="cloud-upc" votes="1" nodeid="3"/>
    <clusternode name="cloud-ictp-1" votes="1" nodeid="4"/>
    <clusternode name="cloud-ictp-2" votes="1" nodeid="5"/>
    <clusternode name="cloud-inesc" votes="1" nodeid="6"/>
  </clusternodes>
</cluster>
\end{verbatim}

The cluster configuration and sync issues do not affect the running of the VMs, since these are managed by their virtualization technologies (KVM or OpenVZ) and they are independent of the Corosync Cluster Engine. It means, the virtual machines are running correctly and are accessible through the Guifi community network independently of if the nodes do synchronize properly through the cluster. We need, however, to investigate more on what it is the best configuration to get the best performance in this specific scenario.

3.1.3.3. Current usage

In the CLOMMUNITY cloud testbed, we have several Proxmox VE installations deployed providing IaaS to perform research experiments. Deliverable D4.2 reports on the testbed we have deployed and deliverable D4.3 describes the experiments which have so far been supported.

\begin{itemize}
\item[\textsuperscript{11}] \url{http://corosync.github.io/corosync/}
\item[\textsuperscript{12}] \url{http://pve.proxmox.com/wiki/Multicast_notes#Use_unicast_instead_of_multicast}
\end{itemize}
3.2. Debian on Intel Galileo boards

Community cloud infrastructures are envisioned to include home gateways, but could even reach IoT platforms. We report here on an exploratory study we performed for extending the CLOMMUNITY testbed (reported in deliverable D4.2) to enable experiments with distributed file systems on a heterogeneous community clouds.

The Galileo board ships with a custom Yocto\(^{13}\) GNU/Linux system prepared at Intel. The base image also comes with a tweaked kernel and some needed kernel modules to get everything working properly.

In order to run our experiments, we needed to be able to install Tahoe-LAFS on a Galileo board. Since Debian stable - currently codenamed Wheezy - packages it, we ported a base Debian install onto the Galileo. There is no official support for Debian on the Galileo from Intel, so we followed some advice we found on the Internet.

Specifically, the steps we followed were largely inspired by a thread\(^{14}\) from the Intel communities forum. That post describes how to create a base Debian debootstrap image, and set it up properly for it to work on Galileo. Most steps are directly taken from that guide, but we made a few changes and additions.

For instance, when installing the root filesystem image on the SD, we created a 512MB swap partition and grew the ext3 rootfs to fill the rest. This is mainly in case we needed a bit of extra memory to run the experiments, or perhaps some more space to install packages and libraries from Debian. After all, 256MB just meets the recommended RAM capacity to run Debian without a graphical interface - depending on what we want to do, we may run out of memory easily.

The forum thread mentions some SSH daemon problems and crashes. We didn’t look into that, since we just connected to the board directly via a serial connection. Most probably, that issue still stands and would be something to solve if we wanted to access these machines remotely.

One problem we did experience is the slowness of Debian running on a class 10 SD card. APT is not prepared to run on extremely low-end devices, such as the Galileo with low RAM and an SD as the main disk, so running apt-get or apt-cache meant long waits of up to a few minutes for them to read or write to the package database. Rather than a problem, this is just something that could be improved in the future. But for now, Debian packages work and that is what we needed to run our tests.

This last problem mentioned is probably why Intel went with Yocto, a customizable GNU/Linux distro for embedded devices, for their Galileo boards. This is not to say that generic distros like Debian can’t run on it, but distros targeted at embedded devices are a better fit for the board.

3.3. Federation solution for OpenStack-based Cloud deployment in CLOMMUNITY

Federation and interoperation of multiple clouds are currently not considered in the design of the OpenStack Cloud computing platform. In this section, we describe our approach to provide some of the federation features in the UPC-KTH OpenStack community Cloud testbed explained in D4.2.

**Federated Identity Management:** Keystone is the identity management service in OpenStack and is built of a group of four internal services: identity, token, catalog and policy. Each of these services can

\(^{13}\)https://www.yoctoproject.org/  
\(^{14}\)https://communities.intel.com/thread/48074
be configured to use a different backend. The backend in each service is defined in keystone.conf file. A way to reach federated identity between multiple OpenStack Clouds is to provide a shared backend between the identity services of participant Clouds while every Cloud has its own backend for the token, catalog and policy services. The possible backends for the identity service are SQL based backend (using SQLAlchemy), PAM (Pluggable Authentication Module) and LDAP. SQL based backend can be used for all of the four services. However, Keystone’s implementation is limited to one backend of each backend type. This means that we cannot have two of the SQL backend type. This also implies that if we use SQL backend for the identity service we cannot use this backend for the other services because we want to keep them separated. To bypass this limitation, we use SQL backend for the identity service, templated backend for the catalog service, KVS backend for the token and rules for the policy. We set up Mysql for the identity as the shared backend between Keystones and the rest are a separate backend per Keystone. This is shown in Figure 3.5. To increase Keystone’s read operations performance, we are planning to provide a cluster of Mysql servers in which data are synchronously replicated across the databases. As it can be seen in Figure 3.6. We will use a synchronous multi-master replication plug-in called Galera Replication. I

**Shared Image Repository:** Image store of OpenStack is called Glance and it is composed of two services: api and registry. The api service accepts Image API calls such as image storage and retrieval and the registry service is for the management of Image metadata. Each of the two services has their own backend. Metadata is stored in a SQL based database which is accessible by the registry service and the actual image files are stored in a different storage repository such as Swift and normal filesystem which is accessible by the api service. In our testbed, we provide a shared image repository by sharing the backend between the Glance Image stores. We use Swift as the backend for the actual image files and Mysql server for the metadata storage.

**Cross-cloud Networking:** Networking in Openstack is mainly handled by Neutron, a highly-
configurable service that provides SDN capabilities. Users can create and manage virtual networking resources such as private networks, virtual routers, floating-ip pools, dhcp servers etc. through API calls or through the graphical web user interface exposed on the dashboard (Horizon). This allows for flexible multi-tier architectures that can be reconfigured on-the-fly upon demand. Furthermore, VMs in OpenStack can be exposed to a public network through the allocation of floating IPs taken from a pre-configured pool. In our approach to offer cross-site network federation between VMs in different clouds we used Guifi.net as the common flat network between different clouds. Through the Guifi.net node management service we sharded the IP space by explicitly allocating subnets to each cloud. Thus, each IP pool is locally managed by the respective Neutron service and possible conflicts are therefore avoided. All Guifi.net IPs are statically assigned to VMs upon their creation which makes them always accessible through any node in the same network. Underlying access to Guifi.net from remote sites such as the KTH campus was achieved through layer-2 GRE tunnelling. In Figure 3.7 we show an example of how the flat network topology is visualised in the Openstack Dashboard. Each cloud maintains a pool of external IPs “ext-net” that is associated with a respective Guifi.net slice. For example the UPC Cloud is assigned with 10.90.228.0/27 and the KTH one with 10.93.0.0/24. Traffic to VMs residing in each private network is being handled by a virtual router as displayed in the same Figure.

Figure 3.7: Flat-Network Topology between Clouds in Guifi-net.
4. Platform Services

This chapter introduces the platform services that are being developed for the CLOMMUNITY software architecture. These services can be used by community network applications to provide such basic functions as highly available storage, a distributed filesystem, or NAT Traversal support. In the context of cloud-computing, they are called platform-as-a-service, with the additional feature that these services can be provisioned automatically. In our case, we provide our services built-in to the Cloudy distribution, so automated provisioning is not a requirement of our platform services, yet.

4.1. CaracalDB

In the networked world of today basically all services are data driven in one way or another, giving rise to a necessity to store and access large amounts of data in a reliable manner. And as such, distributed storage systems have become widespread in basically any large scale service deployment. This is true for the industry as well as research organisations and community networks.

The largest of these systems are likely being employed by web giants like Google, Amazon, Facebook and Apple. But research organisations like CERN produce and store tens of petabytes per year, as well. And not only has all this data to be stored somewhere, it also needs to be located, accessed, processed and maybe even modified.

All these requirements imply the necessity for storage solutions that provide massive scalability, working on single rack installations just as well as on globally distributed multi-datacenter deployments with thousands of servers. This also implies that system management should be federated in some way to avoid bottlenecks or lack of availability. Furthermore, on these scales failures, be they in software or in hardware, cease to be an exception and become the rule. Even if any given component has a minuscule probability of error, the sum of all the components permanently operating for years at a time makes failures a given. And thus they need to be considered during planning of solutions and handled in implementations.

In addition to all the inherent challenges to distributed large scale storage, there is also a concern about the kind of data being stored and the way it is accessed. A typical concept here is called big data. And despite there being no clear definition as to what data actually qualifies, the idea to be considered is that big data is so huge it’s challenging to handle with the available resources or naïve solutions. When it comes to access, workloads are typically being categorised as online transaction processing (OLTP) for transaction based data entry and retrieval, and online analytical processing (OLAP) for analytical loads. These concepts are mostly used for relational database systems, though, and modern applications require more flexible solutions for processing. This trend is well exemplified by the MapReduce [3] pattern which was originally implemented by Google but today is the base for many open source systems, like Apache’s Hadoop framework. The important consideration for these processing frameworks is that they deal with large datasets and long running tasks, where placement decisions for operations in relation to the data they are working on are crucial for feasibility. This is opposed to a service that only queries in the kilobyte range, but needs to minimise the latency of such requests, possibly because a (human) user is waiting for the reply and humans are not generally known for their patience. Additionally, streaming applications that require medium sized chunks of
data in a specific order are common today. Consider Spotify or YouTube as examples. Either of these access patterns might deal with huge amounts of data, but a storage solution would be hard pressed to cater efficiently to all of them while treating their data undifferentiated.

It is clear from the elaborations above that the development of flexible, scalable and reliable distributed storage solutions is a necessity. Of course there is a large number of products already available today. Among them are relational database management systems (RDBMSs) like MySQL, which may be deployed in a distributed way with replication and sharded or as MySQL Cluster. Their limiting factor is usually scalability, and companies have to expend great effort to make their large-scale deployment feasible, as Facebook has done [4], for example.

In the NoSQL world there are systems like Amazon’s Dynamo [5], Apache’s Cassandra [6], Google’s BigTable [7] or Riak. These systems scale very well and provide a range of options when it comes to consistency, availability and partition tolerance (CAP) trade-offs [8]. Generally, they don’t provide transaction support like classical RDBMSs, though.

When it comes to the storage of large data blobs there are Amazon’s simple storage service (S3), the Google file system (GFS) [9], Apache’s Hadoop distributed file system (HDFS) [10] or the approach of Nightingale et al. [11]. These systems are basically distributed filesystems, intended for batch processing applications. Yet, projects like Apache’s Hive [12] or Cloudera’s Impala [13] strive to provide ad-hoc services on top of these distributed filesystems.

However, what is very rare is a system that provides support for small low-latency queries as well as long running batch processing tasks or streaming applications. One example of something similar is Yahoo!’s Walnut [14] system, which aims to provide a combined storage layer for PNUTS [15], a key-value store for small data under 100kB, MObStor, an object store for unstructured medium sized data, and HDFS [10]. Walnut does not provide RDBMS-like transactional operations with atomicity, consistency, isolation and durability properties (ACID), though.

On the front of HDFS there is another problem which concerns metadata storage. In HDFS metadata is stored at a special node (or a group of nodes) called namenode. While work has been done to improve reliability of the namenode, like adding fail-over nodes, the actual problem is limited storage space of a single node, consequently limiting the capacity of the whole Hadoop cluster.

From these examples follows the need for a system that addresses their shortcomings, providing ACID low-latency access to small data, and (quasi-)unlimited storage capacity for big data as well as support for flexible processing solutions. And at the core of these requirements is a scalable and fast consistent key-value store with transaction support.

We present CaracalDB, a distributed storage framework that provides solutions for bootstrapping, lookup, and load balancing, while remaining flexible on the choice of replication algorithm and underlying local storage.

4.1.1. Design

Concepts

CaracalDB is basically a key-value store offering put and get operations, but the exact semantics of operations it can support depend mostly on the choice of replication algorithm and local storage. For example, with Paxos state machine replication (SMR) CaracalDB could easily provide a compare-and-swap (CAS) operation.

CaracalDB has the concept of a schema, which is similar to a collection in Walnut [14]. A schema
Figure 4.1: CaracalDB logical view.

contains policies for the replication algorithm, the size of the replication group, the quorum size in that group, and the underlying storage implementation. It is likely that it will be extended in the future to also include information about the structure of values.

For every schema there exist one or more partitions in the system which have a maximum size of data they are responsible for. We call this limit the capacity of the partition and the exact value is configurable for the system. Realistic values are probably between 100MB and and 1GB, but there are no inherent assumptions on this. A partition is uniquely identified by an id from the key space (s. below). In CaracalDB partitions are the unit of replication.

Each partition is mapped to exactly one replication set that has the correct number of nodes for the partition’s schema in it. Multiple partitions can be assigned to the same replication set. We call the nodes hosting copies of a partition its replication group.

Key Space

CaracalDB is technically not a distributed hash table (DHT) since it does not hash its keys. However, node ids and entry keys are still drawn from the same key space (s. below), making it very similar. Also its key space is not circular as in systems like Chord [16] or CATS [17, 18, 1, 19], but linear. It is defined herein:

Let $B$ be the set of byte values. For ease of use we shall represent them in hexadecimal notation, hence $B = \{00, 01, \ldots, FF\}$. A finite sequence over $B$ is a function $s : N \to B$ where $N \subseteq \mathbb{N}$ finite, such that either $1 \in N$ or $N = \emptyset$ and $N$ forms a ring under modulo-$|N|$ arithmetic. Or in other words
4. Platform Services

4.1. CaracalDB

$N$ is a gap-free prefix of $\mathbb{N}$. The set of all such sequences we shall call $K$ and this is the key space of CaracalDB.

It can be shown that there exists an injective map $\kappa : K \rightarrow \mathbb{R}$ [20] and it remains to define the ordering relation. So let $<_{K} \subset K \times K$ be a relation such that for all $x, y \in K$ the expression $x <_{K} y \iff \kappa(x) < \kappa(y)$ holds. Since $\kappa$ is injective and $<$ is a total ordering on $\mathbb{R}$, $<_{K}$ imposes a total ordering on $K$. Due to the construction of $\kappa$ in [20] $<_{K}$ forms a prefix ordering over the byte sequences in $K$ with infimum of $K$ being $s_{0} : \varnothing \rightarrow \{\varnothing\}$ and the supremum of $K$ is given by $s_{\infty} : \mathbb{N} \rightarrow \{FF\}$. For the remainder of this document whenever two keys in $K$ are compared with $<, >, \leq, \geq, \min$ or $\max$ the usage of $<_{K}$ is implied.

**Virtual Nodes**

CaracalDB employs the concept of virtual nodes (vnodes), which are globally uniquely addressable entities that share a network port on a certain physical node, which we shall henceforth call a host to avoid confusion between the terms. A vnode is assigned a key $k \in K$ upon creation, which will not be changed over the lifetime of the vnode. Both hosts and vnodes are addressable over the network, with hosts having an address of the form $(ip, port, \bot) = (ip, port)$ while vnodes’ addresses are $(ip, port, k)$. Please note that, while $k$ will not usually be globally unique, the triples $(ip, port, \bot)$ and $(ip, port, k)$ will be.

In Kompics terms a vnode represents a subtree of the full component tree, connected to the common network port via a channel filtered on the vnode’s id. It may also share other components from the host, but generally those connections won’t be filtered but either be a one-way communication from vnode to host, or utilise a request-response-pattern which keeps track of the exact path an event travelled.

In CaracalDB a partition $p$ with key $k_{p} \in K$ of schema $s$ with replication degree $deg_{s}$ will be assigned to at least $deg_{s}$ vnodes on different hosts with addresses of the form $(ip, port, k_{p})$. This assignment makes all the vnodes with $k_{p}$ members of $p$, call them $m_{p}$ and the hosts of $m_{p}$ the replication group of $p$.

There are three main reasons for the usage of vnodes in CaracalDB. Firstly, we can start a different set of components on a per-vnode basis. This is the reason that CaracalDB can support different replication algorithms and operations per collection.

Secondly, vnodes allow us to distribute the load better over the available hosts in the system and also give us a unit that we can move around in the cluster to change the current load distribution.

And lastly, we can use clever vnodes assignments to replication groups to improve the re-replication time in case of host failures. This is an idea that was exploited to an extreme in [11], but it also applies on our coarser partitioned case. To exemplify the problem, we consider a system where each host stores a single dataset which is replicated at two other nodes. Let hosts $a, b, c$ form a replication group and consider what happens when $a$ dies. Since the data that was on $a$ needs to be re-replicated to preserve the replication degree a new node $d$ is commissioned into the replication group ($d$ could have been a standby node, or carry data in another overlay, it makes little difference). Since the data $d$ needs is only available at $b, c$ it needs to be copied from there making the disk speeds of $b, c, d$ a re-replication performance bottleneck. In this model it could take hours to re-replicate large amounts of data, leaving the system in a state where a single failure of $b$ or $c$ could leave it unable to take decisions on that group or a failure of both hosts could even result in data loss. During all this time the other hosts in the system are sitting idle in terms of re-replication of the data from $a$. They might...
of course be running normal operations or re-replication from another failure.

As opposed to this consider a system where data is partitioned and partitions are mapped to a number of different replication groups. If there are \( p \) partitions all with replication degree \( \text{deg} \) over \( h \) hosts that means there will be \( \text{avg}_p = \frac{\text{deg}}{h} \) vnodes on average per host, if the system is in a well balanced state. Now consider if a single host \( a \) fails. If \( p \) is large enough there is a good chance that every other host in the system has some replication group in common with \( a \). If this system re-replicates by assigning the failed vnode for every partition to a random other host that doesn’t already have a replica of the data, for example, the result will be that re-replication is done using all available disks in the cluster. Every host will most likely be transferring data to and from a number of different hosts. This will be much faster than the previous solution. Of course, it also means that normal operations for all partitions in the cluster are affected. Yet, we argue that it is preferable to have a latency increase on all operations in the cluster for a relatively short time, than to have a latency increase in only a small subset but for a very long time with the added higher possibility for data loss.

### Data Partitioning

As explained CaracalDB provides schemas and splits their data into partitions. However, we have not yet shown how the full key-space \( \mathcal{K} \) is split up into the partitions. To see this, first of all note that from a key space perspective a schema is nothing more but a prefix on all keys in all partitions of the schema. This prefix is at the same time the schema id, so if we want a schema for users we could simply give it the id \((75, 73, 65, 72, 73) \in \mathcal{K} \) (ASCII for “users”). This is certainly a valid schema id, however, one should be careful that different schemas don’t have ids that are prefixes of each other. A safer solution would be to hash schema names to obtain their ids, resulting in equal length ids, which clearly can’t be prefixes of each other (ignoring unlikely hash-collisions). CaracalDB makes no assumptions on how schema ids are created, though, and developers are free to choose them by whatever means seem appropriate.

It was already mentioned that partitions are also assigned an id from \( \mathcal{K} \) and now it is clear that for any schema \( s \) with id \( k_s \in \mathcal{K} \) and partition \( p \) of \( s \) with key \( k_p \in \mathcal{K} \) \( k_s \) must be a prefix of \( k_p \). In fact there will always be exactly one partition with \( k_s = k_p \), so that the layout of schemas on the key space is clearly delimited and no vnode can end up being responsible for data from more than one schema. We call this property alignment, named after the way structs are laid out in memory by languages like C and C++.

Let \( \mathcal{P} \subset \mathcal{K} \) be the set of all partition ids in the system at any particular point in time. \( \mathcal{P} \) is still totally ordered by restricting \( \prec_\mathcal{K} \) to \( \mathcal{P} \). With this in mind let \( \text{succ} : \mathcal{P} \to \mathcal{P} \cup \{ s_\infty \} \) be the function that maps each \( k \in \mathcal{P} \) to the smallest value \( k' \in \mathcal{P} \) with \( k < k' \) or to \( s_\infty \) if \( k \) is the maximum of \( \mathcal{P} \).

We then define the responsibility \( R_p \) of a partition \( p \) with key \( k_p \in \mathcal{P} \) as \( R_p \subseteq \mathcal{K} \), \( R_p = [k_p, \text{succ}(k_p)) \).

This is opposed to the way Chord and CATS handle this where a node with id \( n \) is responsible for the range \( (\text{pred}(n), n] \) [16, 17, 18, 1, 19]. The reason for our decision to diverge from the well established pattern, is that in our linear, lexicographically sorted key space it is more intuitive if a partition is responsible for any key of which its own id is a prefix, up to the id of its successor. Furthermore, this system allows us to have the very intuitive alignment property presented above.

### Locating Data

CaracalDB is keeping a full view of the system membership and responsibilities at every node in the system and could even distribute this view to clients outside the storage system, as Nightingale et al. are doing for example [11]. Such a system can guarantee \( O(1) \) routing and is aimed at reducing
the latency of operations by avoiding as many network trips as possible. Depending on how current
this global view is kept at different locations, such a system can do single-hop lookup, by sending the
request directly from a client to the responsible system node. It could also implement two-hop lookup,
by sending the request from client to a random node in the system and forwarding from there to the
responsible node. If views can be outdated at the system’s nodes some bounded multi-hop lookup can
be implemented, whereby the client sends to a random node in the system and that node forwards to
the node its current view of the system (which might be outdated) suggests is responsible and so on,
until the an up-to-date view is found and the right node located.

Clearly the space requirements for such a full view increase linearly with the number of nodes in the
system, making it very difficult to scale to very large cluster sizes. Despite this fact, CaracalDB uses
exactly such a full view system with possibly outdated views which can optionally be installed at
clients. We simply want to provide the fastest possible steady state lookup, even if we might pay for
this during high-churn episodes. In this sense CaracalDB provides all three full view lookup versions
described above. It provides single-hop lookup if the system is in a steady state and a client has a
recent version of the global lookup table (LUT) (s. below) installed. If the client is thin we provide
two-hop lookup in the steady state. And if the system is undergoing churn CaracalDB provides multi-
hop lookup with a configurable bound $\epsilon$ (s. [20]).

The Lookup Table

The LUT is a data structure that maintains the current view of the CaracalDB system. It provides four
types of information:

1. A mapping from four byte integer ids, call them host ids for short, to host addresses.
2. A mapping from four byte integer ids, call them replication group ids, to lists of host ids and an
eight byte version number.
3. An ordered mapping from partition ids ($\in K$) to replication group ids.
4. A mapping from schema ids ($\in K$) to associated schema information (s. 4.1.1).

The first two are not a major size problem. Even for large systems with $n = 10000$ nodes, (1) will be
around 120kB. For (2) we assume an average replication group size of three and also $3n$ replication
groups and get 600kB. Note that systems like [11] use $n^2$ replication groups for this table, but their
load distribution is dependent on the size of the table, while ours is not or at least to a lesser degree.
Thus, 3 seems to be an amply large factor, which can of course be adapted if necessary. The larger
this factor, the more even will be the distribution of vnodes over hosts and the larger the diversity of
replication groups per host. Furthermore, (4) will likely not be an issue since it only grows in the
number of schemas, which is probably relatively small even for a large system. Clearly the major
issues are with (3). Hence [20] presents some ideas on limiting the size of the LUT which are not, yet
implemented.

Managing the LUT

In order to make a global LUT viable and also to allow for a bound $\epsilon$ on the multi-hop lookup scheme
described above there should be an always current canonical version of the LUT maintained some-
where in the system. CaracalDB uses a special partition for this, called the zero partition and depicted
as $0_P$. The range $R_0$ of $0_P$ is $[0_K, (00, 00, 00, 01))$ and this range is also reserved in CaracalDB for
maintenance information, in a similar fashion to the usage of directories in [21]. Additionally, $0_P$ is
always assigned to the replication group with id 0, which always contains exactly three hosts. The
replication protocol for $0_P$ is Paxos SMR.
In order to avoid superfluous changes in the LUT, which might involve expensive operations such as
data transfer, only the member of $0_P$ that currently considers itself the leader (using the same leader
detection component as the Paxos implementation) proposes new changes to the LUT. However, during
periods of churn or network partitions, the Paxos algorithm still guarantees orderly transitions
from one LUT version to the next, even if they might be based on false suspicions. Additionally, to
ensure that hosts and vnodes at least have current information about themselves (i.e. their assigned
partitions, their current view, their current range), for every decision that changes the LUT a sec-
ondary Paxos instance is run, which includes all hosts that are affected by the change (even the ones
suspected to have failed) in addition to the members of $0_P$ and requires a majority quorum to succeed.
To ensure that this majority can be achieved a proposed change should not include more than one host
that is suspected to have failed.
For all non-affected hosts the changes to the LUT can be either simply broadcasted or gossiped.
Nodes should take care, however, to apply changes always from one LUT version to the next without
skipping. If a node finds that an update is missing in between its current state and an update that it got,
it can request the missing update(s) from one of the members of $0_P$ by reading from the appropriate
position in the reserved range.

Replication

While CaracalDB is not limited to a specific replication algorithm, it does provide Paxos SMR which
provides atomic consistency. It enforces a total order on all operations in a partition and supports
cross-partition range queries.
For custom algorithms it is required that they implement a reconfiguration protocol that includes
decisions about when and how data transfer has to take place. CaracalDB provides a component that
can handle data transfer and report its status on request, but replication algorithms need to decide
when to initialise data transfer in relation to the time they get a reconfiguration message, and also
how to handle normal operations during this syncing phase.
Furthermore, replication components are directly connected to a storage component and need to issue
the required commands to it in response to operation requests that are routed to them. Whether or not
these commands are issues before or after replication depends on the algorithm.

Storage

Just as replication, storage implementations are pluggable. However, they are not handled on a per-
vnode basis, but on a per-host basis. The reason for this is the circumstance that some databases are
optimised to minimise seeks on hard disk drives (HDDs) [22] and having one of those for each vnode
would defeat that purpose. For the same reason one should be careful not to locate two different
databases on the same HDD. For solid state disks (SSDs) this problem is not as pronounced.
CaracalDB is designed with the following storage implementations provided, but really any engine
that provides operations for put, get, iterate and batch write can be used.

InMemory provides non persistent, fast, but also very limited storage capabilities. Since the size of
RAM is limited and the speed is so much superior to network, the actual implementation of this
engine does not make a huge difference. It needs to be lexicographically sorted, though, ruling
out hash maps, for example. CaracalDB currently uses Java’s standard TreeMap implementa-
tion.
4. Platform Services

4.1. CaracalDB

LevelDB [23] is an engine based on log structured merge trees. Riak [24] for example has a persistence layer using LevelDB.

4.1.2. Architecture

Server

Figure 4.2 shows a logical view of the distribution of CaracalDB server components over host-level and vnode-level. Failure detection is shared on a host-level to avoid sending pointless duplicate heartbeats to hosts that are watched by multiple vnodes. The components identified with “replication” and “persistence” are place-holders for the component tree of an actual implementation for these services. Paxos SMR, for example, consists of an execution engine, a replicated log and a leader election component.

Client

CaracalDB only provides a ‘thin’ client at this point, and hence provides at best two-hop routing. The client consists of a number of Kompics components that act as workers and handle one operation at a time. Communication to a blocking Java interface is done via a blocking-queue for easy synchronisation. A client manager class provides factory services for new workers. During start-up every worker may ask the bootstrap server for a number of sample hosts from the system to avoid sending all requests through the same initial host. This is purely optional, however, as this information is also gathered over time by inspecting the source addresses of operation responses.
### Listing 4.1: CaracalDB Server Start Script.

```bash
#!/bin/bash
LFJ=\{path to log4j.properties file\}
APPC=\{path to application.conf\}
SIG_LIBS=\{path to sigar native libs\}
nohup java -Dlog4j.configuration=file://$LFJ -Dconfig.file=$APPC -Djava.library.path=$SIG_LIBS -jar caracaldb-server.jar &> nohup.out &
eval 'echo $!' >> caracal.pid
```

### Listing 4.2: CaracalDB Server Stop Script.

```bash
#!/bin/bash
if [ ! -f "caracal.pid" ]; then
    for PID in $(cat caracal.pid)
    do
        kill $PID
    done
    rm caracal.pid
fi
```

### 4.1.3. Deployment Guide

#### 4.1.3.1. Server Nodes

A CaracalDB server node deployment consists of the following files:

- A configuration file “application.conf”. See the “reference.conf” in the source repository (or the packaged .jar) for configuration options.
- A folder with Sigar native binaries. It is enough to have the binary for your specific system, but having a folder with all of them is easier if the deployment environment is heterogeneous.
- A Log4J properties file.
- Simple start and stop scripts similar to listings 4.1 and 4.2.

It is important to note, that one of the deployed nodes must be designated as bootstrap node in all the config files. It is advantageous if that were to be the most stable host in the system.

In addition to the files described above, it can also be helpful to have a deployment script like listing 4.3 to manage a cluster, if no other more sophisticated mechanism exists.

### REST API Service

A REST API service for CaracalDB is provided as a separate application that is written in Scala and based on CaracalDB’s Java client code and the Spray web-server. It can be deployed as any client node on its own host or on the same hosts as actual server nodes. The interface is JSON based and, of course, needs open ports for use with the HTTP protocol (SSL is not supported at this point). Forwarding through nginx is an option if the API is combined with an HTML UI front-end app and...
Listing 4.3: CaracalDB Deployment Script.

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "Usage: deploy | boot | shutdown"
    exit
fi

USER={SSH user existing on all target hosts}
HOST_FILE={a file with IP addresses of target hosts one per line}
REMOTE_PATH={path to copy the files to}
LOCAL_PATH={path to copy the files from}/*
BOOTSTRAP={IP of the bootstrap node}
SERVERS=$(cat "$HOST_FILE")

howmany() { echo $#; }

function deploy {
  parallel-ssh -h "${HOST_FILE}" -l ${USER} mkdir -p ${REMOTE_PATH};
  parallel-scp -h "${HOST_FILE}" -l ${USER} -r ${LOCAL_PATH} ${REMOTE_PATH}
  for s in $SERVERS; do
    NUMS=$(howmany $SERVERS)
    ssh ${USER}@${s} "cd ${REMOTE_PATH}; chmod +x *.sh; echo server.
    server_hostname=${s} >> application.conf; echo bootstrap_address.hostname=${s} >> application.conf; echo caracal.bootThreshold=${NUMS} >> application.conf"
    echo "Uploaded to ${USER}@${s}:"
  done
}

function boot {
  parallel-ssh -h "${HOST_FILE}" -l ${USER} "cd ${REMOTE_PATH}; rm -f *
    simulation.log; rm -f nohup.out; ./caracal_start.sh"
  for s in $SERVERS; do
    echo "Started CaracalDB on ${USER}@${s}:"
  done
}

function shutdown {
  parallel-ssh -h "${HOST_FILE}" -l ${USER} "cd ${REMOTE_PATH}; ./killall.sh"
  for s in $SERVERS; do
    echo "Killed all the things on ${USER}@${s}:"
  done
}

case $1 in
  deploy) deploy
  ;;
  boot) boot
  ;;
  shutdown) shutdown
  ;;
esac
```
4.1. CaracalDB

4.1. Platform Services

needs to avoid cross-site scripting protection problems.

A CaracalDB REST API deployment consists of the following files:

- A fat executable “caracaldb-rest-api.jar” with dependencies.
  Download: http://cloud7.sics.se/caracal/caracaldb-rest-api.jar
- A configuration file “application.conf”. See the “reference.conf” in the `rest-api` source repository (or the packaged .jar) for configuration options.
- A Log4J properties file (preferably in XML format).
- Start and stop scripts similar to those in section 4.1.3.1.

Client

CaracalDB client code is not stand-alone but must be deployed as part of another application. The Maven [25] dependencies for this are “se.sics.caracaldb.caracaldb-core-0.0.3-SNAPSHOT” and “se.sics.caracaldb.caracaldb-client-0.0.3-SNAPSHOT” which can be found in the Kompics Maven repository at http://kompics.sics.se/maven/snapshotrepository.

4.1.4. User Guide

REST API

CaracalDB’s REST API provides the following services:

**PUT/GET/DELETE** at `/schema/{schema}/key/{key}` on key in schema. If type is PUT then the body of the request must be a string (or a JSON object as a string) of the value to be put.

**GET** at `/schema/{schema}/key/{fromkey}/tokey` performs a range query of the range `[fromkey,tokey]` (prefixed by schema).

**GET** at `/schema/{schema}/prefix/{key}` performs a range query of all entries with the prefix key in schema.

**GET** at `/schema/{schema}` performs a range query of all entries with in schema.

All requests are responded to in a JSON format.

Client

**Listing 4.4:** Minimal Example of Caracal Client Usage

```java
import se.sics.caracaldb.Key;
import se.sics.caracaldb.client-blockingclient;
import se.sics.caracaldb.client.ClientManager;
import se.sics.caracaldb.operations.GetResponse;
import se.sics.caracaldb.operations.ResponseCode;

class SomeClass {
    public boolean someMethod(Key k, byte[] value) {
        BlockingClient client = ClientManager.newClient();
        ResponseCode putResp = client.put(k, val);
        if (putResp != ResponseCode.SUCCESS) {
            return false;
        }
        GetResponse getResp = client.get(k);
    }
```
A new client worker can be created with `ClientManager.newClient()` as seen in 4.4 line 9. As creation of a new worker has some overhead (e.g. sample requests to the bootstrap server) it is highly desirable to work with a fixed size pool instead of creating a new worker as the example might suggest. Lines 10 to 13 show how to execute a `PUT` and verify that it was successful. Lines 14-17 show how to execute a `GET` and verify that it was successful, while line 23 verifies that that value read it same as the previously written value (i.e. no one else wrote between the `PUT` and the `GET`). Finally lines 18 to 22 show how to construct a range query, execute it and verify that it was successful. Line 24 verifies that the the range contains something for `k`.

### 4.2. CaracalDB Data Model

#### 4.2.1. Introduction

The key-value interface provided by CaracalDB offers the foundation for the data model layer that brings support for storing and more complex searching of structured data. While the key-value interface provides a very fine grained control over what is being stored, it is also seen sometimes as being too low level. Some applications prefer to deal with a layer that provides support for storing data and taking into consideration its structure and subcomponents when providing the search capabilities. With the data model layer, data can still be searched using the key it was stored under, but the store can also be queried for data that contains certain patterns.

#### 4.2.2. Concepts

The provided data model stores structured data in a row oriented fashion. In order to be able to store a piece of data into the store, the data model needs to know the structure of this data in the form of a type definition. From now on, we will call this structured data as objects. Each object contains a set of fields and the type definition of this object contains the description of each of these fields in the form of field name and field type. Supported field types include: int, long, float, double, string and byte array. In order to provide the store with the ability to query data based on user provided patterns, secondary indexes are used. The type definition also contains information as to which indexes are defined for each type. A type can have zero or more fields indexed and the data layer provides support for indexing the following field types: int, long, string. Indexed fields of type string are limited to 255 characters in length.

Users can define databases as a private scope for their objects. Databases group together objects and metadata used for searching and querying these objects. The information schema contains the type definition for the saved objects and provides a mapping from human-readable database and type names.
4.2. CaracalDB Data Model

Databases, types and objects have identifiers and a combination of the three uniquely identifies a certain object in the whole store. Each of the three identifiers are byte arrays represented as hexadecimal strings.

4.2.3. Data model operations

The designed data model has two types of operations depending on what data model abstraction they operate on: type definition or objects.

Type operations

- `putType(databaseId, typeId, typeDefinition)`
- `getType(databaseId, typeId) : typeDefinition`
- `getAllTypes(databaseId) : {< typeName, typeId>}`
- `deleteType(databaseId, typeId)`

The `putType` operation requires a `databaseId` and a `typeId` in order to create a unique key for this type definition and add it to the store. The `getType` operation returns a type definition based on the `databaseId` and `typeId` that was provided. If no type definition can be identified by the provided identifiers, no type definition will be returned. The `getAllTypes` operation returns a collection of the type’s name and id. The `deleteType` operation removes the type definition identified by the provided database id and type id and also removes all objects of this type. The store should not contain any objects that do not have a type definition.

Object operations

- `putObject(databaseId, typeId, objectId, JSONObject)`
- `getObject(databaseId, typeId, objectId) : JSONObject`
- `query(databaseId, typeId, queryCommand) {JSONObject}`
- `scan(databaseId, typeId, objectIdRange) {JSONObject}`

The `putObject` operation requires the `databaseId`, `typeId` and `objectId` to uniquely identify the object and the object is stored as a JSON string. The `getObject` operation requires the same triplet: `databaseId`, `typeId`, `objectId` that uniquely identify the required object which is then returned as a string in JSON format. The `query` operation requires the pair: `databaseId` and `typeId` to identify the type of objects the user is requesting. The second part of the query operation includes the `queryCommand` that identifies the required objects of this type. At the moment the `queryCommand` is limited to one field - as can be seen in the examples from the user guide. The `scan` command contains the same pair of `databaseId` and `typeId` to identify the type of the required objects and also provides a range for the scan - if no range is provided, the scan will go over this entire type returning all the objects of this requested type.

All these operations are available through an object mapping interface similar to JPA or directly through the data model client.

4.2.4. Architecture

The data model features a simple architecture - fig. 4.3, with two similar layers on both the client and server. On the server side, the DataModel component encapsulates all the logic of the data layer transforming all the object and schema operations into PUT, GET and RANGEQUERY requests. The
ServerInterface components translates these PUT, GET, RANGEQUERY requests into the key-value store’s respective requests, in this case into CaracalDB requests. The ServerInterface also keeps track of pending CaracalDB requests and encapsulates logic for timeouts of messages. The server data model components are installed on hosts that also contain the key-value store. The data model sends the key-value requests to the same host and relies on the underlying store for routing these requests to the correct node.

On the client side, the ClientDataModel component contains functionality for assembling the data model requests from requests originating from the REST API or the object mapping layer. The ClientInterface layer handles the routing of requests as well as behaviour related to timeouts of requests.

![Data Model architecture.](image)

**Figure 4.3:** Data Model architecture.

### 4.2.5. Deployment Guide

The data model comes as part of the CaracalDB, in both client and server, and does not have any specific deployment requirements. Similar to the CaracalDB client, the DataModel client must be deployed as part of another application. Following the deployment guidelines from CaracalDB will also ensure that the data model is deployed and ready to use.

### 4.2.6. User guide

The data model has two interfaces that can be used: the pure client that expects both type definitions and objects to be given as a string with JSON format and the object mapping framework that parses the java classes into JSON type definitions and the java object instances into JSON objects. The put operation from both interfaces expect the type definition to be already present in the store and will not extract it from the object instance.
Client interface

As stated above, the client interface operations work on JSON strings. The expected format for the type definition and objects given in JSON strings are given in the following listings: listing 4.5 gives the formal JSON schema for our type definitions, listing 4.6 offers an actual example of a type definition as a JSON string, listing 4.7 gives the formal JSON schema for the our objects, listing 4.8 gives an actual example of an object defined as a JSON string. Listing 4.9 shows an example of a JAVA client and the usage of the datamodel features.

Object mapping interface

In order to ease the usage of our data model, a Java framework, similar to JPA is provided. Users familiar with JPA will be able to use our framework to put, get, scan and query for objects. The object mapping framework performs the translation from classes to JSON type definitions and from object instances to JSON objects. Similar to JPA we support entities, in the form of lightweight Java classes, which represent the actual type definition used by the data model. This relation between stored object and entity class can by specified using the @Entity annotation directly in the entity class file.

The users interact with the data model through the EntityManager abstraction. The EntityManager performs the type definition operations automatically, by parsing the entity class into a type definition and saving it to storage.

Currently supported annotations include:

- **Entity** can be applied on Class level. Marks a class as a type definition and allows objects of this particular type to be stored though the EntityManager
- **Id** can be applied at Field level. Marks a field as the object identifier
- **Index** can be applied at Field level. Creates a secondary index on this field
- **Immutable** can be applied at Class or Field level. Marks the class or field as being immutable. No updates of the marked object or object field will be allowed.
- **BigData** can be applied at Field level. Marks the field to be compressed or saved in the file system. No secondary indexes are allowed on this field
- **Denormalize** can be applied at Field level. Needs as argument an indexed field and will duplicate the value of the object’s field into the secondary index’s value

The EntityManager interface gives access to the data model object operations:

- `get(objectClass)`
- `put(object)`
- `scan(range, objectClass)`
- `query(objectClass)`

An example of a query using the object mapping interface is given in listing 4.10, which shows some of its basic features. Users familiar with JPA should find it easy to use this interface to the datamodel.
Listing 4.5: Type definition schema

```json
{
    "title": "type.definition.schema",
    "type": "object",
    "properties": {
        "typeName": {
            "type": "string"
        },
        "fields": {
            "type": "array",
            "items": {
                "description": "field.description",
                "type": "object",
                "fieldName": {
                    "type": "string"
                },
                "fieldType": {
                    "type": "string"
                }
            }
        },
        "indexes": {
            "type": "array",
            "items": {
                "description": "all.of.the.indexes.defined.for.this.type",
                "type": "object",
                "indexName": {
                    "type": "string"
                },
                "fieldName": {
                    "type": "string"
                }
            }
        }
    }
}
```
Listing 4.6: Type definition example

```json
{
    "typeName": "Type1",
    "fields": [
        {
            "fieldName": "field1",
            "fieldType": "string"
        },
        {
            "fieldName": "field2",
            "fieldType": "integer"
        }
    ],
    "indexes": [
        {
            "indexName": "field1Index",
            "fieldName": "field1"
        }
    ]
}
```

Listing 4.7: Object schema

```json
{
    "title": "object_schema",
    "type": "object",
    "properties": {
        "typeDefinition": {
            "type": "string"
        },
        "id": {
            "description": "hexadecimal string",
            "type": "string"
        },
        "fields": {
            "type": "array",
            "items": {
                "description": "values per field",
                "type": "object",
                "fieldName": {
                    "type": "string"
                },
                "fieldValue": {
                    "type": "string"
                }
            }
        }
    }
}
```
Listing 4.8: Object example

```java
1 {  
2     "typeDefinition": "Type1",  
3     "id": "A2_7F",  
4     [  
5         "field1": "one_string",  
6         "field2": "12"  
7     ]  
8 }
```

Listing 4.9: Example of DataModel Client Usage

```java
class SomeClass {
    public boolean someMethod() {
        ...  
        DMBlockingClient client = ClientManager.newDMBlockingClient();  
        String databaseId = "2A";  
        String typeId = "14_7C";  
        String objId = "11";  
        Pair<ResponseCode, String> get = client.get(databaseId, typeId, objId);  
        if (get.getValue0() == ResponseCode.SUCCESS) {  
            // get.getValue1() - object as string in json format  
        }

        String newJsonObject;  
        String newObjId = "AB";  
        ResponseCode put = client.put(databaseId, typeId, newObjId, newObj);  
        if (put == ResponseCode.SUCCESS) {  
            // success  
        }

        String beginObjId = "AA";  
        String endObjId = "FF_FF";  
        IdRange range = IdRange.open(beginObjId).open(endObjId);  
        Pair<ResponseCode, Map<String, String>> scan = client.scan(databaseId, typeId, range);  
        if (scan.getValue0() == ResponseCode.SUCCESS) {  
            for (Entry<String, String> object : scan.getValue1().entrySet()) {  
                // object.getKey() - objId  
                // object.getValue() - obj in json format  
            }
        }
    }
}
```
Listing 4.10: Object mapping framework example

```java
// An Entity example (type definition)
@Entity
class Photo {
    @Id String id;
    @Index String name;
    @Index int field;
    @Denormalize("name") string url;
    int size;
    @BigData byte[] photo;
    GeoTags tags;
    Date timestamp;
}

// EntityManager operations
...

try {
    String databaseName = "myDB";
    EntityManager em = EntityManager.getManager(databaseName);

    Photo photo1;
    em.put(photo1);

    String photo2Id = "A9_3F";
    Photo photo2 = em.get(photo2Id, Photo.class);

    String beginPhotoId = "F9";
    String endPhotoId = "A2_00";
    IdRange range = IdRange.close(beginPhotoId).open(endPhotoId);
    Set<Photo> photoSet1 = em.scan(range, Photo.class);

    Set<Photo> photoSet2 = em.query(Photo.class).where("name").is("\n\t\tStockholmPhoto")
        .open(range).close(endPhotoId);

    Set<Photo> photoSet3 = em.query(Photo.class).where("field").is(4);
    Set<Photo> photoSet4 = em.query(Photo.class).where("field").lt(4);
}
```
4. Platform Services

4.3. Tahoe-LAFS

4.3.1. Introduction

Tahoe-LAFS [26] is a decentralised storage system with provider-independent security. Hence the user is the only one who can view or modify disclosed data. The storage service provider never has the ability to read or modify the data thanks to standard cryptographic techniques. The general idea is that the client can store files on the Tahoe-LAFS cluster in an encrypted form using cryptographic techniques. The clients maintain the necessary cryptographic keys needed to access the files. These keys are embedded in read/write/verify "capability strings”. Without these keys no entity is able to learn any information about the files in the storage cluster. The data and metadata in the cluster is distributed among servers using erasure coding and cryptography. The erasure coding parameters determine how many servers are used to store each file which is denoted with N, and how many of them are necessary for the files to be available, denoted as K. The default parameters used in Tahoe-LAFS are K=3 and N=10, which means that each file is shared across 10 different servers, and the correct function of any 3 of those servers is sufficient to access the file. This makes Tahoe-LAFS tolerate multiple storage server failures or attacks.

The Tahoe-LAFS grid or cluster consists of a set of storage nodes, client nodes and a single coordinator node called the Introducer. The main responsibility of the Introducer is to act as a kind of publish-subscribe hub. The storage nodes connect to the Introducer and announce their presence and the client nodes connects to the Introducer to get the list of all connected storage nodes. The Introducer does not transfer any data between clients and storage nodes, but the transfer is done directly between them. The Introducer is a single-point-of-failure for new clients or new storage peers, since they need it for joining the storage network. We note that for a production environment, the Introducer must be deployed on a stable server of the community network.

When the client uploads a file to the storage cluster, an unique public/private key pair is generated for that file, and the file is encrypted, erasure coded and distributed across storage nodes (with enough storage space). To download a file the client asks all known storage nodes to list the number of shares of that file they hold and in the subsequent round, the client chooses which share to request based on various heuristics like latency, node load etc.

4.3.2. Tahoe-LAFS architecture in CONFINE testbed

In order to deploy Tahoe-LAFS in a realistic, community cloud like setting, we use the Community-Lab testbed, a distributed infrastructure provided by the CONFINE project where researchers can deploy experimental services, perform experiments or access open data traces. We use some of the available nodes (nodes in a production state) of that testbed for deploying the Tahoe-LAFS storage service. Nodes (e.g., research devices in CONFINE terminology) in the testbed run a custom firmware (based on OpenWrt) provided by CONFINE which allows running on one node several slivers simultaneously implemented as Linux containers (LXC). A sliver is defined as the partition of the resources of a node assigned to a specific slice (group of slivers). We can think of slivers as virtual machines inside of a node.

In order to deploy Tahoe-LAFS in the testbed nodes, we deploy the Cloudy distribution which contains Tahoe-LAFS, and place the Introducer node, the storage nodes, the client nodes, and the gateway

---

1http://community-lab.net/
2https://openwrt.org/
(optional) inside of the slivers of the testbed. Figure 4.4 shows the resulting Tahoe-LAFS architecture used by our experiments in the Community-Lab testbed.

![Figure 4.4: Tahoe-LAFS deployed in the Community-Lab testbed](image)

### 4.3.3. Tahoe-LAFS integration in Cloudy

The configuration of a Tahoe-LAFS node in a Cloudy node can be done almost automatically using the web interface, where the user only needs to type some very basic information.

![Figure 4.5: Partial screenshot of the Cloudy web interface showing the service discovery section. (Four different Tahoe-LAFS storage grids are shown.)](image)

The process of deploying a whole storage grid is also assisted by the Avahi service discovery mechanism [27]. First, to create a Tahoe-LAFS Introducer on a node, only a couple of fields in a form need to be filled in to provide the basic configuration parameters (the grid name, the node name and optionally if the default value is inadequate for some reasons the port the service will be listening to). As soon as the service is started, if it is marked as public, it is announced to the rest of Cloudy nodes. After that, adding a storage node to join the grid is just a matter of three steps:

- Listing the available services and browse to the Tahoe-LAFS storage section to find the desired introducer
- Clicking on the "Join grid" button for the desired grid (see Figure 4.5)
- Choose a name for the storage node and apply changes to deploy it (see Figure 4.6)
4.4. NatTraverser

NATs are a major challenge when building P2P applications. It is estimated that over 90% of all nodes on Guifi.net are located behind a NAT, and most wireless community networks have similar ratios. We call nodes behind NATs private nodes, as they do not support direct connectivity - a node outside the NAT cannot connect directly to the private node without some protocol or infrastructural support for traversing the NAT.

There are no TCP NAT traversal protocols that have had much success when deployed on the open Internet. Even Spotify’s music streaming client does not run NAT traversal protocols, as peers communicate using TCP [28].

The reason for this is that TCP NAT traversal protocols rely on an obscure feature of TCP (SYN-SYN) that is not uniformly supported by all operating system implementations of TCP [29]. The SYN-SYN feature of TCP is when two communicating nodes send each other a SYN packet and receive a SYN from the other node before receiving a SYN-ACK packet. In this case, the TCP connection should be established directly. However, this mechanism is very brittle as it requires both communicating nodes to have their clocks synchronized when sending the SYN packet and the skew between both clocks cannot exceed the latency between the communicating nodes.

In general, existing overlay network protocols are not practical over community networks because of the ubiquitous presence of NATs. In Guifi.net, each wireless antenna is assigned an open IP address in the Guifi network and given three extra open Guifi IP addresses. However, the typical home installation connects the Guifi wireless antenna to a consumer wireless router that acts as a NAT using one of these open Guifi IP addresses. This means that applications running on user devices will have private IP addresses. Hence, we need to use NAT traversal protocols to communicate with private nodes, except in the rare case where the source and destination nodes both resides behind the same NAT.

4.4.0.1. NatTraverser Middleware

We have developed four NAT-related middleware services (illustrated in figure 4.7) that are encapsulated in a library called NatTraverser. The services are:
4.4 NatTraverser

Figure 4.7: Nat-traversal middleware as a set of components. If a node has is deployed on the CLOMMUNITY nodes, it will act as a server providing Nat Type Identification (Stun) relay server (RvP), and hole-punching server (RvP). Private nodes are only clients of these services, using the ParentMaker component to find and maintain CLOMMUNITY nodes that will route control traffic to them.

1. a NAT type identification service (stun client/server),
2. a NAT relaying service (RvP),
3. a NAT hole-punching service (holePuncher and RvP), and
4. a ParentMaker service that connects private nodes to public nodes so that traffic can be routed to private nodes.

Nodes running NatTraverser services are lightweight and can be easily deployed on CLOMMUNITY nodes as cloud helper nodes to provide NAT-traversal services to client applications deployed on community networks behind home routers. As private nodes do not support direct connectivity, NatTraverser enables communication with private nodes by relaying traffic through a cloud helper node. Client applications using the NatTraverser library will transparently find and maintain a small number of cloud helper nodes (typically three) that will route control traffic to them. The ParentMaker component is responsible for finding and maintaining these cloud helper nodes. For brevity, we call cloud helper nodes parent nodes and clients on private nodes child nodes. NatTraverser requires that nodes can somehow discover other nodes in the system: parents need to discover other parents to provide the Stun service (which requires two public IP addresses, one from each parent), and children need to discover parents. To enable nodes to discover one another, ParentMaker is typically deployed with a NAT-aware peer sampling service, such as Croupier [30], that periodically provides a random sample of both public and private nodes in the system. Finally, although the standard use-case in the CLOMMUNITY network is that parent nodes will be deployed on the CLOMMUNITY nodes, they can, in principle, be run on node that supports direct connectivity.
4. Platform Services 4.4. NatTraverser

Implementation

In our implementation of NatTraverser, the middleware is written in Java using the Kompics framework. Kompics is an event-based programming model with support for network communication using the Netty framework (http://netty.io). Events are non-blocking (sent and received asynchronously), and network communication involves sending events to a network port that then forwards those events over the network using Netty to the destination component(s). The NatTraverser component queues up events destined for the network component. It queues the events until either a direct connection (preferred) has been established to the destination node or when a timeout has expired or when a relayed connection has been established. While events are queued, the NatTraverser component runs the highest priority hole-punching protocol in parallel. If the hole-punching algorithm successfully completes, the queued events are now immediately sent over the direct connection.

4.4.1. NAT-aware address format

NatTraverser provides its own overlay network address format, as nodes need to be able to send packets to other private nodes, and traditional IP-based address endpoints do not work for private nodes. To this end, we have defined the following NAT-aware addressing scheme, whereby a node can communicate with a private node by talking to any one of its parents. In fact, to reduce communication latency with private nodes, clients send messages in parallel to all parents, and the first message to arrive is received, while duplicate messages that arrive later from other parents are dropped by the ParentMaker component. Our NAT-aware node address format is defined in table 4.4.1.

<table>
<thead>
<tr>
<th>NatType</th>
<th>Ip</th>
<th>Port</th>
<th>Id</th>
<th>OverlayId</th>
<th>p1{ip,port.id}</th>
<th>p2{ip,port.id}</th>
<th>p3{ip,port.id}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>4 bytes</td>
<td>2 bytes</td>
<td>4 bytes</td>
<td>4 bytes</td>
<td>10 bytes</td>
<td>10 bytes</td>
<td>10 bytes</td>
</tr>
</tbody>
</table>

The NAT Type we use in our address format is based on the IEEE Behave specification [31] instead of the more traditional Stun classification [32]. Here is a brief description of what the different columns refer to, but the reader is referred to [31] for more detailed explanations of the different policies.

NatType = OPEN | NAT

Public nodes have an OPEN NatType, while private nodes have a NAT NatType. If a node has NatType of NAT, then the following NatType extension fields also need to be read to more precisely classify the NAT: MappingPolicy, AllocationPolicy, and FilteringPolicy. More restrictive NATs are, in general, more difficult to traverse.

MappingPolicy

The MappingPolicy, in more restrictive ordering, can be one of:
ENDPOINT_INDEPENDENT, HOST_DEPENDENT, PORT_DEPENDENT.

AllocationPolicy

The AllocationPolicy, in more restrictive ordering, can be one of:
PORT_PRESERVATION, PORT_CONTIGUITY, RANDOM.
FilterPolicy

The FilteringPolicy, in more restrictive ordering, can be one of: ENDPOINT_INDEPENDENT, HOST_DEPENDENT, PORT_DEPENDENT.

For private nodes, the address format will also include (by default 3) parent ip:port addresses for nodes that can route messages to the private node. The other fields, common to public and private nodes, are an *Id*, that uniquely identifies the node, and an *OverlayId* used by applications to uniquely identify application-level endpoints at a node. For example, in Gvod, we use the *OverlayId* to identify the destination video (from one of potentially many videos) for a particular packet.
5. Application Services

This chapter introduces the application services that are being developed for the CLOMMUNITY software architecture. These application services have been designed to provide services that have been deemed to be desirable by users of community networks, such as VoD. In D2.1, we also presented a plugin for Firefox that provides a network-aware HTTP proxy service for Guifi.net, and it not discussed further in this deliverable.

5.1. Video-on-Demand (Gvod)

VoD is a digital service that enables users to find and watch video content on-demand, at any time of their convenience. VoD is delivered over a network, often the Internet, and existing VoD systems support up to millions of concurrent users. P2P technology has been used to build VoD systems, by enabling nodes to collaborate in the downloading of video files, thus saving on the cost of bandwidth to the source of the video, and helping the system to scale to handle a larger number of concurrent downloaders than would be possible using only the upload bandwidth capacity of the source node. VoD contrasts with live-streaming technology that involves the broadcasting, in real-time, of video content to a large number of users at the same time.

5.1.1. Existing VoD systems

There are many successful VoD systems available on the Internet, such as Netflix and YouTube. What they have in common is that they are typically delivered by content distribution networks, with multi-level caching infrastructures that span the globe.

Although there are some P2P-enabled VoD systems, such as Popcorn (https://github.com/popcorn-time), there is no dominant, widely deployed P2P-VoD platform for the Internet. The main reasons for this are both technical and legal. Technically, many users on the Internet have asymmetric upload and download bandwidths on their home networks, due to the popularity of DSL network connections. DSL technology enables users to download at higher bandwidths than they can upload. In a P2P network, the aggregate download bandwidth is equal to the bandwidth provided by all nodes in the system. Since upload speeds are lower, on average, than download speeds for nodes in P2P networks on the Internet, a centralized service, typically backed by a content-delivery network (CDN), can provide better quality videos using its higher available bandwidth. Examples of such services are Netflix and YouTube.

Asymmetric network bandwidths are not the only factor reducing available upload bandwidth. The high proportion of nodes behind NATs (between 80-90% of all nodes) and the low success rate for NAT hole-punching protocols (that establish a direct connection to a node behind a NAT) has meant that existing P2P systems only provide a small fraction of their potential available upload bandwidth. In addition to the technical challenges, there are also legal impediments to P2P-VoD systems. Existing P2P-VoD systems typically lack copyright control, leading to the widespread illegal distribution of copyrighted material.

Given these challenges, there are still opportunities for P2P-VoD systems. The widespread adoption
of advertising by existing VoD services, such as YouTube and Dailymotion, has meant that users are now more willing to consider new advert-free VoD services – as long as they provide high quality video.

For wireless community networks, such as Guifi.net, the available aggregate download bandwidth from the Internet is relatively limited. Also, wireless networks have symmetric upload/download bandwidth capacity. The NAT problem still exists, but it is a challenge we address in our P2P-VoD system.

Existing P2P-VoD systems have either been based on the BitTorrent protocol, such as Popcorn, or heavily influenced by the BitTorrent protocol, such as Tribler/P2P-next [33]. However, a major problem with using BitTorrent for VoD is that designers are forced to have larger piece sizes for sharing video, as too many small pieces ends up creating a storm of control traffic, with peers constantly notifying all their neighbors for every small piece they download.

Our VoD system, Gvod, is based on a key insight. A node is only interested in another node’s video file pieces if its download position in the video file precedes the download position of the other node. We capture this neighbour relation using the VoD overlay network, a gossip-generated P2P topology. The VoD network topology self-organizes into logical concentric rings, such that nodes at earlier download positions in the file are found at increasing distances from the centre, while nodes that have downloaded the whole file are located in the centre of the topology. Nodes can now use knowledge of their neighbors’ position in the topology when both requesting new file pieces and when advertising file pieces. This vastly reduces the amount of control traffic that needs to be shared between nodes and enables us to reduce the piece size to 16 KB (instead of a more typical 1 MB). Smaller piece sizes means that nodes can more quickly identify slow neighbors and react to pieces that may miss their playback deadline. Finally, the Gvod protocol runs over UDP, which allowed us to integrate it with the NatTraverser library. As UDP has no native congestion control protocol, Gvod uses a simplified version of Ledbat (Low Extra Delay Background Transport), which backs off to TCP in the event of congestion at the client. This means that a user’s bandwidth is overloaded, video downloading will yield bandwidth to TCP-based services like web browsing and email.

5.1.2. Gvod

Gvod is a VoD service that has been adapted for CLOMMUNITY, based on previous work on video-on-demand carried out at SICS [34].

Gvod has a google chrome web-ui(fig.5.1) developed in AngularJS. The web-ui uses VideoJS as a video player which connects to the java P2P application and uses http streaming with byte range requests in order to play the video.

5.1.3. Gvod client features

The following interactive services are offered by the Gvod client:

- **play/resume** - play a video from the beginning or resume after pausing;
- **stop** - permanently stop the video;
- **pause** - Freeze the video, enabling it to be resumed;
5. Application Services

5.1. Video-on-Demand (Gvod)

Figure 5.1: AngularJS web-ui with VideoJS as video player

- **jump forward** - Jump to a particular time in the video in a forward direction;
- **jump backward** - Jump to a particular time in the video in a backward direction.

5.1.4. Gvod architecture

As mentioned already, Gvod consists of:

- AngularJS web-ui with VideoJS as video player.
- a REST API
- java P2P application.

VideoJS is an open-source video player that can play mp4/h.264 videos, the defacto standard for VoD playback. Users can search for videos using the Sweep search service, click on a link that then asks the P2P java application to download the video and plays it in the VideoJS player. The google chrome web-ui communicates with the Java application through the REST API and is thus able to ask it to download/upload a video, play/pause/jump through the video at play time. The video will start playing as soon as the java application has enough data and does not wait for the whole video to be downloaded.

VideoJS player, receives the end point of the java application Http Server. The player then uses http streaming with byte range requests in order to play the video.

When the user fast-forwards, stops, or pauses the video, the web-ui sends this commands through the REST API to the Java client which may cause the client to start downloading a new part of the video or pause downloading, respectively.

Deliverable D2.3
The Java client is also responsible for bootstrapping with connections to other nodes, sharing file pieces, and storing a list of active/downloaded videos in a library. The Java client is layered over NatTraverser, so it can establish connections with private nodes. The Java client is implemented using the Kompics Java framework. The Java client can manage the storage of video files as either memory-mapped files (for guaranteed smooth playback) or buffered files, which reduces memory overhead but may lead to a file being read from non-contiguous disk locations. The Java client uses Netty as a high-performance network library, and protocol messages are encoded into a compact binary protocol using hand-written message serialization functions.
5. Application Services

5.1.4.1. Cloud-assisted Vod

Gvod nodes can also be deployed as ordinary VoD peers on CLOMMUNITY boxes to provide an improved quality-of-service of video downloading for Gvod clients, see figure 5.3. Due to the unreliable nature of P2P VoD clients - they can leave the system whenever they want - the availability of some "Seeder" VoD peers on the CLOMMUNITY network should enable improved availability of videos and also reduced download times, as the VoD seeders will share their available bandwidth with Gvod clients in the system. As VoD seeders also have public IP addresses within the CLOMMUNITY network, they can also provide NatTraverser services: relaying, hole-punching and STUN services. This helps reduce the time for Gvod clients to perform hole-punching and to be contactable (by quickly finding parents).

![Figure 5.3: Cloud-assisted Vod and cloud-assisted Nat Traversal using nodes running Gvod and NatTraverser on the CLOMMUNITY cloud](image)

5.2. Decentralized Search

Full-text search is the de-facto standard for finding information on the Internet. From google to piratebay, people are used to writing words in natural language, pressing return, and finding what they are looking for. Currently, there is no decentralized, full-text search service available on the Internet or in wireless community networks. As part of CLOMMUNITY, we have been designing and building a fully decentralized search system, where clients can reliably and with low latency search for resources in the system.

The main challenges are to:

- build a reliable distributed search system using unreliable nodes and an unreliable network;
- build a low-latency distributed search system using a network and nodes with variable latency responses;
- integrate with NatTraverser to support NAT traversal;

Deliverable D2.3
5.2. Decentralized Search

Our overriding goal is to build a robust and reliable decentralized search service that supports low latency search - an upper bound of 1-2 seconds before a client receives search results. The service must also ensure integrity of the search index, protecting it from attacks to corrupt or bias the index.

The search service must work with peers behind NATs. The search service is designed so that nodes providing the search service can be either nodes in a community network (peers) or nodes on CLOMMUNITY boxes (cloud-assisted nodes). In D2.1, we described two possible architectures - either a CLOMMUNITY-supported search service or a fully decentralized search service. We have decided to adopt the fully decentralized search service model, as we can use the CLOMMUNITY nodes as ‘leader’ nodes to ensure the integrity of the index, and our search service will also be usable on the Open Internet as a P2P application.

5.2.1. Decentralized Search Service

We now present the architecture and basic functions of our decentralized search service. Firstly, the index, is sharded, as in fig. 5.4 into categories, and within each category, there may be a number of partitions. Larger categories will have more partitions. The idea is that search applications can support any set of categories they like. So an ecosystem of different search engines will be enabled. The index is implemented using a local Apache Lucene instance at each node with a bounded index size.

The user will interact with the application using a simple web-ui(5.1) providing a search box, similar to Google search. The service is implemented as a “large fanout service”, where the query is forwarded to a large number of servers in different search categories in parallel. Results are returned to the client that then orders them and presents them to the user. A timer is started when a query is issued, and a result should be returned to the user by at the latest $M$ milliseconds, as it is preferable to return any results (even incomplete) to the user in reasonable time rather than have the user wait too long. $M$ should be set by default to around 2 seconds, but can grow or shrink, dependent on user feedback and testing. To overcome variation in network latencies, nodes should send the query in parallel to $k$ (default 3) nodes in each of the $N$ partitions it searches in parallel. This also improves the reliability.
of the search service. After the first query returns from each partition, subsequent responses from the same partition are silently ignored. Depending on which happens first, either a result is received from all partitions or the timer expires, the client then re-orders the search results before returning to the user.

5.2.2. Search Architecture

The architecture of the search application is illustrated in figure 5.5. Internally, the system consists of a number of subcomponents: a index manager component, a routing component, a leader selection component and two overlay network protocols (Gradient and Croupier).

Croupier Overlay Network

Croupier is a NAT-aware peer sampling service [30]. Croupier periodically provides a node with a set of random NodeDescriptors of both public and private nodes in the system. NodeDescriptors are references to nodes that contain useful information about the node. We include in NodeDescriptors:

- the percentage of the Index currently available at that node
- the node’s utility value (used by the Gradient protocol for leader election)
- the node’s category,
- the node’s partition number within its category,
5.2. Decentralized Search

- the number of partitions within the node’s category.

**Gradient Overlay Network**

The Gradient topology is an overlay network where every node has a local utility value, and nodes preferentially connect to one another such that, given that all nodes have unique utility values, the topology will form a Gradient with the node the highest utility value at the center of the graph and the nodes with the lowest utility values will be leaves in the tree/graph. We use the Gradient overlay network to implement a Leader selection protocol, where the Leader group will be responsible for managing updates to the Index within a particular partition.

### 5.2.3. Design of the utility function

Although we could order a Gradient by nodeId, it would not help us with three problems:

- only nodes with good enough performance (high enough bandwidth, available disk space and cpu) should become members of the leader group;
- the number of index entries is the first part of the utility value – nodes with the set of full index entries should have higher utility values;
- only trusted nodes should become members of the leader group.

Our current format for the utility value at each node (subject to change) is: NatType : trusted : performance : NodeId : leaderId

- NatType – if the node has a NatType that is impossible or hard to traverse, then it shouldn’t be able to become a leader;
- trusted (boolean) – if the peer is a CLOMMUNITY node, it is considered trusted. Alternatively, we could use social network credentials to identify if a node is trusted or not;
- performance (int) – represents CPU/Memory/Network resources (up to 10 points for each). 30 points – max and the best value. Only for replacing a slow leader with a better one;
- NodeId (long) – id of a peer, smaller the value – better;
- leaderId (<long, String>) – used for conflict resolution when two leaders are presented in the system.

### 5.2.4. Leader Selection

Classical approaches to implement the leader abstraction in distributed systems use a leader election algorithm, where all nodes reach agreement on who is the current leader process. For example, before a task is started, all nodes may be unaware of which node will serve as the “leader”, of the task. When a leader election algorithm has been run, all nodes reach agreement that a particular, unique node is the task leader. Leader selection is the problem of how a system can reach agreement on one or more nodes as a leader, while bounding the cost of the algorithm, in terms of convergence time and message complexity. That is, the algorithm’s cost for any individual node should be independent of the size of the system.

We implement Leader selection by constantly building/maintaining a Gradient topology, and the nodes with the highest utility run an agreement algorithm amongst one another to determine which node will act as leader. In effect, we are transforming Leader selection problem into a localized...
search of the overlay network that has been pre-ordered using the Gradient.

We use a modified version of the Bully algorithm [35] to implement the agreement algorithm between the highest utility peers in the system. The modification we made to the Bully algorithm was to use a quorum-based failure detector to detect leader failure, instead of relying on an individual node’s failure detector. The modified algorithm works roughly as follows:

- when an individual node’s position in the Gradient has converged, each peer checks if it has a higher utility value than all its neighbors;
- if it does, it announces itself as a leader with a message to its neighbours (containing a list of its neighbors) that has to be ack’d by a majority of its neighbors,
- if, after a suitable timeout, no other node responds saying it sees a higher utility value, the node becomes leader and then it decides on the members of the Leader Group and it informs the other nodes about membership of the Leader Group.

5.2.5. Index Manager Component

When we have built the Gradient and selected a leader, we use the Gradient to propagate updates to the search index (entries in Apache Lucene), instead of using an anti-entropy (gossiping) algorithm. This is the key insight in our system. Instead of nodes having to pass enormous Bitmaps about which index entries they contain or compute unfeasibly large Merkle Trees [36], nodes only need to exchange their “first missing search index entry” value with one another. They can then pull any subsequent missing entries from other nodes. This is made possible, because all updates are performed using the leader – and nodes in the Leader Group have strong agreement on the entries in the index. Other nodes pull their updates from either the Leader Group or other nodes further down the Gradient.

One problem we have investigated is how to safely pull updates from nodes that are not members of the Leader Group (to load balance index updates over all nodes within a partition). To do this, we pick 3 random neighbours at random, but higher up utility levels in the Gradient, pulling the hashes of a bounded number of missing index entries. If we receive the same hashes from all nodes, then we assume the index entries are correct and pull the content from any one of the nodes (with higher probability a node lower in the Gradient, as higher nodes will probably have higher load). Then, validate the index entries match the hashes before inserting them locally.

5.2.6. Format of an index entry

Currently, we define a minimum number of fields that are required for the search service to work properly. It is assumed that the search service returns URLs for services (e.g., Video-on-Demand videos). This set of fields could be extended or made configurable.

1. IndexId (required, not searchable)
2. URL (required, not searchable).
3. Name of file (required, searchable)
4. File size (not required, searchable)
5. Date uploaded (not required, searchable)
6. Language (not required, searchable)
7. Category of the Content (video, books, music, games, etc) (required, searchable)
8. Free text description of the file, up to a limit of 5000 characters (required, searchable)
5.2. Decentralized Search

9. InfoHash of searchable content (required, not searchable)

10. LeaderId (required, not searchable)

---

**Figure 5.6:** Modified 2-Phase Commit Protocol. LG is the LeaderGroup, “ids missing by the leader” refers to any index entries the leader may have missed, for example during a network partition.

### 5.2.7. Adding index entries

First, you need to identify the category an index entry will belong to. Within that category, there may be many partitions. We need to load balance index entries across all partitions, so we will endeavour to pick a random partition in which to insert this index entry. The leaders in that partition will give the entry a unique index id. To do this, the client (inserting the index entry) will pick a random id, and one partition from the partitions in that category will be responsible for that random id. The reason why we don’t pick a random number from the number of partitions is that the client may have stale info about the number of partitions in the category. The client will contact 3 nodes that it believes are in the correct partition and asks them to insert the index entry on its behalf. If any of the clients knows that the entry should be in a new partition, it tells the client (giving the client some references to nodes in the new partition) and the client has to retry the whole operation on nodes in the new partition. Otherwise, the client controls the routing of AddIndexEntries by doing ‘iterative’ routing. That is, nodes don’t forward AddIndexEntries up to the leader – they send responses to the client, who continues to send requests until the client reaches the leader.

Once the Leader for the correct partition has been identified, the client sends its AddIndex request to it. The leader validates that the client is allowed to insert index values, and the leader adds the request to a batch of index entries that are updated periodically (typically once per second). If this is a duplicate AddIndex event, the leader returns an ACK. Batching enables smoother admission control – e.g., handle no more than 50 insertions per second and higher throughput. The leader of a partition will have its own ID space, and it will assign the next free id in that space to the entry. It will then
execute a 2-phase commit protocol (2PC) with all nodes in the leader group\(^1\); Our 2PC algorithm, 5.6 is simpler than the standard version, as our data is immutable and there are no ordering constraints on the entries. If 2PC commits, the Leader then sends an ACK to the client with a hash of data it wrote. If the client is in the same partition, it then adds the entry to its local Lucene index. If the insertion fails, the client retries at increasing intervals, until either success or a timeout.

### 5.2.8. Threat Model

In this section we identify possible threats to the search service. We attempt to model a strong adversary, who wants to explicitly attack each of the components that make up the search service. If we do not support user authentication, then we will allow Sybil attacks, where an attacker can create as many nodes with as many identities as they like, and try to overwhelm the systems with the many IDs.

**Threat:** Nodes lie about their local state to move up in the Gradient

If a node in the leader group lies about its bandwidth being higher than it is, and it is slow, we should be able to exclude it.

**Threat:** Nodes lie about closest peers in the peer exchange

A peer may claim that some peers are close (by utility function), but they are not. This could slowdown or even partition the overlay.

**Threat:** Nodes lie about amount of peers they can provide during view exchange (claims it’s small)

This will slow down the convergence of the gradient.

**Threat:** Malicious nodes in the leader group

If we use a Byzantine Fault Tolerant agreement protocol to implement leader group abstracts, all decisions taken by the leader group containing \(3f+1\) replicas should tolerate \(f\) failures (malicious nodes).

**Threat:** Leaders lie about their performance

If a leader lies about its performance making a new leader appear that is behaving the same way, it’ll slow down the overall performance of the system.

**Threat:** Unbounded rate of insertion of Index Entries

A solution to this problem must fairly reject extra insertion requests within a given time period. Fairly, means that misbehaving clients cannot use up the entire quota for inserting requests within a given time period.

**Threat:** Replication of data by copying data from a single node may lead to easy poisoning of the index

We should look at ways of either validating the data and the node from which the data was replicated. BAR Gossip is an extreme example of how to do this. Easier ways to do this would be to generate a hash of a range of index entries and validate with \(F\) other nodes that this data is correct. If it is not, reject the data.

**Threat:** Identify malicious nodes in the system

This would be an alternative to using a BFT protocol (where BFT protocols tolerate malicious nodes and leave them in the system). We would need a proof of misbehaviour (PoM) for nodes and a secure and trusted way of disseminating such PoMs.

\(^1\) [http://java.dzone.com/articles/xa-transactions-2-phase-commit]
Threat: **Partition overwhelming**
An attacker could create lots of nodes within one partition with the goal of overwhelming that partition. We should prevent nodes deciding on their own ID.

Threat: **Spy attack**
By logging all the requests each Sybil receives along with their information, like client IPs and client ports we can identify users uploading index entries.

Threat: **Index Lookup Pollution attack**
When a correct client searches for an index entry, it will receive data returned by the attackers instead. A pollution attack could also be at the level of a partition – attackers take over a leader group and start broadcasting polluted index data.

Threat: **Eclipse attack**
A directed attack on an individual node to prevent it being able to search the index. This can be within a partition or globally.

Threat: **Partition Lookup attack**
Other nodes will try to pollute the partition lookup table in the Search component, by filling our table with references to peers in all $N$ partitions with Sybils.

Threat: **Peer lie about which parts of index does it have**
A peer might lie about parts of the index it stores. For example, it can pretend there are no data entries he stores thus slowing down the system.

Threat: **Data substitution on its way through the network**
Every message should be encrypted.

Threat: **Peer lies about public nodes**
A public peer that issues information about public nodes for sampling lies about public nodes. This can partition the system.

Threat: **Peer drops data about parents in Node Address Format**
System might loose some public peers.

Threat: **Corruption of sqlite data**
A peer can corrupt any data stored in sqlite as pulling is done from one node.

Threat: **Peer data inconsistency**
Assume, viewSize attacker peers exist, they are all neighbours of each other. After a while, they start act like a leader for some messages (sending ACK for an ADD message, but not saving anything), for the other messages they just forward it upper in the gradient. Peer adds an index entry on ACK message, at the same time the real leader could add another entry with the same id. It will create a situation where part of an overlay has one value and another part the other one. There is no way in the system to understand which value is right and even notice the difference right now. Might be solved by signed and encrypted message passing to the leader.

Threat: **Byzantine peer claim to be the leader**
Byzantine peer may claim that he is the leader by broadcasting its public key, so write operation could be send to the wrong node.

Threat: **Slowing down the public nodes**
Byzantine nodes may gather information about public nodes, after necessary information is acquired, they can send many request to this nodes slowing them down or even crashing them.

Threat: **Sybil attack with Social Network credentials**
Nothing stops from a Sybil attack as it is possible to create multiple peers with the same Social
Network credentials.

**Threat:** Byzantine peer messes with categories distribution in the partitions

Thus irrelevant data could be returned as a search result or some data might be missing.
6. Cloudy: CLOMMUNITY GNU/Linux distribution

This chapter focuses on the CLOMMUNITY GNU/Linux distribution, codenamed Cloudy, aimed and designed for building clouds in Community Networks (CNs). It starts with the development motivations, followed by the specifications of the global architecture and the general requirements, and finishes with a summary of the current status along with an outlook of future work.

6.1. Motivation for a CN cloud distribution

The state of the art of CN clouds was discussed as part of the tasks in WP4 (see deliverable D4.1 for more information). There, in chapters 2 and 3 it was pointed out that, prior to the general adoption of cloud services, users of CNs already shared or provided services and resources to the community. However, these users were only a minority. One of the reasons identified was the technological barrier. Before actually providing them, users willing to share or provide contents and services with the community had to first take care of the technical aspects such as the deployment of their server with a set of services.

In the past, users from the Guifi.net\textsuperscript{1} CN tried to overcome this problem by releasing a GNU/Linux distribution named Guinux\textsuperscript{2} which provided end users with an easy way to offer network services to the community (HTTP proxy\textsuperscript{3}, DNS\textsuperscript{4} systems and MRTG\textsuperscript{5} graphic interfaces).

The key part of the distribution was a set of scripts that automatized the configuration process. End users were only asked for a few parameters such as their e-mail address and their node identifier. Shortly after the distribution was made available the number of end users sharing resources and providing services proliferated.

According to that, it became clear that lowering (or removing) the technological entry barrier encouraged users to provide more services and share their contents and resources with the community. To this end, one of the goals of the CLOMMUNITY project is to release a GNU/Linux distribution, codenamed Cloudy, aimed at end users, to foster the adoption and the transition to the CN cloud environment.

6.2. Community network cloud environment

Dimensions of CNs range from small ones with a handful of nodes to large deployments like Guifi.net\textsuperscript{6}. In such big networks, each cloud server is integrated within a group of neighbour cloud servers. This aggregation that has a reduced number of servers, which are geographically close, is referred as micro cloud. Cloud servers only announce their services and

\begin{itemize}
  \item \textsuperscript{1}http://www.guifi.net
  \item \textsuperscript{2}http://www.guifi.net/node/29320
  \item \textsuperscript{3}http://www.squid-cache.org/
  \item \textsuperscript{4}DNS: Domain Name System
  \item \textsuperscript{5}The Multi Router Traffic Grapher: http://oss.oetiker.ch/mrtg/
  \item \textsuperscript{6}http://guifi.net/guifi/menu/stats/nodes
\end{itemize}
discover other nodes in the context of their own micro cloud, by means of the Layer 2 network created with the virtual private network (VPN) service (see section 6.2). In Figure 6.1, where the CN cloud environment is depicted, these micro clouds are shown as green rectangular blocks.

Cloud services are not necessarily bounded exclusively to their micro cloud premises. For instance, if two micro clouds are connected by the CN infrastructure, services running on both micro clouds can interact with each other and federate if needed. A distributed storage file system, for example, could be built with nodes from different micro clouds to improve redundancy. The only restriction imposed by belonging to a micro cloud is that the announcement and discovery service does not extend further to other clouds. This is done to keep the cloud performance and the Quality of Experience (QoE) inside satisfactory values; otherwise, cloud servers would potentially start interacting with very distant nodes (even hundreds of kilometres away), leading to poor performance and eventually disturbing the global CN operation.

In the end, regardless of the partitions created by the micro clouds configuration, there is a cloud of services that operates agnostic about the underlying hardware or software managing the interconnection of the servers. This meta cloud can be deployed to fully extend the entire CN; the only limitation might be the performance restrictions imposed by the network in terms of bandwidth, latency and link quality, that will ultimately affect the user’s QoE perception.

Figure 6.1: Cloud servers, grouped into micro clouds (depicted as green rectangular blocks), are spread on different locations inside the community network, forming a meta cloud environment.
6.3. Distribution requirements

The global specifications for the distribution were discussed as part of the task 4.1 (WP4) in the development guidelines of Cloudy, leading to a process of research, testing and selection of the software that better suited the requirements. The following list summarises the specified requirements and the chosen solutions to accomplish them:

- **Distribution**: In order to foster services distribution amongst the CNs, a platform to publish and discover them is required (see points 2.3.3 and 2.3.4 of D4.1 for more details). This way, service access does not depend on static networking deployments and is aware of the dynamically-changing conditions of CNs. The software chosen for service publishing and discovering is Avahi\(^7\), on top of which service specific addons have been deployed.

- **Decentralisation**: As it has been found during WP4.1 and described section 2.2.1 of D4.1, in order to decentralise the resources, a common network layer is required to allow all CN cloud nodes to communicate with each other directly. A good way to build it is to create a virtual Layer 2 based on a VPN service. Among the available options, TincVPN\(^8\) has been chosen, as it fulfills all the requirements while being easy to configure and manage. Additionally, TincVPN is being successfully used in the CONFINE\(^9\) project for remote and automated management of research devices in CNs. To coordinate the devices with TincVPN, a tool to automate the system deployment has been developed as part of the project. This software, named Getinconf\(^10\), takes care of the TincVPN configuration process and the exchange of keys with the rest of the cloud nodes.

- **User-friendliness and experience**: As discussed previously in this chapter (see 6.1), providing tools suitable for technically unskilled users helps on the growing of the CN ecosystem. To this end, a web-based management platform\(^11\) has been developed aimed to integrate all the installation and configuration steps for the cloud services enabled in the Cloudy distribution. A simple web interface (see 6.4.3) is available to the end user as an easy way of configure, administer and monitor the cloud services running in the node.

- **Free, Libre, Open Source Software**: Among the many available options for FLOSS operating systems, the distribution has been based on Debian\(^12\) GNU/Linux. Apart from being one of the most popular distributions and fulfilling all the technical requirements, it has been chosen because the Debian Social Contract\(^13\) safeguards and guarantees that the software will always be open and free. Moreover, the Cloudy distribution is generated using Debian’s Live-build tool, which allows for great customisation and easy re-branding for Cloudy to be quickly adapted to other CN environments.

6.4. Implementation

This section covers the general architecture of Cloudy, the list of contents, the most relevant features of the distribution and the key points of the implementation process. Some screen captures of the

\(^7\)http://www.avahi.org/
\(^8\)http://tinc-vpn.org/
\(^9\)http://confine-project.eu/
\(^10\)https://github.com/Clommunity/getinconf-client
\(^11\)https://github.com/Clommunity/cDistro
\(^12\)http://www.debian.org
\(^13\)http://www.debian.org/social_contract
software in use are included as a graphical visualisation of the performed implementation (Figures 6.3 to 6.10).

6.4.1. Architecture

The architecture of the Cloudy distribution is depicted in Fig. 6.2, where a generic Cloudy instance is represented between the underlying CN infrastructure that provides network connectivity at the bottom and the user on top, representing any possible type of interaction with Cloudy.

A Cloudy instance can be run directly on a bare metal machine or on a virtual machine (see Fig. 2.4 and discussion in Chapter 2), providing almost the same services and capabilities\(^\text{14}\). Whatever the instance runs on, connectivity with the CN is a must in order to fully exploit the potential of Cloudy. The main block in Fig. 6.2 of a Cloudy instance comprehends the community network services, stressing the important role of cloud services in the center of the diagram. These services are the ones that benefit from or embrace the CN cloud environment to operate or offer a richer quality of experience (the list in the diagram is non-exhaustive, but mentions key services like distributed storage or different ways to reach video contents). Among them, virtualization is a special case. While other

\(^{14}\)some specific services like virtualization have strict hardware requirements
services focus on interaction and contents for the end user, provision of Infrastructure as a Service (IaaS) by means of virtual machines focuses on fostering the deployment of other services that run on top of this infrastructure.

Another special service inside a Cloudy instance is the distributed announcement and discovery of services (DADS). On the lower layer it provides the mechanisms and the infrastructure to other services to publish their information all over the CN. This is a valuable resource to orchestrate the CN cloud itself as it allows room for self-discovery, management and federation of services and resources. On the user interaction layer, the DADS allows the end user to discover the available cloud services in the CN and decide which services provider choose according to certain metrics (e.g. network round-trip time (rtt) to the services and number of hops). In general, this service can be used to publish any application running in Cloudy even if not cloud-related.

The Cloudy instance as a whole can be reached via both a web interface or the command line. The first one is designed to provide an easy and accessible interface to end users that covers the basic service operation (installation, configuration and management) and usage. The command line is the traditional approach to the system management, giving all the available administration options to more advanced or experienced users.

### 6.4.2. Services in Cloudy

Cloudy, as a distribution, provides several services in the different categories of the cloud environment. The most relevant ones have been included in table 6.1, organized in the four domains for use cases; IaaS, Network as a Service (NaaS), Platform as a Service (PaaS) and Software as a Service (SaaS)). The identified tasks are named and described, and in the last column is specified the chosen software solution.

### 6.4.3. Web-based management interface

One of the objectives of Cloudy is to allow non technical skilled users to deploy their own cloud servers and services. To this end, a web based management interface is available as part of the distribution. Once Cloudy is installed, the user is prompted with a message indicating the URL which can be used for accessing the web interface (see Figure 6.13) with any standard web browser from within the CN.

---

15 https://openvz.org
16 http://owp.softunity.com.ru/
17 http://docker.io
18 https://linuxcontainers.org/
19 http://www.avahi.org
20 https://www.serfdom.io/
21 https://coreos.com/using-coreos/etcd/
22 [WIP]: work is being done on this task but the outcome is not stable enough
23 https://github.com/CaracalDB/CaracalDB
24 https://coreos.com/using-coreos/etcd/
25 https://www.tahoe-lafs.org
26 https://www.syncthing.net
27 http://peerstreamer.org/
28 http://www.asterisk.org
29 http://www.voip-info.org/wiki/view/DUNDi
30 https://jitsi.org
31 https://github.com/jimdowling/gvod
Table 6.1: Contents of Cloudy, the CLOMMUNITY GNU/Linux distribution. Following the Debian terminology, tasks are sets of software packages that correspond to a functional unit and packages are pieces of application software (commonly called programs).

The management interface is a light website made with PHP\textsuperscript{32} combined with the Bootstrap\textsuperscript{33} framework. This way, non technically skilled users can find themselves comfortable and familiar with it. Figures 6.3 to 6.9 show the management web based interface.

The interface is designed as a whole-framed page with a horizontal bar on top. This bar contains the following drop-down menus to access the website functions (left to right):

- System: access basic system functions (password change, updates, logout...) (Fig. 6.3).
- Languages: change the language of the web interface (Fig. 6.4).
- Search: show the list of services discovered (Fig. 6.8).
- Clommunity: manage the cloud services shipped in Cloudy (Fig. 6.5).
- Guifi: manage the Guifi.net-related services shipped in Cloudy (Fig. 6.6).

### 6.4.4. Plugin-based modular architecture

The different services in Cloudy are integrated to the distribution as a set of plugins. Cloudy provides a set of common tools and mechanisms to interact with the user via the web interface and with the CN cloud via the command line.

\textsuperscript{32}http://www.php.net
\textsuperscript{33}http://getbootstrap.com

Deliverable D2.3
The Cloudy web interface is divided in three layers following a model-view-controller (MVC)\textsuperscript{34} providing the convenient resources for every service that is integrated. At the lower level, there are tools and scripts that interact with the operating system in order to install or remove software packages (either Debian packages or external ones), read and modify configuration files and exchange information with other Cloudy instances in the CN cloud. The higher level of the web interface provides a set of back-end tools to standardize the way information is displayed to the user. This includes a framework to add the services to the web interface menus, to translate web pages consistently and to render pages with a common design. Even if every service is added independently as a plugin, the result is a coherent web interface where the look and feel is kept coherent and consistent.

\textbf{Figure 6.3:} The Cloudy web interface basic system configurations.

\textsuperscript{34}http://en.wikipedia.org/wiki/Model-view-controller
6. Cloudy: COMMUNITY GNU/Linux distribution 6.4. Implementation

Figure 6.4: The Cloudy web interface has multilingual support. To add more languages only one file containing all the text strings needs to be translated

Figure 6.5: The Clommunity menu lists the cloud-related services shipped in the Cloudy distribution.

6.4.5. Distributed Announcement and Discovery of Services

As it has been found during WP4.1 and described in section 2.3.3 of D4.1, one of the most important things to help users participate from the network services are the service announcement and discovery tools. This is the reason why the model has been developed on the basis that each server publishes its available services with the necessary credentials to be used or federated, so that other users and servers can use it.

The Distributed Announcement and Discovery of Services (DADS) operates in parallel at both the global CN cloud level and at the microcloud level. On each of these two levels a different technological approach is used.
Cloudy includes a tool to announce and discover services in the CN cloud based on Serf\(^\text{35}\), a decentralized solution for cluster membership, failure detection, and orchestration. It relies on an efficient and lightweight gossip protocol to communicate with other nodes that periodically exchange messages between each other. This protocol is, in practice, a very fast and extremely efficient way to share small pieces of information. An additional byproduct of having this service distributed all over the CN cloud is it is used to evaluate the quality of the point-to-point connection between different Cloudy instances. This way, Cloudy users can decide which service provider to choose based on network metrics like rtt, number of hops or packet loss.

Another way to provide a DADS under development is based on etcd\(^\text{36}\), an open-source distributed key value store that gracefully handles master election during network partitions and the loss of the current master. The capabilities of this distributed store are many more than those of Serf and looks like a very promising application. Preliminary lab tests in the CN cloud, however, have performed slightly unstable, so integration of this solution will be left for further revision.

The second level of the DADS occurs in the microcloud, where a number of Cloudy instances are federated and share a common, private L2 over L3 network built with Getinconf. At that level, Avahi\(^\text{37}\) is used for announcement and discovery. Originally this solution was to be applied to the whole CN but as more Cloudy instances started to appear it became clear that the solution would not scale further than the tens of nodes. However, in the context of an orchestrated microcloud, it can be used not only for publishing cloud services but also other resources like network folder shares, etc.

In a Cloudy instance, the service announcement daemon periodically checks the status of the available services by means of their corresponding plugins and publishes this information, according to the user preferences, both to the CN cloud and to the microcloud (if it exists). The rest of Cloudy instances are aware of this information and update their lists of services. When a service in a node is stopped or removed, this information is also announced to the rest of cloud servers. The service discovery daemon continuously listens for announcements from other peers, and keeps an updated list of the

---

\(^{35}\)https://serfdom.io


\(^{37}\)http://www.avahi.org
available services in the micro cloud. This list can be consulted directly by the services or by the end user. In this case, operations are performed via the command-line interface (console) or using the Cloudy web-based management interface.

**Figure 6.7:** Cloudy geTinconf configuration page. The user only has to type in four parameters, the rest of the process is automated (key generation and exchange)
6.4. Implementation

6. Cloudy: CLOMMUNITY GNU/Linux distribution

Figure 6.8: The Search menu lists the options to use for the services publish and search.

Figure 6.9: Cloudy web-based services discovery shows the list of services in the micro cloud in a user-friendly way.

6.4.6. Clouder: Docker with Cloudy container

Guifi.net has started the development of a new approach to CN clouds that consists on using Docker\(^{38}\) (a Linux container (LXC) management system) to virtualise the Cloudy distribution. This approach adds a special support for infrastructure services (IaaS), as the cloud nodes are able to create multiple

\(^{38}\)http://docker.io
6. Cloudy: COMMUNITY GNU/Linux distribution 6.4. Implementation

Figure 6.10: Cloudy CLI-based services discovery shows the list of services in text format, with all the details

virtual machine instances for other purposes in addition to the ones dedicated to Cloudy.

Many things of this new design must be fixed and improved, but a first version of this infrastructure generator is publicly available at the Github account of the Clommunity project.\

39 https://github.com/Clommunity/lbdocker

Figure 6.11: Clouder Boot

6.4.7. Build tools

The Cloudy distribution environment comprehends the tools and applications that run inside a Cloudy instance and also the tools needed to generate the ISO and Docker images for installation or ready to run respectively.

Deliverable D2.3
For this purpose the Live Build Make\(^{40}\) tool has been developed, based on the live-build tool. It allows to easily create the image, specifically with the packages used in the Cloudy distribution. Additionally, after the image is generated, it gives the option to generate a Linux container template.

### 6.5. Installation process

As mentioned above in section 6.3, the generation process of the Cloudy distribution is based on Debian’s live-build tool\(^{41}\). It allows to create an installation image that can also be used as a Live CD\(^{42}\) for testing it without modifying the machine’s content.

The distribution can be tested in two ways: an image based installation, directly in a bare metal hardware or a virtualized installation, either with the ISO image in a virtual machine or with the template in a Linux Container. This means, in virtualization terms, installing it as a host or as a guest operating system.

#### 6.5.1. Image based

When starting the server with from the installation media, the user is prompted with a customised screen of GRUB\(^{43}\) showing the available options (Fig. 6.12). In the boot menu, the Install option is selected by default.

The default Cloudy installation process is identical to that of the regular Debian distribution. Most of the installation steps are automated, needing no decision making by the user, who is only prompted for basic parameters such as user-name, password, etc. This does not prevent more experienced users to opt for the advanced installation process at any time and fine-tune the options at their convenience (Debian mirror selection, custom disk partitioning, etc.).

After finishing the installation process, the server boots to Cloudy and performs some automated initial configuration tasks (enabling the virtual Layer 2 overlay network, etc.). By means of a console message (Fig. 6.13) the user is directed to the web administrative interface.

---

\(^{40}\)Live Build Make: [https://github.com/Community/lbmake](https://github.com/Community/lbmake)

\(^{41}\)http://live.debian.net/devel/live-build/

\(^{42}\)http://en.wikipedia.org/wiki/Live_CD

6. Cloudy: CLOMMUNITY GNU/Linux distribution

6.5. Installation process

6.5.2. Virtualized

As mentioned above in this section, Cloudy can be installed as a guest operating system using virtualization technology. It is an easy way to test the distribution which allows to create rapidly multiple instances of the system to test it.

The first option is to create a Virtual Machine (KVM, Qemu, VMware, etc.), selecting the ISO image as a boot device and then start the installation. At this point, the process is the same as it is explained in the previous subsection.

Another option to virtualize Cloudy is to use operating system-level virtualization\(^{44}\), with the template


---

**Figure 6.12:** Screen capture of the boot menu showing the available installation options

**Figure 6.13:** Screen capture of the Cloudy console after boot, showing the URL of the administrative web interface
6.6. Integration of new services

As the Cloudy distribution is adopted by users of a specific CNs, integration of services not envisioned during the development process will probably be needed. These can be new applications that appear or get a critical mass of users in the future (e.g. a secure file storage service with a radically new approach) as well as pieces of software of particular interest in a single CN (e.g. an applet to grab data from a local weather station). Since the initial developers of Cloudy may not be the ones that integrate such new services in the future, the process of integrating a new service or application must be understandable and well-documented.

Cloudy has been developed with modularity in mind. The different services available are integrated in the distribution as a series of plugs, meaning that the core of the system provides an interface and a set of interaction mechanisms which are common to all them. For instance, services are added to the user’s web interface by creating the required controller file in a common location, as well as the menu entries and the corresponding translations. The announcement of services is mostly identical for all of them, only changing the settings specific to the service itself.

The following services, Pastecat and SyncThing, are carefully documented in the project’s wiki to be used as examples or templates to integrate new services.

6.6.1. Pastecat

Pastecat\textsuperscript{51} is a very simple and self-hosted Pastebin service written in Go that supports various storage backends in a modular way. It is designed to remove pastes after a certain period of time but, if a persistent storage backend is used, pastes are kept between restarts. It is a useful tool to share texts and small (less than 2 MB) documents quickly between users in a CN but in an asynchronous fashion.

Figures 6.14b and 6.14a show, respectively, the main page of Pastecat and a web form where to paste a text or a file. The first one also gives the user a brief set of instructions on how to use the application. While Pastecat is a small piece of software (around 900 lines of code), its simplicity and reduced set of options makes it very convenient as a learning tool for including other apps in Cloudy.

Below it is explained in detail how to add a new service in Cloudy using pastecat software as an example.

\textsuperscript{45}https://linuxcontainers.org/
\textsuperscript{46}http://openvz.org/
\textsuperscript{47}http://www.vmware.com/products/workstation/
\textsuperscript{48}http://www.vmware.com/products/vsphere/
\textsuperscript{49}http://sp.parallels.com/products/pcs/
\textsuperscript{50}http://proxmox.com/
\textsuperscript{51}http://github.com/mvdan/pastecat
6.6. Integration of new services

6.6.1. Getting the binary

The first thing we have to figure out is how to download and install the binary on Cloudy. Most software out there is already available as a package on Debian, but Pastecat is not. If it were, it would be a matter of just running the command `apt-get install pastecat` from PHP. Therefore, we will have to get it from someplace else.

One option is to fetch the source and build it ourselves. This often means, however, that Cloudy should include a lot of build tools and libraries. In the case of Go, that would mean having its toolchain installed, which is not very practical.

A better option when the Debian package is not available is to download the binary from upstream trusted sources via HTTPS and, preferably, checking digests or using signatures. We can use Github’s releases page for that. Both options leave us with an executable file that we should be able to run directly on Cloudy.

In this particular case we are going to download the binaries for a Linux with a 386 architecture, from the git repository with the following command line:

```
wget https://github.com/mvdan/pastecat/releases/download/v0.3.0/pastecat_linux_386
```

6.6.1.2. Testing it out

Before adding Pastecat as a Cloudy service, we can configure and start it ourselves directly (i.e. manually), to see if it works and how. This way we can better understand what configuration options or command line parameters are we will need to run it as a service, and also to manage it once it is running.

6.6.1.3. Adding the controller

In `web/plug/controllers` we have one PHP file per service, which we call "the controller". This is the code that will run when we enter the services page on the Cloudy web interface.

Adding the index function

We also want our service to be integrated in the Cloudy web structure. To do this, a few PHP scripts need to be created and added to our device. Altogether, and by the time being, we will need to create a total of two scripts: `pastecat.php` and `pastecat.menu.php`. The first one is the controller itself, this is,
the script that renders the page and has all the information such as buttons or redirections. The other
one is what allows our service to show up in the upper menu bar of Cloudy’s web interface.

The menu code will look like this:

```php
1 <?php
2 // peerstreamer_menu.php
3 addMenu('Pastecat', 'pastecat', 'Clommunity');
```

By now, we’ll use a very simple PHP script in the controller:

```php
1 <?php
2 // pastecat.php
3 \textdolar\title="Pastecat";
4 
5 function index() {
6     global $spath, $title;
7     global $staticFile;
8     
9     $page = hlc(t($title), 4):
10     $page .= hlc(t("Minimalist, pastebin, engine, written, in, Go"), 4):
11     $page .= par(t("Can use variety of storage backends"), 'n', t("This software runs on the"), 'n', "<a href='http://pastecat'>".
12     
13     return(array('type' => 'render', 'page' => $page));
14 }
```

In our Cloudy system, these files must be placed at `/var/local/cDistro/plug/` directory. The first one at
`menus` subdirectory and the second at `controllers` subdirectory. Once we’ve done this, we can go to our
Cloudy system and access our new Pastecat.

**Making the controller install the service**

As said before, this step is made much more easier if the service is packaged in Debian. Since
Pastecat is not, we will have to do it manually. This usually involves a combination of
`wget`, `mv` and `chmod` commands. It is generally a good idea to keep the service’s files under `/opt/SERVICENAME`

In our particular case, the first thing we need is to download the binary from the release. In order
to do this we will make use of the mentioned `wget` command. Given a URL to a file, this command
allows us to download this file in our system, and this is what we will do in our system (as mentioned
before):

```
wget https://github.com/mvdan/pastecat/releases/download/v0.3.0/pastecat_linux_386
```

Once we have the binary we just need to move it to a directory where executable files use to be located.
In our case, we will use the directory `/opt/pastecat/`. To move these files through our system we will
use the command `mv`. However, first of all we need to create the directory where we will place our
binary. To do this we use the `mkdir` command as is shown below:

```
mkdir -p /opt/pastecat/
```

Once we have our directory created, it is time to move the binary there:

```
mv current_directory/pastecat_linux_386 /opt/pastecat/
```

Where `current_directory` is the directory where we previously downloaded the binary. Since the binary
name depends on the architecture, in order to simplify the controller’s code, we will change its name
to something more simple:

```
mv /opt/pastecat/pastecat_linux_386 /opt/pastecat/pastecat
```
Now our binary is called `pastecat` instead of `pastecat_linux_386`.

These steps are the minimum required to install a service which is not provided in the Debian official repositories. However, to an end user, it would look like a nightmare to run these commands in a console connected through ssh to its device, so what we are going to do now, is create a bash script which will be called later from the web interface by clicking a button.

This script is the first version of the Pastecat controller. For the time being, we will just include a function to install Pastecat in a device. Later we will include some other functions to add more facilities to our service.

```bash
#!/bin/bash
PCPATH="/opt/pastecat/

doInstall() {
    if isInstall
        then
            echo "Pastecat is already installed."
        return
    fi

    # Creating directory and switching
    mkdir -p $PCPATH && cd $PCPATH

    # Getting file
    wget https://github.com/mvdan/pastecat/releases/download/v0.3.0/pastecat_linux_386

    # Changing name so controller can invoke it generically
    mv pastecat_linux_386 pastecat
    chmod +x pastecat

    cd ~
}

isInstalled() {
    [ -d $PCPATH ] && return 0
    return 1
}

case $1 in
    "install")
        shift
        doInstall $@
    ;;
    esac

Making the controller use Pastecat

Start the service

The next thing we want is our software to be used through the web interface. In order to do this, we will include a new option to the main page of Pastecat, and also integrate a new function to the controller script to manage the binary. We will add the button after the Pastecat is installed message like this:

```php
$page .= addButton(array('label'="Create a Pastecat server"), 'href'=>$staticFile.'/?pastecatid='.
    'publish');
```

The next thing will be to implement the `publish` function in the same PHP. This function is the responsible of calling the appropriate function in the controller and to announce our server using the Avahi technology. The difference with this function is that it requires a form to introduce data, so in the end we will have a total of two functions; get and post:

```php
function publish_get() {
```
As we can see, in the post function we are invoking another function. The reason to do this is to write a more simple and modular code. In this function, we are finally calling the script:

```php
function publish_post() {
    $port = $_POST['port'];
    $description = $_POST['description'];
    $ip = "";

    $page = "<pre>";
    $page .= pcsourced($description);
    $page .= "</pre>"
    return (array('type' => 'submit', 'page' => $page));
}
```

The next step will be to create the publish function in the controller, then we will add a new function to the basic controller we had. We will add a new flag called publish, so the first executed part of the script will look like this:

```bash
if [ $# -lt 1 ]
then
dohelp
fi
case $1 in
"install")
    shift
doinstall $@
;;
"publish")
    shift
doserver $@
;;
esac
```

84 Deliverable D2.3
As we can see, when the script’s first argument is `publish`, we shift the rest of arguments and call the function `doServer`. In this function we must start the service with the required arguments, so the first thing we will do is to put the arguments into local variables. Once we do that the common thing would be to launch the Pastecat server, but since it might be called with root permissions (not recommended) we must run it as a nobody user. The issue is that the nobody user has merely no permissions and Pastecat needs some permissions to create folders and text files. In order to allow the nobody user to do that, we will create a folder and grant permissions to almost everyone to it. We will use `chmod` again. Now, the user can create files and directories within this directory, so we can now run pastecat. Finally, we keep the PID (process identifier) in a variable in case we want to use it in later updates:

```bash
doServer() {
    local port=${1:-""
    local description=${2:-"
    local ip=${3:-"0.0.0.0"

    # Creating directory with nobody permissions
    mkdir -p "/var/local/pastecat"
    chmod 777 "/var/local/pastecat" && cd "/var/local/pastecat"

    # Running pastecat
    cmd='su $PCUSER -c "$PCPATH$PCPROG -l $PORT" > $LOGFILE 2>&1 &'
    pidpc=$(eval $cmd) # keeping PID for future needs...

    cd -

    # Using the PID, we could carry on process control so if the pastecat process die, we can also # stop the avahi process to avoid "false connections"
}
```

Note that we are using some global variables that were not defined before such as `PCUSER` and `LOGFILE`. By default, we set these variables like this:

- `PCPATH="/opt/pastecat/"
- `PCPROG="pastecat"
- `LOGFILE="/dev/null"
- `PCUSER="nobody"

**Stop the service**

Sometimes, we may also want to stop our service, so we will provide a button to do that. Firstly we create the button, but if we think a little we will figure out that before doing this we need a way to know if our service is running. In addition, we also need a way to stop our service. Since we are running on Linux, we can easily achieve that by using the `kill` command. The point is that to use this command we need the PID, that we already kept it when we created the Pastecat server with `pidpc=$(eval $cmd)`.

Now we have everything we need to kill our process, we are going to provide a way so the PHP can detect whether Pastecat is running or not. An easy and resulting way to do this is storing some useful data in a file and delete this file when Pastecat is stopped. This way, we make sure that this file will only exists when Pastecat is running. This file will be created from the controller adding the following lines right below the sentence we mentioned in the previous paragraph:

```bash
# Writing server info to file
info="$pidpc http://$IP:$PORT"
echo $info > $PCFILE
```

**Deliverable D2.3**
Where $PCFILE$ is `/var/run/pc.info`. Note that the content of this file will be the PID and the complete location of our Pastecat server.

Now we have a way to know if our server is up or down, then we can add the stop button in the web interface. We will modify a little bit the PHP script that we had before, just by adding another advertisement indicating whether Pastecat is up or down, and adding two more buttons if it is running.

So, in our index function, within the condition that checks if Pastecat is installed, we will have the following code:

```php
$page .= "<div class='alert alert-success text-center'>" . t("Pastecat is installed") . "</div>

if (isRunning()) {
    $page .= "<div class='alert alert-success text-center'>" . t("Pastecat is running") . "</div>
    $page .= addButton(array('label'=>$t('Go to server'), 'href'=>"http://" . getCommunityIP() . " : " . $port)) ;
} else {
    $page .= "<div class='alert alert-danger text-center'>" . t("Pastecat is not running") . "</div>
    $page .= addButton(array('label'=>$t('Create a Pastecat server'), 'href'=>"$staticFile/parity/pastecat/publish"));
}
```

In this piece of code we can appreciate two new features. The first one is a check function called isRunning(). It looks very similar to the function we used to check if Pastecat is installed:

```php
function isRunning () {
    // Returns whether pastecat is running or not
    global $pcfile;
    return file_exists($pcfile);
}
```

It is a simple as it seems, it just checks if the file we created when starting the server still exists. The second thing we can notice in the new PHP code is the existence of a new function called stop(). It will invoke another function in the controller which will stop the Pastecat:

```php
function stop () {
    // Stops Pastecat server
    global $pcpath, $pcprogram, $title, $pcutils, $sahibi_type, $port;
    $page = "";
    $cmd = $pcutils . "$stop";
    execute_program_detached($cmd);
    return array('type'=>'redirect', 'url'=>"$staticFile/parity/pastecat");
}
```

In order to make the controller understand this order, we will modify the case and add the new function. In the case statement, we will add the following under the install option:

```php
"stop")
    shift
doStop $@
    ;;
```

This calls the function doStop within the controller. This function will look like this:

```php
doStop () {
    # Stopping pastecat server
    pcpid=$(cat $PCFILE | cut -d' ' -f1)
kil $pcpid

    # Removing info file
    rm $PCFILE
}
This function just gets the Pastecat’s PID from the file we created before, kills the process and finally removes the file; so the PHP scripts can know that Pastecat is now down.

Now we can create a Pastecat instance server and stop it. However, there is still something missing: make the other users see our service.

**Service publishing**

On of the best features in Cloudy is the facility of publishing our service as a publication in the network cloud, allowing other users to know what we are offering and to join our service. To do this, we first need to add a few lines to the PHP controller, just after we have called the controller to start the Pastecat instance.

We will add the following lines:

```php
$description = str_replace(' ', ',', $description);
$temp = avahi_publish($avahi_type, $description, $port, "");
$page .= ptxt($temp);
```

In the end our function will look like this:

```php
function pcsource($port, $description)
{
    global $pcpath, $pcprogram, $title, $pcutils, $avahi_type;
    $page = "";
    $device = getCommunityDev()['output'][0];
    $ipserver = getCommunityIP()['output'][0];
    if ($description == "") $description = $type;
    $cmd = $pcutils. "publish "$port "$description ";
    execute_program_detached($cmd);
    $page .= t($ipserver);
    $page .= par(t("Published this server."));
    $description = str_replace(' ', ',', $description);
    $temp = avahi_publish($avahi_type, $description, $port, "");
    $page .= ptxt($temp);
    $page .= addButton(array('label'=>t("Back"), 'href'=>"$staticFile./pastecat"));
    return ($page)
}
```

With this simple step, we announced our service in the network cloud. However the work does not end here, there is still one more thing to do: create a button and program it in order to when is clicked, it directly goes to our Pastecat server.

To do this there is a folder called *avahi* within the *plug* directory. The scripts that define the function carried on when the button is clicked are defined in different files within this directory, therefore we will create a new file called *pastecat.avahi.php* which will contain this:

```php
1 <?PHP
2 // plug/avahi/pastecat.avahi.php
3
4 addAvahi('pastecat', 'fpceserver');
5
6 function fpeserver($dates)
7    global $staticFile;
8
9    return ("<a class='btn' href='http://" . $dates['ip'] . ":" . $dates['port'] . ">Go to server</a>");
10 }
```

This will create a button beside the service announcement line that will point to our server.
Now that we have our service announced, we want it to disappear when we stop the Pastecat service. This last step is very simple and important. It consist of a few lines in the PHP function called `stop`. Until now, this function just called the controller and stopped the Pastecat, but now it will also stop the service publication and will show a flash comment so the user know it worked:

```php
$temp = avahi_unpublish($savahi_type, $sport);
$flash = ptxt($temp);
setFlash($flash);
```

These lines will be added just after the `execute_program_detached($cmd)` sentence in the stop function.

### 6.6.2. Syncthing

Syncthing\(^{52}\) is an application aimed at replacing proprietary cloud file synchronization services with something which is open, trustworthy and decentralized. It is built with the belief that user data is the user’s alone, so the user deserves to choose where to store it, who to share it with and how to transmit it over the Internet.

Many usages are envisioned for Syncthing in the context of Community Clouds. The service can be used by a single user to synchronize files and folders between his or her devices (this is: desktop machines, laptops, tablets or even smartphones) and the personal cloud node. This way, all the computers can have up-to-date versions of documents, media files, etc. Additionally, Syncthing can be used as a synchronized backup method between different machines, featuring versioning and recovery of removed files (depending on the chosen settings).

![Syncthing installation process in Cloudy.](image)

**Figure 6.15:** Syncthing installation process in Cloudy.

Repositories can be shared by two or more users by federating their respective Syncthing instances. By doing this data can be kept synchronized between different devices belonging to different users. This is a great feature for workteams sharing common documents, media or information since their data does not need to be sent from one user to the Internet cloud and then back to the rest of participants. One particular case of shared repositories can be considered for distributing data from one device with writing permission (master node) to the others that have read-only rights to the repository. This setup could be used, for example, for distributing data gathered by a sensor (atmospheric data, network data, etc.) among several other nodes to process it or take decisions accordingly.

\(^{52}\)http://www.syncthing.net/
6.6. Integration of new services

6.6.3. CaracalDB and the REST API

Both the CaracalDB key-value store and its REST API, including the simple WebUI, were integrated into the Cloudy web-interface, as shown in figures 6.17 and 6.18. The interface includes scripts for checking and installing the java dependency, downloading the caracaldb and caracaldb-rest jar archives, setting the configuration files, starting caracaldb, the REST API and integrating the WebUI into a running Apache webserver and viewing the logs.

The CaracalDB configuration includes:

- CaracalDB bootstrap address: ip and port – used for joining an existing CaracalDB cluster
- CaracalDB server address: ip and port – the address used for this instance

If bootstrap and server address are the same, a new CaracalDB cluster will be created with the current node as bootstrap node.

Figure 6.16: Syncthing’s web interface showing the shared repositories and the list of Synthing nodes that are exchanging data (this screen capture shows a single demo repository).

Syncthing is integrated in Cloudy’s web interface (Fig. 6.15). Installation is invoked by a single click (Fig. 6.15a) that calls a script. It automatically downloads the appropriate package according to the device’s architecture, installs and configures it, freeing the end user from this hassle. Once it is installed (Fig. 6.15b), the user can start and stop the service and choose whether to publish the service or not. Syncthing’s actual web interface (Fig. 6.16) is password protected and, besides that, is the same as if the application was installed outside Cloudy.

In order to provide a tutorial for CN users to add more services into Cloudy, the integration process of Syncthing has been documented in the project’s wiki page.
6.6. Integration of new services

Figure 6.17: CaracalDB integration in Cloudy.

Figure 6.18: CaracalDB REST API integration in Cloudy.

Figure 6.19: GVoD integration in Cloudy.

The CaracalDB REST API configuration includes:

- CaracalDB bootstrap address: ip and port – defines which running cluster to connect to (any member will work)
- CaracalDB client address: ip and port – the address used for this instance
- Hostname and port for this instance – defines where the REST API will be reachable via HTTP
6.6. Integration of new services

6.6.4. Gvod

Gvod, the video on demand service was integrated in the Cloudy web-interface (fig. 6.19). It includes scripts for checking and installing the java dependency, downloading the gvod jar archives, setting the configuration files, starting gvod and viewing the logs.

The gvod configuration includes:

- local address: ip and port
- CaracalDB bootstrap address: ip and port
- location of video library: directory path

6.6.5. Sweep

Sweep, the search service was integrated in the Cloudy web-interface (fig. 6.20). It includes scripts for checking and installing the java dependency, downloading the sweep jar archives, setting the configuration files, starting sweep and viewing the logs.

The sweep configuration includes:

- local address: ip and port
- sweep bootstrap node address: ip and port
7. Software Development and Quality

In this chapter we outline how the software developed as part of the CLOMMUNITY software architecture is created, tested and prepared for releases.

All software developed as part of CLOMMUNITY is open source (code references can be found in appendix B) and is developed by small teams or individuals using variants of Agile Software Development. The projects use relatively short release cycles and features included in releases are driven by the current needs of the platform or other services. In the current development of Cloudy the developer community is of small size and they agree among them to when an unstable version is released as stable.

Due to their different nature, the software services (i.e. application services and platform services) are tested differently from the Cloudy distribution. Most of the software services developed for the CLOMMUNITY software architecture are written for the Kompics component-framework[1], which is targeted as distributed systems which are well known to be difficult to test. Thus we begin by describing Kompics’ testing facilities.

7.1. Kompics Testing & Simulations

Kompics[1] is a programming model for distributed systems that implements protocols as event-driven components connected by channels. Kompics provides a form of type system for events, where every component declares its required and provided ports, which in turn define which event-types may travel along their channels and in which direction. The channels themselves provide first-in-first-out (FIFO) order exactly-once (per receiver) delivery and events are queued up at the receiving ports until the component is scheduled to execute them. A component is guaranteed to be only scheduled on one thread at a time and thus has exclusive access to its internal state without the need for further synchronisation. Different components, however, are scheduled in parallel in order to exploit the parallelism expressed in a message-passing program. A scheduled component is executed one handler at a time up to a configurable maximum, after which the component will placed at the end of the queue of components waiting to be scheduled. This behaviour provides a tradeoff between efficiency – re-using component state once it is loaded into cache – and fairness, i.e. avoiding starvation of components with fewer events. Events in Kompics are not addressed to components in any way, but are instead broadcasted across all connected channels. In this way the same event can be received by many components. The components themselves decide which events to handle and which to ignore by subscribing event handlers on their declared ports. Matching of events to handlers is based on the events’ type-hierarchy.

In addition to its runtime features, Kompics also provides a simulation environment targeted at repeatable and deterministic execution of a specific scenario. In this environment the normal thread pool based scheduler is replaced with a single thread executor which pulls events from a single global event queue in a deterministic fashion. Additionally the byte code is instrumented such that calls for system time and random values are replaced which calls to the simulation time and simulation random number generator. In this mode generation of new threads within the code is prohibited.
As can be seen in figure 7.1 the P2pSimulator component, which implements the discrete-event simulation system, provides three ports: The Network and Timer ports replace the real components normally used in the system during runtime. The network is simulated based on a pluggable network latency model and the timer is linked to the simulation time. The third port provided is a custom experiment port which bridges the scenario domain specific language (DSL) with the actual implementation system.

A scenario is a parallel and/or sequential composition of stochastic processes. A stochastic process, a finite random sequence of events, is called with a specified distribution of inter-arrival times of events to be triggered on the experiment port. It is then the responsibility of the custom simulation component (MySimulator in figure 7.1) to ensure they have the desired effects on the peer(s) that is/are being simulated.

In order to use Kompics simulations as tests, they are wrapped in JUnit test cases, with every case corresponding to a different scenario or different setup. Additionally, it is often necessary to add some kind of validator class to the JUnit test that can inspect either the system state itself or collects system responses and checks them against the specifications of the system.

7.2. Software Services

The following services developed for the CLOMMUNITY software architecture are written in Kompics and use the testing approached described in section 7.1 in addition to normal class based unit tests: CaracalDB, CaracalDB Data Model, NatTraverser, GVoD, and Sweep.

CaracalDB and Data Model have two types of large Kompics simulation based integration tests. Both of them boot up a small 6 node cluster in simulation mode. They then proceed to test random interleavings of 1) PUTs and GETs, or 2) PUTs and RangeQueries. The same operations are also applied to a ValidationStore that is part of the simulation component and acts as a global reference.
store for how the distributed version should behave. After a scenario the contents of the distributed store are validated against the global store.

**GVoD and Sweep** use Kompics simulation based tests to check the correctness of its middleware components: *Gradient* and *Croupier* as well as the system as a whole. A simulation scenario is used to verify the fact that the peer sampling service, Croupier, provides random samples under no churn as well as under different churn rates. The Gradient service is also tested through simulation if it convergences under different churn and changing node utility scenarios. GVoD simulation scenarios includes booting a GVoD system and checking the downloading mechanism under different network properties like lossy links or link latency. Sweep simulations include booting a sweep system with different number of nodes (100, 200, 400) and verifying the dissemination time of an index entry, verifying the system stability under churn or constant index entry additions.

### 7.3. Cloudy

The Cloudy distribution, based on Debian GNU/Linux, consists of the base operating system layer and the CN cloud software layer. Most of the development for the CLOMMUNITY project was performed based on Debian 7.x *Wheezy*, which was the stable release at that time. Later, when Debian 8 *Jessie* and the current version 8.1 were published, development was switched over to the latest. This move included a process of adapting small bits of the software to the new release. Therefore, the base layer of Cloudy follows the same quality standards as the Debian stable release (i.e. *the production release of Debian, the one which we primarily recommend using*¹)

The key component of the CN cloud software layer is the web interface that provides user-friendly mechanisms to install, configure and manage the different applications available in Cloudy. This interface is built on top of a custom framework with a modular design based on Bootstrap². The framework provides an abstraction layer for Debian packages installation and uninstallation, HTML pages generation (e.g. consistent design, forms), configuration files management, etc. This way, a programmer integrating a new application in Cloudy does not need to learn how to perform all the required tasks (installing packages, designing HTML pages, acting on configuration files, etc.) but to only use the readily available routines.

The source code of Cloudy is produced by a small team of cross-functional developers using agile software development practices including scrum development, continuous integration, test-driven development and pair programming. Before releasing the software to the public, it undergoes a series of testing steps to ensure its quality. These steps include localization, installation testing, functional testing, compatibility testing, regression testing, usability testing and finally beta testing performed by active CN users.

#### 7.3.1. Cloudy releases policy

Cloudy can be obtained as an installable ISO image for x86 and x86_64 architectures, or downloaded and integrated into an already installed Debian-based system.

Installable ISO images for Cloudy are offered in two flavours, *stable* and *unstable*. The stable release provides a frozen snapshot of the Cloudy distribution where all the key components have been

¹Debian releases: [https://www.debian.org/releases/](https://www.debian.org/releases/)

²Bootstrap: [http://www.getbootstrap.com](http://www.getbootstrap.com)
thoroughly tested and are ready for a production environment. However, as development goes on, new features are added to the Cloudy distribution. For this reason, the unstable release is recompiled everyday in a *nightly build*. This allows users to install an up-to-date version of Cloudy on their machines.

As of writing this document, the current stable version of Cloudy is based on Debian 7.8 *Wheezy*, published as *oldstable*[^3], and the unstable version is based on the current stable Debian 8.1 *Jessie*. Based on this latest, a new stable release of Cloudy is expected to be published in the next future after the delivery of this document.

Users willing to integrate Cloudy into their already installed Debian-based systems can do it on top of both Debian 7.x or 8.x, while the latest is always recommended. By default, the resulting system is equivalent to installing the unstable Cloudy version.

Additionally, the web interface includes a section to update the system, both the Cloudy applications and the operating system packages.

### 7.3.2. Applications integrated into Cloudy

The different applications integrated into Cloudy are developed using the same *agile software development* practices described in Section 7.3. Two categories can be distinguished here: *core* components and *generic* applications.

The core components are those applications that provide a service or a characteristic that must be available for every Cloudy instance, no matter which hardware it runs on. An example of this would be the Distributed Announcement and Discovery of Services, which plays an important role in the deployment of a CN cloud, or the updates system. For this reason, the core components are payed special attention to ensure software quality and they are tested to work correctly on all the supported hardware architectures.

In general terms, the applications integrated into Cloudy are based on the versions published as stable software by their own developers. There are, however, some applications included in Cloudy that are under continuous, active development where their stable version is updated very often (weekly, or even every 4 or 5 days). In those cases, before updating the application in Cloudy, a number of backwards-compatibility tests are run in order to ensure that the software is usable and does not cause any problem to nodes running older versions, etc.

### 7.3.3. Third-party components

In order to facilitate CN users to integrate their own applications into Cloudy, a detailed step-by-step tutorial[^4] has been published showing how to create a new plug-in that adds a simple application (*PasteCat*[^5]) to the system. The tutorial contains all the information needed to learn how to create a new plug-in for Cloudy’s web interface, how to manage the installation of required packages, etc. so that users with a minimum experience in programming can contribute to the distribution.

[^3]: Debian OldStable: [https://wiki.debian.org/DebianOldStable](https://wiki.debian.org/DebianOldStable)
[^5]: *Pastecat*: [https://github.com/mvdan/pastecat](https://github.com/mvdan/pastecat)
8. Conclusions

The main software system contributed by CLOMMUNITY is the Cloudy Linux distribution that provides decentralized kernel and network services required to build and manage cloud-based services for wireless community networks. The Cloudy distribution is available for download now, and many of the platform services and applications developed in the project are already integrated in the distribution, while the remaining services will be integrated several months before the end of the project.

Cloudy is a Debian-based Linux distribution. Cloud platforms deployed by community networks can use Cloudy to be able to offer users the ability to use the application services developed in the project (such as video-on-demand) and also to deploy their own custom services. To support user-built services, Cloudy also provides a number of decentralized platform services, such as key-value (object) storage and a file system. Such services dramatically reduce the barrier to entry for less experienced developers in building useful services and applications for wireless community networks.
A. System Requirements

In order to enable and provide a community Cloud, i.e. IaaS and 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 services, including network-aware services such as service location and discovery, auto-managed DNS, adaptive HTTP proxy and NAT traversal and storage, federated identity management, Cloud federation and configuration that are needed to build community Clouds, as well as higher level Cloud-based services and applications in a community network.

Final revision of the aforementioned requirements for a community cloud are explained and discussed in the following sections.

A.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.

A.2. 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.

A.2.1. 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.
A.2.2. 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.

A.2.3. 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 NAT devices and firewalls. Hence, we need NAT traversal protocols to communicate with the nodes reside behind NAT.

A.2.4. 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.

A.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.

A.4. Configuration of Micro Clouds

In Clommunity, resources hosting Cloud services are geographically distributed and connected through community network. There are many ways to configure micro clouds on resources shared by the community. However, the topology of the micro clouds’ infrastructure can affect the quality of services. Therefore, one of the requirements of Clommunity is to find optimal micro clouds configuration in terms of better performance, reliability and availability.
A.5. Federated Identity Management

Authentication of individuals to use Clommunity services is one of the requirements. A federated identity management system should enable Clommunity users to authenticate to micro clouds using a single identity. Indeed, it should enable to define the amount of resources (quota) which can be consumed by every user.

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.

A.6. Monitoring

Services and infrastructure in Clommunity need to be monitored in order to enable system administrators or service managers to make decisions based on the aggregated data. Monitoring of Cloud application and services can be done in two levels of the infrastructure and the application. Metrics related to infrastructure services such as CPU, memory and bandwidth are required to be monitored for the decision making process such as allocation decisions. Monitoring on the application level can collect data related to Cloud resource usage of an application or service quality metrics.

A.7. 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.
B. Software Packages

The list of software delivered as part of this deliverable is maintained on the CLOMMUNITY Wiki here:

http://wiki.clommunity-project.eu/soft:sourcecode

The above link will contain the latest links to the software developed in the project. For convenience, the list of software packages delivered are reproduced here:

**Cloudy**

**lbmake** https://github.com/Clommunity/lbmake
Build and package the CLOMMUNITYdistro (cloudy)

**avahi-ps** https://github.com/Clommunity/avahi-ps
Avahi Search & Public services over TincVPN in community IP

**cDistro** https://github.com/Clommunity/cDistro
System php scripts to configure Services in a Linux Distro

**getinconf** https://github.com/Clommunity/getinconf
Web application to manage networks made TincVPN

**getinconf-client** https://github.com/Clommunity/getinconf-client
Client agent to configure TincVPN

**Clouder**

**lbdocker** https://github.com/Clommunity/lbdocker
Build the docker distro with cloudy distro like a container

**Platform and Application Services**

**CaracalDB** https://github.com/CaracalDB
CaracalDB key-value store

**GVoD** https://github.com/o-alex/GVoD.git
P2P Video-on-demand for Clommunity Networks.

**Sweep** https://ghetto.sics.se/jdowling/peersearch.git
Decentralized P2P Search System based on Apache Lucene.

**common gvod/sweep** http://ghetto.sics.se/alidar/globalcommon.git
Common code, used in both GVoD and Sweep

**natTraverser** https://github.com/jimdowling/nat-traverser.git
Peer-to-Peer Nat Traversal Middleware
**network-aware Proxy for Guifi.net**  https://cloud7.sics.se/clommunity-proxy.zip
Network-aware Proxy Plugin for Firefox for Guifi.net.
C. Obsolete Software

During the research in WP3 and WP4 some software systems that were included in previous deliverables have been judged to be unsuited for the final architecture. They are included here for reference. Those systems are mostly concerned with cloud management and multi-cloud management, which was finally decided to be unnecessary for CLOMMUNITY as all our services run directly from within the Cloudy distribution as described in chapter 6. As a consequence of moving away from cloud management software, ElastMan became obsolete, as well, since it is targeted at cloud environments.

C.1. Eucalyptus Cloud Platform

Eucalyptus\(^1\) is a cloud management platform that is free and open-source and supports building Amazon Web Services (AWS)-compatible private clouds. Another main advantage is that Eucalyptus can be used to set up hybrid cloud set-ups where resources from AWS can be combined with private infrastructure, as shown in Figure C.3. Eucalyptus supports KVM and VMware ESXi hypervisors. The overall platform architecture of Eucalyptus system (Figure C.1) consists of six main components (Figure C.2). These are Cloud Controller (CLC), Walrus, Cluster Controller (CC), Storage Controller (SC), VMware Broker and Node Controller (NC). The different controllers of Eucalyptus are installed on different machines, though Eucalyptus also provides a Cloud-in-a-Box edition that includes all the components in a single installer allowing to quickly set up Eucalyptus on a single machine. More machines can be added as worker nodes later on.

For our testbed, we installed Eucalyptus Cloud-in-a-Box edition on a machine with 4x Intel(R) Core(TM) i7-3770 CPU 3.40GHz, 16 GB RAM, and 1 TB hard disk. This cloud is located in UPC Campus Nord Lab in Barcelona city, and accessible from Guifi.net network (IP address: 10.1.24.5\(^2\)). The installation process was straightforward and set up Cent OS as host operating system, on top of which all the components of the Eucalyptus were deployed. Eucalyptus provides both a web-based graphical interface as well as command-line utilities for configuring the cloud platform. After the installation, using the web access of Eucalyptus from any machine in the Guifi.net, virtual machines can be created and managed.

Post installation, we only encountered minor issues with setting up network access, as Eucalyptus provides a number of network modes with different features. We used the managed virtual LAN mode and Eucalyptus required preferably a reserved contiguous IP addresses block for allocating to the different machines. We had sufficient IP addresses (total 26) in our subnet for this small testbed, and all the virtual machines were directly assigned addresses from Guifi.net. However we anticipate that as we add more machines in future, the advanced networking features of Eucalyptus have to be enabled and configured to use IP addresses that are not in a contiguous range.

We explored the different options available in Eucalyptus management dashboard, and tried different ISO images. We have set up a number of Debian and Ubuntu virtual machines on Eucalyptus cloud and they have performed stable for the last few months. We also explored integrating Eucalyptus with

\(^{1}\)https://www.eucalyptus.com

\(^{2}\)102
federated cloud management system (see section C.4), like Scalr and ECManaged, however, because of limited number of users and the lack of federated identity management, we find the interface of Eucalyptus to be sufficient and powerful enough for our use case for the time being. Overall we have found Eucalyptus to be very stable and robust and easier to install and manage with sufficient functionality for our use case. We plan to keep this machine with Eucalyptus running for CLOMMUNITY testbed to contribute VMs for cloud-based services and to experiment with federation options between Cloud Management Systems.

C.2. Scalr Cloud Management System

Scalr\(^2\) is an open-source cloud management platform which allows automating deployment and management of multiple clouds installations. Scalr supports most of the popular public and private cloud

\(^2\)http://www.scalr.com
providers such as Amazon EC2, Google Compute Engine, Rackspace Open Cloud, OpenStack, OpenNebula and Eucalyptus. Scalr software is offered as a hosted subscription-based service to the enterprises, however, users can also install the free version on their own premises. Scalr provided advanced functionality like failover, autoscaling and orchestration, and RightScale\(^3\) is one of its main competitors in this area. Figure C.4 shows the overview of Scalr system and how it supports managing both public and private clouds at the same time, and Figure C.5 shows the different components of Scalr interacting in the overall architecture.

There are two core concepts in Scalr which are important to know in order to use it: Farm and Role. A farm is a the blueprint for the architecture of the deployable service. Roles are the building blocks of a farm. A role is composed of images and automation. The image is created inside the cloud provider and stored as a snapshot of an instance. For example, a role can be an instance running a Mysql Server. A role can have multiple images in order to enable a multi-cloud setup. The automation, control roles on their life cycle and is stored in Scalr. For example, automation can be configuring password and firewall rules on a database server, automatic discovery of backend servers and orchestrating fail over between a database master and a database slave. A rich guide for Scalr is available on its wiki page\(^4\).

We setup Scalr in our infrastructure to federate different cloud installations for better streamlined access to the multiple cloud installations. We installed Sclar cloud management system within a VM on Proxmox-based machine with 4x Intel(R) Core(TM) i7-3770 CPU 3.40GHz, 16 GB RAM, and 1 TB hard disk. This Sclar node is located in UPC Campus Nord Lab in Barcelona city, and accessible from Guifi.net network (IP address: 10.1.24.9). We explored managing OpenStack and

\(^3\)http://www.rightscale.com/
\(^4\)http://scalr-wiki.atlassian.net
Eucalyptus cloud from Scalr management system. Installing Scalr was straightforward, however, the integration of OpenStack and Eucalyptus was complicated because of networking and firewall issues. Most of the machines in CLOMMUNITY testbed are based on Proxmox, which is not supported by Scalr, so the utility for our use case was limited. Moreover, unlike RightScale, Scalr does not support federated identity management for multiple clouds so this was another limitation for our scenario, since users need to set up credentials for each cloud individually before they can use access their VMs from Scalr. We plan to keep Scalr setup running, since it will help scale management as more clouds are added to CLOMMUNITY testbed.

C.3. ECmanaged Multi-cloud management

ECmanaged\(^5\) is a commercial multi-cloud management platform. CLOMMUNITY has reached a collaboration agreement with Ackstorm\(^6\) to use ECmanaged for community clouds in Guifi.net. In the following, we show some of the features of ECmanaged that we tested. As result of these tests,

\(^5\)http://www.ecmanaged.com/
\(^6\)http://www.ackstorm.es/
the following options seem feasible: 1) ECManged itself is a tool that is hosted by Ackstrom in the Internet. The OpenStack manged clouds within Guifi will have public IP address, to be reachable by ECManged. 2) Instead of a public IP address for the community cloud OpenStack manager, a VPN will be set up with ECManged.

After login, the user sees the ECmanaged dashboard that is shown in Figure C.6. ECmanaged offers the option to add additional cloud providers (see Figure C.7). Through this option, community cloud providers could bed added. In particular the OpenStack Cloud Management platform is supported.

A set of different templates are offered through ECmanaged, in order to deploy applications in the cloud infrastructure. Figure C.8 shows given templates and Figure C.9 shows the option to add customized templates. This feature could enable selecting CLOMMUNITY’s Cloudy distribution (see chapter6).

C.4. Cloud Federation in Community Clouds

In this section, first, the notion of Cloud Federation is defined based on our research. Then the motivation for federating clouds are discussed with some use cases in the community clouds. Afterwards the requirements and specifications of a cloud federation solution are summarized and explained. The outcome of our research on the current existing open source cloud federation projects are written and their pros and cons are explained.
C. Obsolete Software

C.4. Cloud Federation in Community Clouds

Figure C.6: ECmanaged Dashboard

Figure C.7: Add additional cloud providers to ECmanaged.

Figure C.8: Select from given templates.
C.4. Cloud Federation in Community Clouds

C.4.1. Cloud Federation

There is no single definition of cloud federation [37, 38, 39]. One thing which is common between all the different definitions of cloud federation is that, in a federated cloud, users are able to utilize the resources of multiple cloud providers in which each cloud provider can have heterogeneous resources and cloud computing platforms. Generally, federation accommodates beyond mere interface adaptation of different heterogeneous cloud providers. A federation can act as a broker for the resource pool comprised of different providers’ resources, can facilitate resource migration, applying global service level agreement (SLA), managing power consumption and so on.

C.4.2. Why Cloud Federation Is Important?

C.4.2.1. Motivation

The motivation to choose federation solutions comes from two aspects. One aspect is the bottlenecks and problems of using a single cloud provider and the other aspect is the benefits that can be gained by having a federation of different cloud providers.

Being bound to one cloud provider has the threat of single point of failure. To provide high availability, the best solution is to leverage multiple cloud computing providers. Another issue is the data lock-in problem. Every cloud provider has its own proprietary API and this makes it difficult for customer to move their data and services from one cloud provider to another. Cloud users are also worried about the security of the confidential data and the ”audit-ability” of their data [40]. For some services, such as video-on-demand, the distance of the service provider to the user is important to meet the Quality of Service (QoS). A single cloud provider may not have data-centers to cover the users in all geographical locations.

Cloud federation can make small cloud providers to share their resources, so when one cloud is out of resource it can leverage from the available resources of other providers. It can provide an elastic pool of resources between multiple cloud providers in which the participants can join and leave dynamically. Indeed, ease of switching between different clouds makes the competition between cloud providers higher and results a better and more economic services for the end user.
C.4.2.2. Use cases in a community cloud

A community cloud is build to provide cloud solutions for users of a community. It can be build of multiple small and medium size clouds sharing their free resources. As the number of cloud providers increases, it will be infeasible to manually manage and control resources located on different cloud providers. Indeed, some cloud providers may run out of resources and cannot provide the whole client’s demands. A federation solution can provide a unified access point to manage and monitor those resources.

Elasticity in a community cloud is a critical requirement. Multiple cloud providers may join or leave the community. The federation solution can support this feature by creating a pool of resources on top of all the clouds in the community.

In a community cloud every small cloud is located in a different geographical location. For the services, such as social media streaming applications, a low response latency is vital. The ideal solution would be a geo-distributed federated cloud which can scale the application over multiple clouds to satisfy the quality of service.

C.4.3. Requirements for Community Clouds Federation

1. **Federated Identity Management:** One of the requirements of a cloud federation solution is the user management. The solution should be able to create an identity for users in order to authenticate them in the all participant clouds. This means the user can access to multiple cloud providers through authentication using a single identity and through one interface. Indeed, It should be possible to designate different access levels to different users and to determine what resources the users are authorized to access. The federation solution should also enable to define the amount of resources can be consumed by every user.

2. **Resource Provisioning:** Resource provisioning is the management of cloud resources such as computing, network and storage to enable application service providers to effectively provide and deliver services to customers. A federation solution can provide a pool of resources from the resources of participant clouds and to assign them effectively to the services. Buyya et al. mentioned some research issues [41], among them are application service behavior prediction and flexible mapping of services to resources. These two problems can be addressed by an efficient resource provisioning. The federation solution can support dynamic scaling and de-scaling of the services. The aforementioned feature creates another requirement which is prediction of the demands and behavior of the services. It is also important that the federation solution assigns resources to the services based on the QoS.

3. **Service Selection:** In a federated environment of multiple clouds where there are multiple service providers available, there is a requirement for a service selection tool to help the user to find the best service among them. User can define criteria such as the amount of resources (number of cores, bandwidth, memory and persistent disk), price and geo-location preferences and provide it as an input to the service selection tool. As a result, it can find the best service that fits the user criteria.

Facilitating IaaS service selection in a community cloud such as CLOMMUNITY is an important feature due to two major reasons: a) A community cloud is built up of multiple cloud providers with unknown backgrounds in terms of quality of service and the amount SLA violation. b) The resource quota of cloud providers are limited so they may not have enough resources for the user.
4. **Live Virtual Machine Migration**: Live migration of a virtual machine (VM) is the process of transferring a VM from one physical node to another without pausing the guest OS or hindering services running on the VM. Live migration can facilitate cloud users to move their VMs without any service shut-down. Cloud users need to move their instances for different reasons. During the infrastructure maintenance, a part of data-center and their running VMs will become unavailable. A cloud provider may also violate agreed SLAs or user’s criteria for selecting a provider may change over time.

5. **Network Support**: In a federation of different cloud providers, every cloud provider has its own network service and policies. Indeed, in such an environment VMs are allowed to migrate from one provider to another provider. These characteristics appeal a set of requirements for the virtual network service connecting the VMs. As it is mentioned by Hadas et al[42], network support should provide an elastic infrastructure agnostic network service with independent address space in order to support VM migration and connection across multiple clouds.

6. **Image Management**: Service providers (cloud users) package their service components into one or multiple virtual machine images and they regularly update these images with new release of their services. In a multi-cloud environment where every cloud provider has its own image repository due to their heterogeneous computing platforms supporting different image formats. It is a cumbersome task for the multi-cloud users to ensure the availability of the images on all the participant cloud providers and to keep the images updated. Existence of an image management service to facilitate the build and distribution process of these images to all the cloud providers can help the service providers to overcome the aforementioned problems.

**C.5. A Review of the Existing Multi Cloud Federation Solutions**

In this review, the solutions are categorized into four groups:

**Cloud Computing Platform**: Software to turn datacenters into IaaS clouds. They usually provide a set of toolkits to manage and provision resources such as CPU, network, memory and persistent disk for the users. OpenStack and OpenNebula are two open source examples of such software. Their initial purpose is not to support multiple cloud providers or to work in a multi-cloud environment. However, recently, there have been an interest in these communities to move toward multi-cloud adaptation and better integration with other cloud providers. For example, in OpenStack community a blueprint for a federated identity management is proposed which is planned to be shipped in the next distribution of OpenStack Icehouse. Indeed, the appearance of standard specifications such as Open Cloud Computing Interface (OCCI), open specifications and APIs mainly for IaaS services, Open Virtualization Format (OVF) which is an open standard for packaging software to be run in VMs and Cloud Data Management Interface (CDMI) which is about data management.

**Cloud Management Platform**: Provides a web based user interface to manage and monitor resources on multiple clouds. It facilitates the design and deployment of the services running on clouds and provides monitoring features to control and administrator those services from a single place.

**Cloud Broker**: A service that acts as an intermediary between users and cloud providers. It usually provides a language for users to define their service specifications and then negotiates on behalf of users to provide the resources.

**Multi-Cloud API**: Provides a general programming interfaces to use services of heterogeneous cloud computing platforms. Multi-cloud API can be used to develop application and services which needs to work with heterogeneous clouds.
In Table C.1, the selected projects, to what category they belong and whether they are open source or not are demonstrated.

In a multi-cloud environment, the participant cloud providers could have different cloud computing platforms. In Table C.2, it is shown which of the open source cloud platforms are supported by each of the cloud brokers, cloud management platforms and cloud APIs.

We reviewed and analyzed the cloud management platforms and the cloud brokers based on what and how they solve each of the requirements for a community cloud federation. Table C.3 shows the summary of the study.
C.5. Existing Multi Cloud Federation Solutions

C.5.1. Contrail

The federation architecture proposed in Contrail project is a three layers architecture which exploits cloud providers based on Contrail cloud infrastructure and commercial cloud infrastructures[39]. The top most layer is the interface layer which can provide multiple access points. The mid layer is called core layer, responsible to fulfill functional and nonfunctional requirements. And the bottom layer, adapters, retrieves information and operate on different cloud providers.

SLAs in Contrail are defined in standard OVF format without any need for extensions[43]. SLA management model in Contrail is based on a multi-level negotiation of SLAs. The negotiation can be done in an arbitrary number of rounds, in which user negotiates a SLA with the Contrail federation in the top layer and then federation layer negotiates with several providers to select the best one. At each round the user receives the best SLA offer for the given application. The comparison, in federation SLA manager, is defined by comparing the distance between the SLA offers and the user’s criteria mapped in a multidimensional space where each dimension is a term defined in the SLA.

The images are stored in a Global Autonomous File System based on the XtreemOS which transfers the images to the cloud provider upon the request[44].

The applications are monitored periodically and the collected data are stored in the VEP database for on-demand retrieval. Every application’s statistical data is linked to the application by the Universally Unique Identifier assigned to the application by Contrail federation. Their design of identity management allows multiple Federation Access Points which share data through replicated databases[45]. Users can authenticate to the federation layer either through a web browser or through a non-browser based client. The web browser based authentication is possible either by Federation identity username and password or in a single sign-on method using an external Identity Provider called OpenID. The non-browser based client authentication is by means of a short-lived End Entity Certificate which is obtained from an online Certificate Authority web service associated with the Federation Identity Provider. The authorization developed in Contrail is based on Usage Control (UCON) model.

C.5.2. CompatibleOne

CompatibleOne is an on going cloud broker with the aim of providing an interoperable middleware for the description and federation of different cloud providers. It includes two major components. One is an object based model to describe cloud resource and the other one is an execution platform to provide resources based on the model. Using the model, user can describe its resource specifications and then execution platform will provide resources after validation of the model. The model enables user to describe resources both for Infrastructure-as-a-service and Platform-as-a-service. CompatibleOne supports a good range of private and public clouds including OpenStack and OpenNebula.

CompatibleOne’s Security Service is responsible for the identity management. It handles authentication based on a username and password created beforehand for a service provider/consumer and after by the first authentication they receive a token which should be provided for the subsequent operations.

CompatibleOne provides a network solution through a network service called CompatibleOne Network Services[46]. It manages Cloud Networking Gateways (CNG) which interconnect virtual machines running on multiple heterogeneous cloud providers. CNGs are packaged as a VM which run inside each of the cloud providers. A CNG can be used to connect intra cloud VMs and save public IP addresses through NATing. Indeed, it can connect inter-cloud VMs through VPN tunnel created between CNGs.
C.5.3. Scalr

Scalr is an open-source cloud management platform which allows to automate the deployment of services on multiple clouds. We have set up Scalr in our testbed, and more details about Scalr and our deployment are discussed in Section C.2.

C.5.4. Commercial Closed Source Projects

We also reviewed some of the commercial closed source projects providing multi-cloud solutions. This brings us a good vision on the feature sets that each of the contributors in the multi-cloud area have been providing.

RightScale\textsuperscript{7} and ECmanaged\textsuperscript{8} (see section C.3) are two cloud management systems provided as a Software as a Service model. They provide management of multiple clouds through a single dashboard. A big advantage of RightScale over the other Cloud Management Systems is its Network Manager. The network manager provides an abstraction of the network resources across cloud providers. It is equipped with a visualization tool that enables users to view the configuration of all networks across all their clouds in one place.

C.5.5. Conclusion

In this review, we grouped multi-cloud solutions into four categories of cloud computing platform, cloud management platform, cloud broker and APIs. We chose some of the important projects from each category and compared them regarding the features they provide to satisfy the requirements for the community cloud federation.

Contrail project addresses all the requirements for federating community clouds but it does not support OpenStack computing platform which is widely used. CompatibleOne supports OpenStack and is designed to address majority of the requirements except VM migration. However, CompatibleOne is in the development phase. The existing cloud management platforms are mainly to facilitate the cloud administrators to manage resources in multiple clouds by providing a dashboard. They mainly provide the identity management and resource provisioning.

\textsuperscript{7}www.rightscale.com/  
\textsuperscript{8}www.ecmanaged.com
Bibliography


[32] J. Rosenberg *et al.*, “Stun - simple traversal of user datagram protocol (udp) through network address translators (nats),” 2003. 4.4.1


Licence

The CLOMUNITY project, December 2014, CLOMUNITY-201412-D2.3:

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