Project acronym:	**PrEstoCloud**
Project full name:	**Proactive Cloud Resources Management at the Edge for efficient Real-Time Big Data Processing**
Grant agreement number:	**732339**

D3.9 Spatiotemporal Processing Capabilities - Iteration 1

Deliverable Editor:	**Panagiotis Gouvas (UBITECH)**
Other contributors:	**Kostas Theodosiou(UBITECH), Panagiotis Parthenis(UBITECH), Giannis Ledakis (UBITECH)**
Deliverable Reviewers:	**Quentin Jacquemart & Guillaume Urvoy-Keller (CNRS), Yevgeniya Sulema (LiveU)**
Deliverable due date:	**30/4/2018**
Submission date:	**26/6/2018**
Distribution level:	**Public**
Version:	**1.0**

This document is part of a research project funded by the Horizon 2020 Framework Programme of the European Union

Change Log

Version	Date	Amended by	Changes
0.1	26/3/2018	Panagiotis Gouvas	Table of Contents
0.2	3/4/2018	Kostas Theodosiou	Introduction
0.3	23/4/2018	Panagiotis Gouvas, Giannis Ledakis	Chapter 2 – PrEstoCloud requirements for temporal storage
0.4	15/5/2018	Panagiotis Gouvas, Kostas Theodosiou	Chapter 2 – Overview of DHTs & Problem Statement on maintaining a temporal storage in Mesh
0.5	26/5/2018	Panagiotis Gouvas, Kostas Theodosiou	Chapter 3 – State of the art analysis
0.6	13/6/2018	Panagiotis Gouvas	Chapter 4 – Analysis of PrEstoCloud Stack
0.7	15/6/2018	Panagiotis Gouvas	Conclusions, Executive Summary, Review Version
0.8	19/6/2018	Panagiotis Gouvas	Incorporation of 1st Reviewer Comments
0.9	22/6/2018	Panagiotis Gouvas	Incorporation of 2nd Reviewer Comments
1.0	26/6/2018	Panagiotis Gouvas	Final Version

Table of Contents

List of Tables

List of Figures

List of Lists

No table of figures entries found.

List of Abbreviations

The following table presents the acronyms used in the deliverable.

Table 1 – List of Acronyms

Abbreviation	*Description*
CPU	Central Processing Unit
DHT	Distributed Hash Table
DC resource	Data Center resource
GPU	Graphics Processing Unit
HDA	Highly Distributed Applications
MCC	Mobile Cloud Computing
MEC	Mobile Edge Computing
OS	Operating System
PDS	PrEstoCloud Device Stack
PSTL	PrEstoCloud Spatio Temporal Library
TOSCA	Topology and Orchestration Specification for Cloud Applications
UAV	Unmanned Aerial Vehicle
VM	Virtual Machine
XML	Extensible Markup Language

Executive Summary

This deliverable reports on the work performed under the Task 3.5 which aims at the development of a Spatio Temporal library that can be used for persistence of data during computations performed at the edge. More specifically, when performing distributed computations, there is a need for reading and writing to a data storage structure which is accessible by all compute nodes. This data structure must expose an API (i.e. read/write functional primitives) which will be used by the business logic of the computations per se. In case the computations are performed in a data center there are many data structures that can be used. In fact, most of the big data frameworks depend on underlying storage engines (e.g. HDFS) in order to handle immutable partition collections such as Resilient Data Sets in Spark.

Such data storage structures operate on highly stable data centers and rely on preconfigured redundancy elements that are placed by DevOps. Furthermore, the communication of the big data workers with the storage engine is extremely efficient since the network latency between the workers and the storage engine is minimal (less than 1ms in some cases). Finally, these data structures are able, by design, to handle parallel reads and writes by independent workers. However, **such structures cannot be used in the case of edge computations**. In case of a distributed computation that is performed in the edge part of the network the operational prerequisites of these data structures are totally invalidated. Instead of stable Data Center resources, the operational environment consists of resource limited devices that formulate temporal connections using mesh connectivity principles. Such connections can be established or broken at any time based on the mobility profile of the edge devices. One possible solution regarding the lack of existence of such a structure is to offload all persistence requests to the backhaul part of the network.

Unfortunately, this solution is not viable because of many reasons. First, this solution would assume that edge resources are continuously connected to DC resources which is not the case in general. Furthermore, the connectivity delay that would be paid as a penalty of the offloading process would raise a significant overhead to the computational task that would interact with the storage (even in a good case 50ms cannot be compared with 1ms). Finally, offloading data to the backhaul would result to unnecessary utilization of the network capacity.

An elegant solution to the problem of lack of storage relies on the usage of a Distributed Hash Table (hereinafter DHT). A DHT is a data structure that is created and maintained by many network participants. Such a structure is used widely today for temporal storage in extremely sophisticated frameworks such as Consul, etcd, etc. The challenge in the PrEstoCloud paradigm is that this structure must **operate on top of decentralized dynamic networks**. A network that consists of nodes that formulate **temporal connections with its adjacents** and in parallel do not rely on a central node for routing is called '**mesh network'**. The requirement of being operational on top of mesh networks raises many challenges such as **a)** how edge resources join seamlessly in a mesh using zero-touch configuration? **b)** how resources are globally addressable taking under consideration that mesh networks may split or join on demand? **c)** how parallel reads and writes are handled in a consistent way? and **d)** how PrEstoCloud computation tasks make use of the storage API?

All these requirements can be satisfied using a combination of protocols in a layered manner that are encapsulated in a so-called **PrEstoCloud Device Stack** (PDS). PDS is a software package that upon installation performs all appropriate configurations so that an edge resource is able to accept computational tasks and in parallel able to interact with the DHT that is member of. The cornerstone technologies that have been used in order to realize PDS are **a)** the **802.11s** protocol (layer-2 mesh networking protocol); **b)** the **CJDNS** IPv6-based routing protocol; **c)** the **JPPF** distributed computing framework, **d)** the **Docker runtime engine** and **e)** an implementation of the **Chord DHT** protocol (**Hazelcast**). It should be mentioned that the PDS is currently operational in Raspberry-based devices (ARM-based architecture). In the second phase of the project, additional architectures will be supported.

1. Introduction

1.1 Scope

The scope of this deliverable is to elaborate on the Spatiotemporal Processing capabilities that will be offered by the PrEstoCloud framework. These capabilities relate to the **need that PrEstoCloud Tasks have for storing and retrieving datasets** from other tasks **during execution of jobs on the edge devices**. The major **difficulty** that has to be tackled is that edge devices are **loosely connected** and thus they **cannot rely on an existing storage protocol** that is usable in reliable data centers. The **alternative** of sending and requesting data during a task execution to a datacenter is a priori unacceptable since it would **radically increase the delay** and the **traffic between the edge** (also addressed as fronthaul in the telecommunications jargon) **and the datacenter** (also addressed as backhaul). This problem is illustrated on the figure below (Figure 1.1).

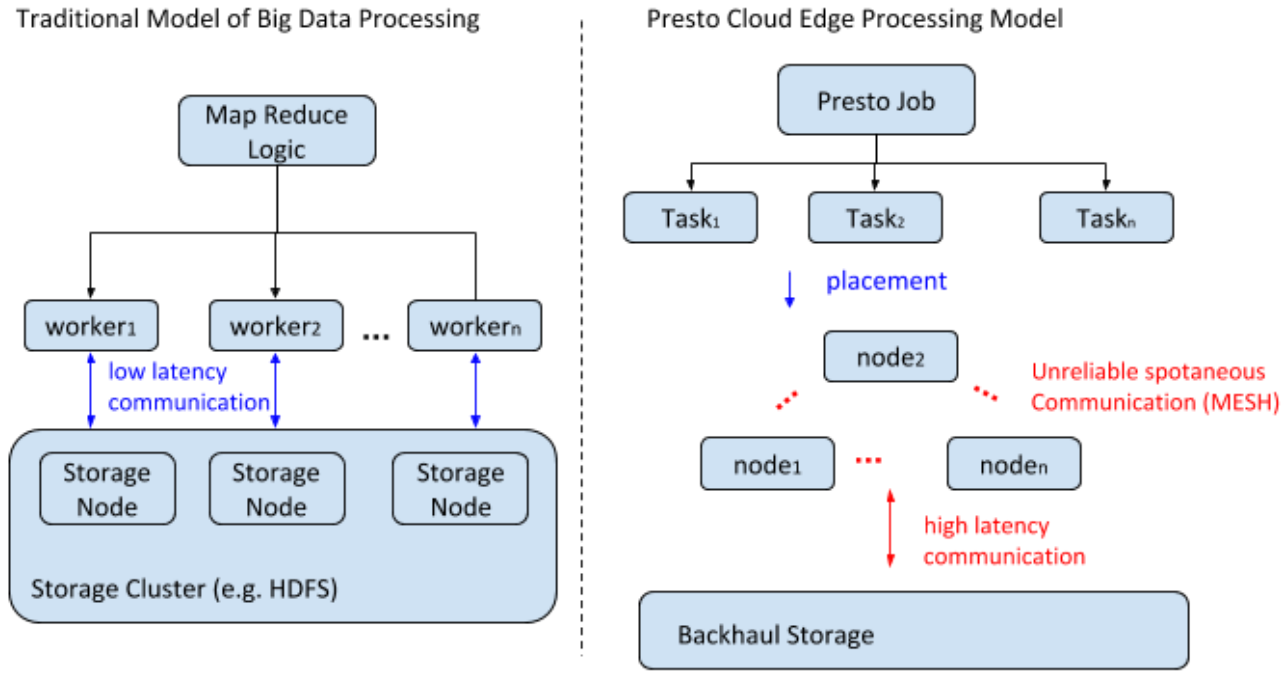

Figure 1.1 Lack of reliable low-latency storage during task execution on the Edge

In order to achieve **efficient** and **transactional** storage of data during PrEstoCloud Task execution **a PrEstoCloud Spatio Temporal Library** (hereinafter **PSTL**) has been developed which can be used unconditionally by any PrEstoCloud Task that is executed on an Edge Device. The storage library is responsible for storing and retrieving key-value sets with consistency guarantees irrelevant to the dynamicity of the environment. To do so, a layered approach will be followed which will be elaborated in detail.

1.2 Relation to PrEstoCloud Architecture

As it can be illustrated on Figure 1.2 (see Deliverable D2.3[1]), PSTL library is positioned on the device layer. More specifically, the library is part of a **complex device stack** that is provided during the onboarding of a device to a Mesh network. The **PrEstoCloud Device Stack** (hereinafter **PDS**) is responsible to undertake many functionalities such as **a)** layer-2 connectivity on a mesh network; **b)** layer-3 IP address autoconfiguration (avoiding static IP configuration); **c)** monitoring; **d)** the installation of the management agent (i.e. onloading/offloading Agent) and **e)** the initiation of PSTL.

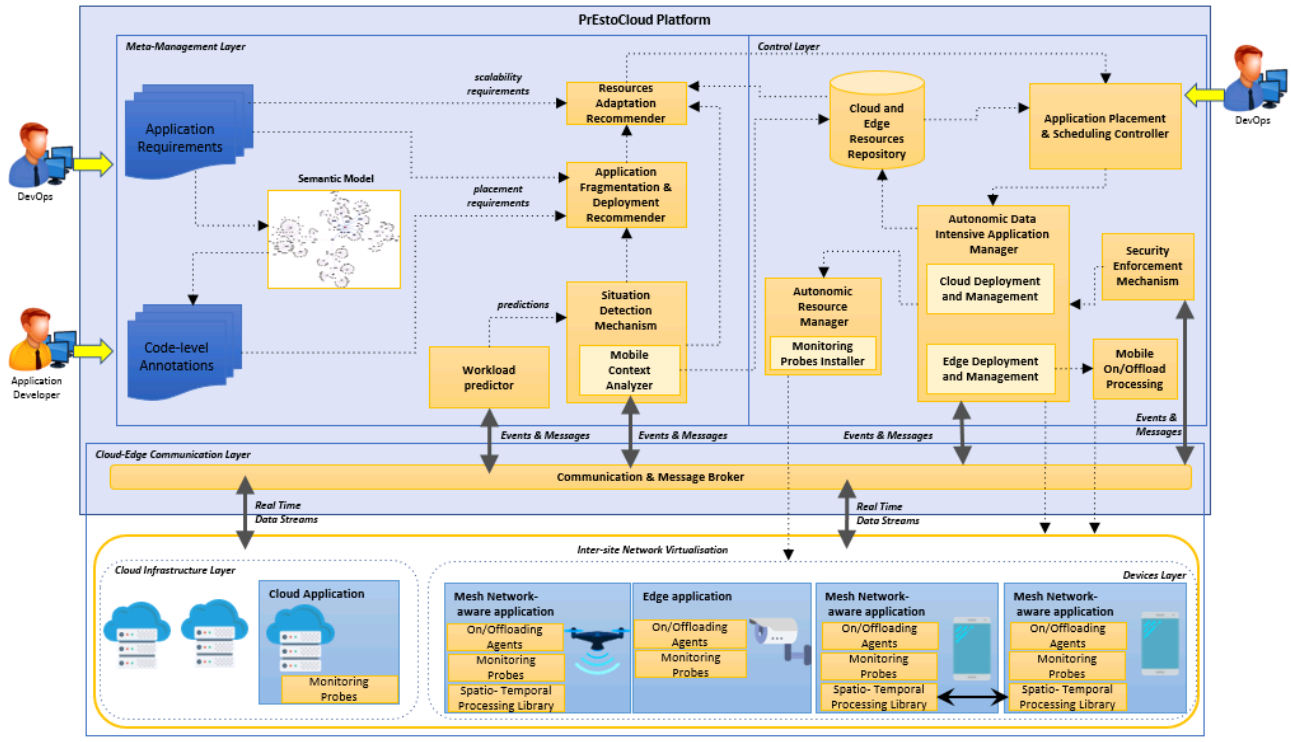

Figure 1.2 The logical positioning of the PSTL library

Although the purpose of the deliverable is to shed light on the Spatio Temporal Library the entire PrEstoCloud Device Stack will be briefly explained in order to achieve maximum comprehension from the reader.

1.3 Structure

The deliverable is structured as follows:

- Chapter 2 will elaborate on the problem statement of "maintaining a consistent temporal storage structure on top of Mesh Topologies". Towards these lines, the difficulties for building and maintaining such a structure will be analyzed. The cornerstone technology of such structures is the usage of **Distributed Hash Tables** (hereinafter **DHT**s)
- Chapter 3 will provide a state of the art analysis regarding the problem that has been raised above. More specifically, existing techniques **for building and maintaining DHTs on top of structured and unstructured networks is provided**.
- Chapter 4 is dedicated to the analysis of the **PrEstoCloud Device Stack (PDS)**. As it will be explained, part of the PDS is the PSTL per se. Yet the entire stack will be elaborated.
- Chapter 5 **concludes** this deliverable.

2. The problem of low-latency storage in Mesh Topologies

2.1 Temporal Data Storage during PrEstoCloud Task execution

The aim of the PrEstoCloud project is to deliver an efficient **real-time stream processing framework** tailored for edge resources. Based on this, one of the most critical aspects is the selection of an appropriate **distributed computing framework** which will be extended in order to include advanced resource management policies. In the frame of PrEstoCloud, the **JPPF framework**[1] has been selected based on **two main reasons**. The first has to do with its ability to be used on **resource limited** devices and second relates to its **ability to be able to dynamically expand and shrink** its processing nodes (workers) in a fault-tolerant way.

According to the JPPF terminology, which is de-facto adopted in PrEstoCloud, each processing Job is split in several Tasks that can be executed in parallel since they have distinct execution contexts i.e. non-correlated inputs. These tasks are dynamically allocated to cluster nodes which are part of the edge resources. In the frame of PrEstoCloud the JPPF framework had to be extended in various ways since the **Task-allocation** policy has to consult the **load prediction** module. During the execution of a task there is a need for persistence in order for the task to store intermediate or final results that are accessible/observable by all tasks that belong to the same job. This flow is depicted on the figure below.

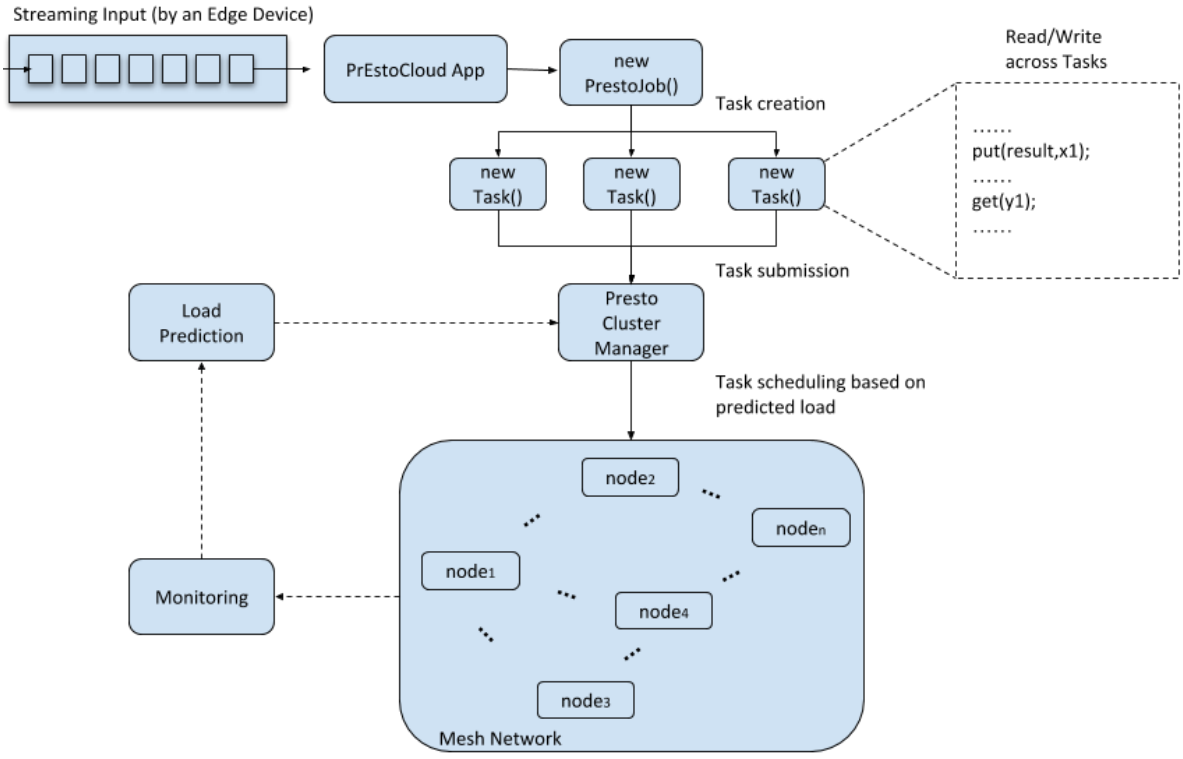

Figure 2.1 Lack of reliable low-latency storage during task execution on the Edge

[1] https://www.jppf.org

As illustrated, a PrEstoCloud application is processing a streaming input and based on a specific business logic it performs **segmentation of the input** in order to process it in parallel. The segmentation business logic is performed through the **extension of a base class** which is addressed **PrEstoJob.**

Processing per se is performed by another class which must extend the class **PrEstoTask**. In other words, **PrEstoJob** and **PrEstoTask** are extended JPPF classes and one **PrEstoJob** consists of multiple **PrEstoTasks**. During the execution of the **PrEstoTasks** a developer may store and retrieve results that should be queryable by other tasks. Such persistence storage should offer pure decentralization, scalability, transactional guarantees and fault-tolerance. To achieve these requirements the temporal storage will rely on a **reference implementation of a Distributed Hash Table**. As we will see, such a structure has inherent distribution and **scalability** properties; yet it is rather difficult to be maintained within a loosely coupled topology of edge devices. Such a network topology where nodes are temporarily connected without any form of centralized management is addressed as Mesh Network.

2.2 DHTs at a glance

A Distributed Hash Table is a class of decentralized distributed system that provides a lookup service similar to a hash table; (key, value) pairs are stored in the DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a **minimal amount of disruption.** This allows DHTs to **scale** to extremely large numbers of nodes and to handle continuous node arrivals, departures, and failures.

DHTs form an infrastructure that can be used to build more complex services, such as distributed file systems, peer-to-peer file sharing and content distribution systems, cooperative web caching, multicast, anycast, domain name services, and instant messaging, social applications etc. Notable distributed networks that use DHTs include BitTorrent's distributed tracker[2], the Kad network, YaCy[3], and the Coral Content Distribution Network [2].

2.2.1 History

Research on DHT was originally motivated, in part, by peer-to-peer systems such as Napster[4], Gnutella[5], and Freenet[6], which took advantage of resources distributed across the Internet to provide useful applications. In particular, they took advantage of increased bandwidth and hard disk capacity to provide a file sharing service. These systems differed in how they found the data their peers contained. Napster had a central index server: each node, upon joining, would send a list of locally held files to the server, which would perform searches and refer the 'querier' to the nodes that held the results. This central component left the system vulnerable to attacks and lawsuits.

Gnutella and similar networks moved to a flooding query model—in essence, each search would result in a message being broadcasted to every other machine in the network. While avoiding a single

[2]http://bitconjurer.org/BitTorrent

[3]http://yacy.net/Technology.html

[4]http://www.napster.com

[5] Gnutella Protocol Specification http://wiki.limewire.org/index.php?title=GDF

[6]http://freenetproject.org/

point of failure, this method was significantly less efficient than Napster. Moreover, Freenet was also fully distributed, but employed a heuristic key-based routing in which each file was associated with a key, and files with similar keys tended to cluster on a similar set of nodes. Queries were likely to be routed through the network to such a cluster without needing to visit many peers. However, Freenet did not guarantee that data would be found.

On the other hand, Distributed Hash Tables use a more structured key-based routing in order to attain both the decentralization of Gnutella and Freenet, and the efficiency and guaranteed results of Napster. One drawback is that like Freenet, DHTs only directly support exact-match search, rather than keyword search, although such functionality can be layered on top of a DHT.

From 2001 to 2004, six systems—CAN [3], Chord [4], Pastry [5], Tapestry [6], Kademlia [7] and Viceroy [8] — ignited DHTs as a popular research topic, and this area of research remains active. Outside academia, DHT technology has been adopted as a component of BitTorrent and in the Coral Content Distribution Network.

2.2.2 DHT Properties

DHTs characteristically feature the following properties:

- ***Decentralization***: the nodes collectively form the system without any central coordination.
- ***Scalability***: the system should function efficiently even with thousands or millions of nodes.
- ***Fault tolerance***: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.

A key technique used to achieve these goals is that any node needs to coordinate with only a few other nodes in the system – most commonly, O(log n) of the n participants – so that only a limited amount of work needs to be done for each change in membership.

Some DHT designs seek to be secure against malicious participants [9] and to allow participants to remain anonymous, though this is less common than in many other peer-to-peer (especially file sharing) systems. Finally, DHTs also deal with more traditional distributed systems issues such as load balancing, data integrity, and performance (in particular, ensuring that operations such as routing and data storage or retrieval complete quickly).

2.2.3 DHT Principles

The structure of a DHT can be decomposed into several main components. The foundation is an *abstract keyspace*. A *keyspace partitioning* scheme splits ownership of this keyspace among the participating nodes. A logicalnetwork then, connects the nodes, allowing them to find the owner of any given key in the keyspace. This logical network is addressed as **overlay network**.

Once these components are in place, a typical use of the DHT for storage and retrieval might proceed as follows. Suppose the keyspace is the set of 160-bit strings. To store a file with given *filename* and *data* in the DHT, the *SHA-1* [7] hash of filename is generated, producing a 160-bit *key k*, and a message *put(k,data)* is sent to any node participating in the DHT. The message is forwarded from node to node through the overlay network until it reaches the single node responsible for *key k* as specified by the *keyspace partitioning*. The appropriate node stores the key and the data. Any other client can retrieve the contents of the file by again hashing filename to produce *k* and asking any DHT node to

[7] https://en.wikipedia.org/wiki/SHA-1

find the data associated with *k* with a message *get(k)*. The message will again be routed through the overlay to the node responsible for *k*, which will reply with the stored data.

These principles are depicted at Figure 2.2 where the inner circle represents the physical topology of the mobile nodes while the outer circle represents the DHT overlay. The general idea is that every node that is registered to the DHT is able to publish and retrieve data. Please note that the same Hash function that is used for node registration in the overlay is used for data registration. This is very crucial since it is related to the keyspace partitioning.

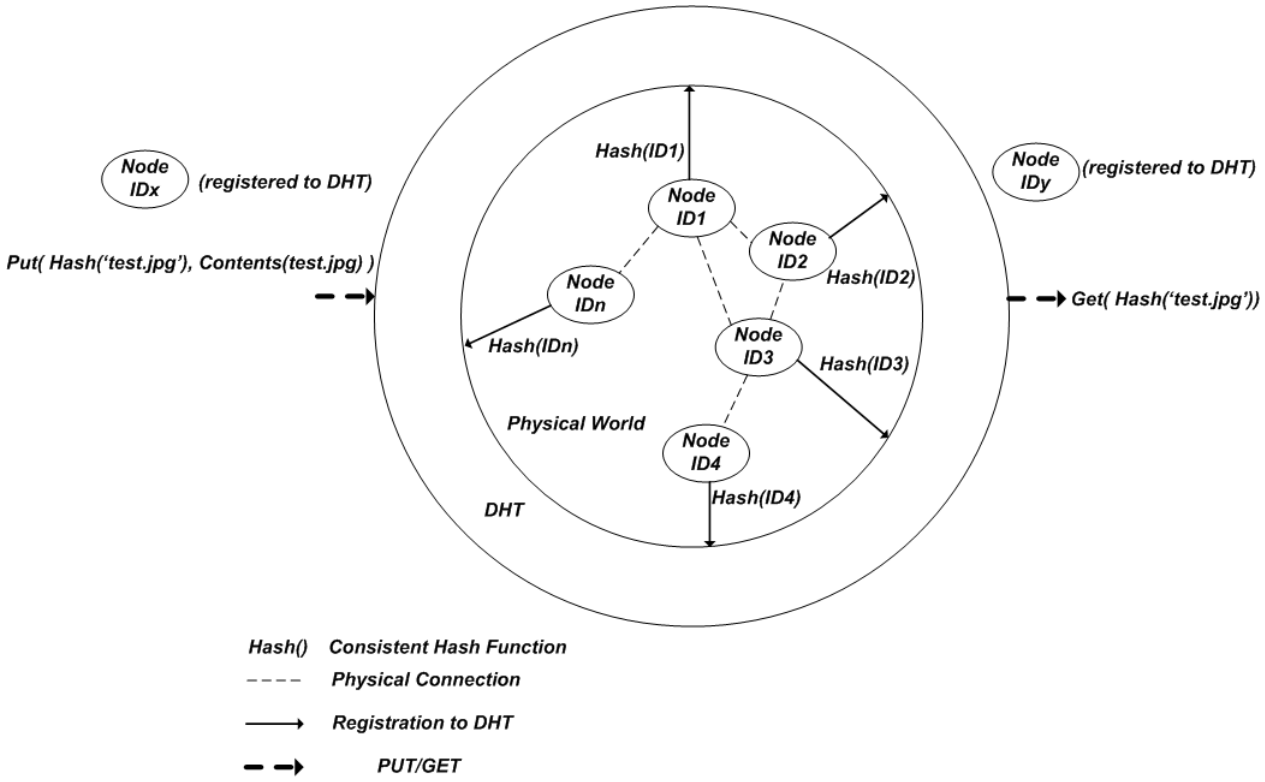

Figure 2.2 DHT overview

The *keyspace partitioning* and *overlay network* components are described below with the goal of capturing the principal ideas common to most DHTs; many designs differ in the details.

2.2.3.1 Keyspace partitioning

Most DHTs use some variant of *consistent hashings* [10] **,** to map keys to nodes. This technique employs a function $\boldsymbol{\delta(k_1,k_2)}$ which defines an abstract notion of the distance from key $\boldsymbol{k_1}$ to key $\boldsymbol{k_2}$, which is unrelated to geographical distance or network latency. Each node is assigned a single key called its identifier (***ID***). A node with ID $\boldsymbol{i_x}$ *owns all the keys* k_m for which $\boldsymbol{i_x}$ is the closest ***ID***, measured according to $\boldsymbol{\delta(k_m,i_n)}$.

In order to make keyspace partitioning clearer, let us consider an example from a real DHT implementation. The Chord DHT treats keys **as points on a circle**, and $\boldsymbol{\delta(k_1,k_2)}$ is the distance traveling clockwise around the circle from $\boldsymbol{k_1}$ to $\boldsymbol{k_2}$. Thus, the circular keyspace is split into contiguous segments whose endpoints are the node identifiers. If $\boldsymbol{i_1}$ and $\boldsymbol{i_2}$ are two adjacent ***ID***s, then the node with ID $\boldsymbol{i_2}$ owns all the keys that fall between $\boldsymbol{i_1}$ and $\boldsymbol{i_2}$. This is depicted in Figure 2.3 where a Chord DHT is bootstrapped. The DHT is configured to have replication factor equals to two. This practically means that every key-value pair that is assigned to the node-responsible is automatically assigned to the next two successors in the overlay. So, if a key-value pair with key:K is stored (e.g. by node-F) to the

DHT and A<K<B then the authoritative physical node that must store this pair is the one that has Hash(NodeID)=B. Because of the replication factor, nodes B,C and D store keys in the range of A-B.

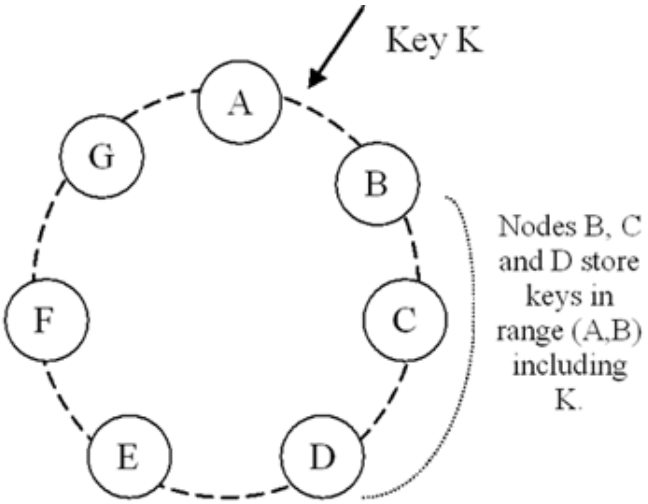

Figure 2.3 Chord's keyspace partitioning

Consistent hashing is based on mapping items to a real angle (or equivalently a point on the edge of a circle). Each of the available machines (or other storage buckets) is also pseudo-randomly mapped on to a series of angles around the circle. The bucket where each item should be stored is then chosen by selecting the next highest angle which an available bucket maps to. The result is that each bucket contains the resources mapping to an angle between itself and the next smallest angle.

If a bucket becomes unavailable (e.g. because the computer it resides on is not reachable), then, the angles it maps to will be removed. Requests for resources that would have mapped to each of those points now map to the next highest point. Since each bucket is associated with many pseudo-randomly distributed points, the resources that were held by that bucket will now map to many different buckets. The items that mapped to the lost bucket must be redistributed among the remaining ones, but values mapping to other buckets will still do so and do not need to be moved.

A similar process occurs when a bucket is added. By adding an angle, we make any resources between that and the next smallest angle map to the new bucket. These resources will no longer be associated with the previous bucket, and any value previously stored there will not be found by the selection method described above. The portion of the keys associated with each bucket can be altered by altering the number of angles that bucket maps to.

Consistent hashing has the essential property of minimal disturbance of the network during removal or addition of nodes since topology-changes affect only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected. On the other hand, in traditional hash tables addition or removal of one bucket causes nearly the remapping of the entire keyspace. Since any change in ownership typically corresponds to bandwidth-intensive movement of objects stored in the DHT from one node to another, minimizing such reorganization is required to efficiently support

high rates of churn (node arrival and failure). The most common consistent Hashing function is SHA-1.

2.2.3.2 Overlay Network

Each node maintains a set of links to other nodes (its neighbors or routing table). Together these links form the overlay network. A node picks its neighbors according to a certain structure, called the network's topology.

All DHT topologies share some variant of the most essential property: for any *key* ***k***, each node either knows a node ***ID*** which owns ***k*** or has a link to a node whose node ***ID*** is closer to ***k***, in terms of the keyspace distance defined above. It is then easy to route a message to the owner of any key ***k*** using the following greedy algorithm (that is not necessarily globally optimal): at each step, forward the message to the neighbor whose ***ID*** is closest to ***k***. When there is no such neighbor, then we must have arrived at the closest node, which is the owner of ***k*** as defined above. This style of routing is sometimes called key-based routing.

Beyond basic routing correctness, two important constraints on the topology are to guarantee that the maximum number of hops in any route (route length) is low, so that requests complete quickly; and that the maximum number of neighbors of any node (maximum node degree) is low, so that maintenance overhead is not excessive. Of course, having shorter routes requires higher maximum degree. Some common choices for maximum degree and route length are as follows, where ***n*** is the number of nodes in the DHT, using Big O notation (see Table 3-1):

- Degree O(1), route length O(*n*)
- Degree O(log*n*), route length O(log*n* / loglog*n*)
- Degree O(log*n*), route length O(log*n*)
- Degree O($\sqrt{n}$), route length O(1)

The third choice is the most common even though it is not quite optimal in terms of degree/route length tradeoff, because such topologies typically allow more flexibility in choice of neighbors. Many DHTs use that flexibility to pick neighbors which are close in terms of latency in the physical underlying network.

Maximum route length is closely related to diameter: the maximum number of hops in any shortest path between nodes. Clearly the network's route length is at least as large as its diameter, so DHTs are limited by the degree/diameter trade off which is fundamental in graph theory. Route length can be greater than diameter since the greedy routing algorithm may not find shortest paths.

2.2.3.3 Variations of diverse Implementations

The most notable differences encountered in practical instances of DHT implementations are discussed below. First of all, several real world DHTs use 128 bit or 160 bit keyspace. Furthermore, some real-world DHTs use hash functions other than SHA1. Additionally, in the real world the key ***k*** could be a hash of a file's content rather than a hash of a file's name, so that renaming of the file does not prevent users from finding it.

Moreover, some DHTs may also publish objects of different types. For example, key ***k*** could be node ***ID*** and associated data could describe how to contact this node. This flexibility allows publication of presence information and is often used in Instant Messaging applications, etc. In simplest case ***ID*** is just a random number which is directly used as key ***k*** (so in a 160-bit DHT ***ID*** will be a 160 bit number, usually randomly chosen). In some DHTs publishing of nodes ***ID***s is also used to optimize DHT operations.

Redundancy can be added to improve reliability. The ***(k,data)*** key pair can be stored in more than one node corresponding to the key. Usually, rather than selecting just one node, real world DHT

algorithms select ***i*** suitable nodes, with ***i*** being an implementation-specific parameter of the DHT. In some DHT designs, nodes agree to handle a certain keyspace range, the size of which may be chosen dynamically, rather than hard-coded.

Some advanced DHTs like Kademlia [11] perform iterative lookups through the DHT first, in order to select a set of suitable nodes and send put(***k***,***data***) messages only to those nodes, thus drastically reducing useless traffic, since published messages are only sent to nodes which seem suitable for storing the key ***k***; and iterative lookups cover just a small set of nodes rather than the entire DHT, reducing useless forwarding. In such DHTs forwarding of *put*(***k***,***data***) messages may only occur as part of a self-healing algorithm: if a target node receives a *put*(***k***,***data***) message but believes that k is out of its handled range and a closer node (in terms of DHT keyspace) is known, the message is forwarded to that node. Otherwise, data are indexed locally. This leads to a somewhat self-balancing DHT behavior. Of course, such an algorithm requires nodes to publish their presence data in the DHT so the iterative lookups can be performed.

2.3 Difficulties in maintaining a distributed key-value store in Mesh Environments

In chapter 2.1 the principles of DHTs and the merits that are derived from them were presented. However, DHTs are designed to operate on the Internet environment and not on a Mesh environment. More specifically, the assumptions that are made by the majority of DHT implementations which are met by the Internet environment are the following:

Efficient underlay routing & efficient connection establishment: e.g. assume that a *newnode* enters the overlay in a Chord DHT implementation between *node1* and *node2* (we remind the reader that Chords uses a circular overlay topology). Since new segments have been formulated i.e. *node1-newnode*&*newnode-node2*, specific key-value entries have to be transferred from *node2* to *newnode*. This transfer has to be done efficiently without a big communication start-up cost since a possible query for a specific key *k* in the structure may end-up at *node1*. If *newnode* is the corresponding node according to the consistent hashing function, *node1* will consult its finger table and will propagate the query to *newnode.* Consequently, *newnode* should be ready to respond to the query performer as quick as possible in order to prevent blocking issues.

Long lasting connections& stationary peers: assume that in the example described above, *newnode* enters and leaves the topology instantly. This would not be catastrophic for the DHT structure's coherency (since mechanisms that handle timeouts during key-value transfers exist) but it would generate a signaling cost both in the network and the overlay layer. This cost may be insignificant in fixed networks since a predefined routing scheme exists (see below) but in a Mesh Environment it would be too expensive.

Hierarchical routing scheme: Assume that in the example described above, *node2* is notified that specific *key-value* pairs must be transferred to *newnode*. However, *node2* is not aware on how *new node* will be reached (i.e. it is not a concern of node2), since Internet's hierarchical structure implies that the transferable key-value entry(ies) will be routed to *node2*'s default gateway and through TCP/IP entries will reach their destination.

Dedicated peers: As explained above, the task of overlay topology construction and maintenance is undertaken by low level mechanisms which in most of the cases are *centralized or semi-centralized*. Indicatively, such mechanisms are used in the Gnutella network [12] , in which topology creation may be achieved by using a pre-defined address-list of working nodes included within a compliant client or by using web caches of known nodes, a.k.a. Gnutella web caches. Similarly, Chord pre-assumes that nodes are ordered in a ring and are aware of their successor and predecessor in the overlay ring topology. Chord also relies on underlying mechanisms for the overlay network bootstrapping [13] . In fact, p2p protocols are able to react to topology changes (and automatically re-assign key-value pairs) but are not responsible for creating and maintaining the overlay topology.

This indicative issue, among others, is delegated to some super-peers that are also accessible due to the hierarchical routing scheme.

Stable network: Stable network refers to the backbone network and not to the endpoints that constitute the DHT peers. A stable backbone network ensures the efficiency of the hierarchical routing scheme and prevents island formulation. In order to clarify this issue, assume that in the example of a dynamic network topology - *newnode* is connected to *node2* through a unique valid route e.g. *newnode-nodex-nodey-node2* and the connection between *nodex* and *nodey* during key-value pair transfer is lost. This would result in the formulation of two islands consisting of i) island 1:*newnode*, *nodex* and all their local neighbors and ii) island 2: *nodey*, *node2* and all their local neighbors.

It is obvious that none of the aforementioned assumptions can be considered as granted in a Mesh environment. Nodes are not stationary and links are considered unreliable (see Figure 2.4). Moreover, there is no form of centralization and no dedicated peer in order to coordinate overlay construction. Consequently, the construction and maintenance of the overlay must be accomplished in an ad-hoc mode.

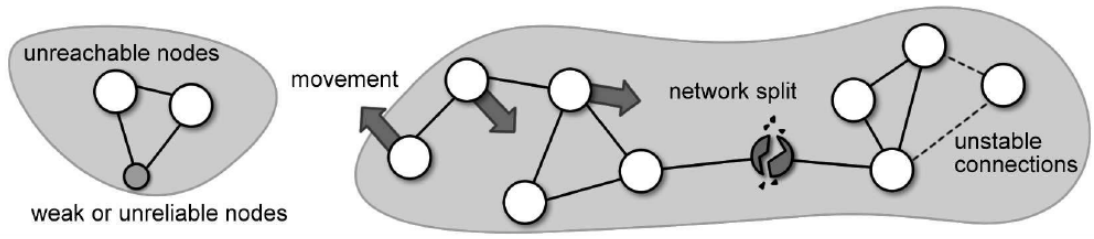

Figure 2.4 Mesh Networking Environment

Furthermore, low level routing among nodes is an important issue since no predefined routing scheme can be taken for granted. Therefore, alternative routing policies have to be adopted. Such routing protocols will be described at chapter 4.2.3.

Finally, the most critical problem is the lack of topological hierarchy which results in loosely temporarily created graphs. Such graphs (a.k.a. islands) of interconnected nodes may merge and split according to the current topology. This affects significantly the DHT operation. E.g. assume that in two existing separate islands, as depicted at Figure 2.5,the Chord protocol has bootstrapped. The mechanisms that have been used by Chord to bootstrap do not work in this example. According to Chord all participant nodes are directed to circular overlay topology for both islands. Assume that one node from one island committed *put(key_1,$value_x$)* and another node from the other island

committed in his circle/DHT *put(key$_1$,value$_y$)*. In the next step, the nodes that comprise the two islands come closer and formulate one big island (see Figure 2.6).

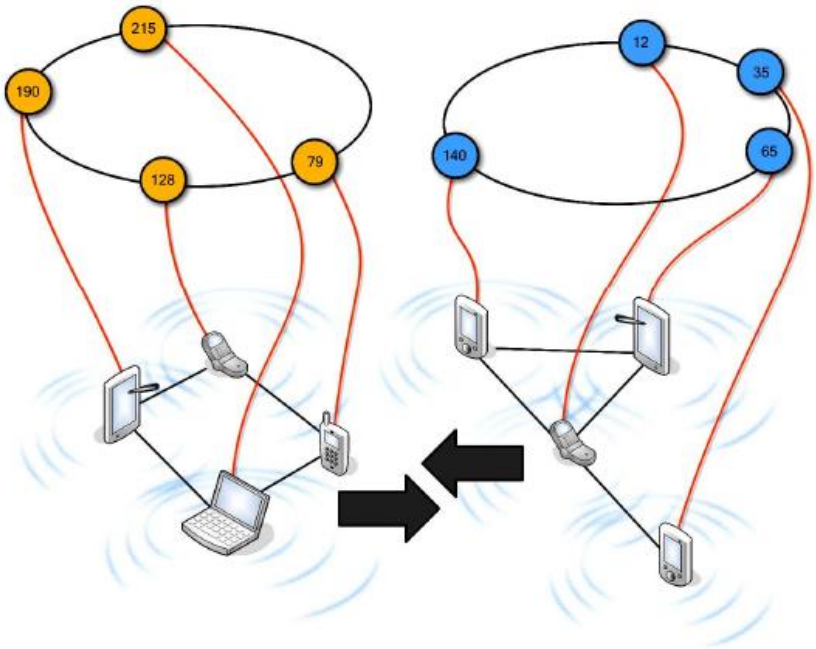

Figure 2.5 Two islands before they merge

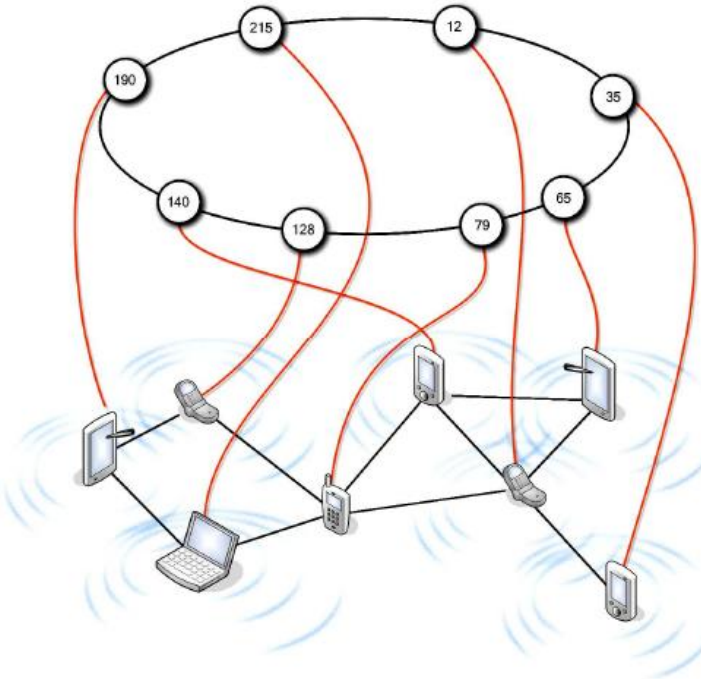

Figure 2.6 Two islands already merged

The goal that has to be achieved after merging is the efficient identification of the responsible node that maintains the value of *key$_1$* after the query of a third-party node. Normally, according to the DHT consistency principle, values should be merged and a possible *get(key$_1$)* should result in the

multi-value response *value$_x$-value$_y$*. However, a successful response pre-assumes that DHT signaling has been completed. On the other hand signaling is of minor importance for DHTs that are bootstrapped on fixed networks but it is considered too expensive for Mesh Environments.

Moreover, node-splitting is a scenario where DHT consistency is not guaranteed. E.g. assume that the big island of Figure 2.6 splits in two sub-islands of Figure 2.5. Suppose that a specific key$_1$-value$_x$ pair is published, the DHT protocol is Chord and the *HashOf(key$_1$)=130*. According to Chord algorithm the responsible node for storing this pair is the successor node of 130 i.e. node 140 in our case. It must be clarified here, that node 140 actually means node with *HashOf(NodeID)=140* where NodeID is something common among the nodes e.g. their MAC address. According to the splitting scenario, the physical topology is split and node 140 belongs to the 'blue' island. Now the question is what will the result be when node '79' from the orange islands commits *get(key$_1$)*? The answer is *null* since in the 'orange' island the authoritative node for providing the response is node 190 (i.e. the successor of 130). Node 190 has no info about key$_1$.

Similarly, assume that during *put(key$_1$,value$_x$)* (and before splitting up) there was a redundancy policy and the key-value pair was stored to its successor and to the next node (one node for redundancy). After the split-up, when node '79' from the orange islands commits *get(key$_1$),* the result would be value$_x$. So, redundancy is the key parameter as far as network splitting is concerned. Consequently, merging and splitting, in general, results in significant signaling cost on Mesh environments.

3. State of the art analysis on DHT structures on top of Mesh Topologies

3.1 Comparison of dominant DHT implementations

As already described, DHTs belong to the category of structured peer to peer systems according to which the location information of data-object is placed deterministically at a specific peer identified by the data object's unique key. DHT-based systems have the advantage of consistent assignment of data-objects to the nodes that constitute the network.

As it is clarified up to now, data objects are assigned unique identifiers called keys, chosen from the same identifier space. Keys are mapped by the overlay network protocol to a unique live peer in the overlay network. The P2P overlay networks support the scalable storage and retrieval of *{key,value}* pairs on the overlay network, as illustrated in Figure 3.1. Given a key, a store operation *put(key,value)*and a lookup retrieval operation *value=get(key)* can be invoked to store and retrieve the data object corresponding to the key, which involves routing requests to the peer corresponding to the key. Each peer maintains a small routing table consisting of its neighboring peers' NodeIDs and network addresses. In the case of MESH networks, centralized routing protocols cannot be utilized. Lookup queries or message routing requests are forwarded across overlay paths to peers in a progressive manner utilizing the NodeIDs that are closer to the key in the identifier space.

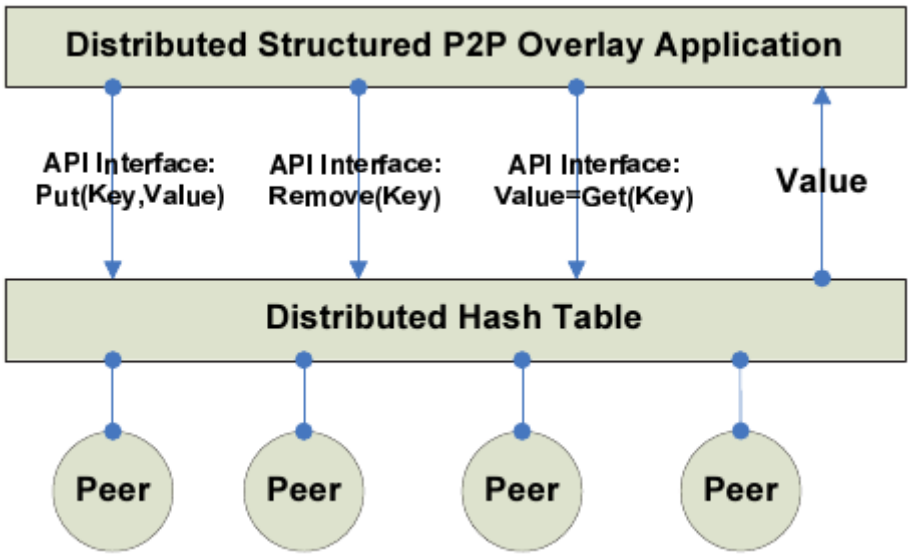

Figure 3.1 Application Interface for Structured DHT-based P2P Overlay Systems

Different DHT-based systems have different organization schemes for the data objects and their key space and routing strategies. In theory, DHT-based systems can guarantee that any data object can be located in *O(logN)* overlay hops on average, where *N* is the number of peers in the system. The underlying network path between two peers can be significantly different from the path on the DHT-based overlay network. Therefore, the lookup latency in DHT-based P2P overlay networks can be quite high and could adversely affect the performance of the applications running over it. [14] provides an elegant algorithm that achieves nearly optimal latency on graphs that exhibit power-law expansion [15] , at the same time, preserving the scalable routing properties of the DHT-based system.

DHT-based systems [10] are an important class of P2P routing infrastructures. They support the rapid development of a wide variety of Internet-scale applications ranging from distributed file and naming

systems to application-layer multicasting. They also enable scalable, wide-area retrieval of shared information. In 1999, Napster pioneered the idea of a peer-to- peer file sharing system supporting a centralized file search facility. It was the first system to recognize that requests for popular content need not to be sent to a central server but instead it could be handled by many peers that have the requested content. Such P2P file-sharing systems are self-scaling in that as more peers join the system, they add to the aggregate download capability. Napster achieved this self-scaling behavior by using a centralized search facility based on file lists provided by each peer, thus, it does not require much bandwidth for the centralized search. Such a system has the issue of a single point of failure due to the centralized search mechanism. However, a lawsuit filed by the Recording Industry Association of America (RIAA) forced Napster to shut down the file-sharing service of digital music — literally, its killer application.

However, the paradigm caught the imagination of platform providers and users alike. Gnutella is a decentralized system that distributes both search and downloads' capabilities, establishing an overlay network of peers. It is the first system that makes use of an Unstructured P2P overlay network. An Unstructured P2Psystem is composed of peers joining the network with some loose rules, without any prior knowledge of the topology. The network uses flooding as the mechanism to send queries across the overlay with a limited scope. When a peer receives the flood query, it sends a list of all content matching the query to the originating peer. While flooding-based techniques are effective for locating highly replicated items and are resilient to peers joining and leaving the system, they are poorly suited for locating rare items. Clearly this approach is not scalable as the load on each peer grows linearly with the total number of queries and the system size. Thus, Unstructured P2P networks face one basic problem: peers readily become overloaded, therefore, the system does not scale when handling a high rate of aggregate queries and sudden increase in system size.

Although Structured P2P networks can efficiently locate rare items since the key-based routing is scalable, they incur significantly higher overheads than Unstructured P2P networks for popular content. Consequently, over the Internet today, the decentralized Unstructured P2P overlay networks are more commonly used. However, there are recent efforts on Key-based Routing (KBR) API abstractions [16] that allow more application-specific functionality to be built over this common basic KBR API abstractions, and OpenHash (Open publicly accessible DHT service) [17] that allows the unification platform of providing developers with basic DHT service models that runs on a set of infrastructure hosts, to deploy DHT-based overlay applications without the burden of maintaining a DHT and with ease of use to spur the deployment of DHT-based applications.

In contrast, Unstructured P2P overlay systems are Ad-Hoc in nature, and do not present the possibilities of being unified under a common platform for application development. In Table 3-1, we will describe the key features of Structured P2P and Unstructured P2P overlay networks and their operational functionalities. After providing a basic understanding of the various overlays schemes in

these two classes, an evaluation of these schemes is provided [18] followed by some comparative results based on the following attributes:

- Decentralization: examine whether the overlay system is distributed.
- Architecture: describe the overlay system architecture with respect to its operation.
- Lookup Protocol: the lookup query protocol adopted by the overlay system.
- System Parameters: the required system parameters for the overlay system operation.
- Routing Performance: the lookup routing protocol performance in overlay routing.
- Routing State: the routing state and scalability of the overlay system.
- Peers Join and Leave: describe the behavior of the overlay system when churn and self-organization occurred.
- Security: look into the security vulnerabilities of overlay system.
- Reliability and Fault Resiliency: examine how robust the overlay system when subjected to faults.

Although all protocols that are discussed in Table 3-1 are candidate ones for being adopted in PrEstoCloud Cloud, the architectural simplicity of Chord along its good performance under mobility scenarios [19] urged us to select it as the cornerstone for implementation.

Table 3-1 Comparison of Structured P2P approaches

Algorithm Taxonomy	Structured P2P Overlay Network Comparisons					
	CAN	Chord	Tapestry	Pastry	Kademlia	Viceroy
Decentralization	DHT functionality on Internet-like scale					
Architecture	Multi-dimensional ID coordinate space.	Uni-directional and Circular NodeID space.	Plaxton-style global mesh network.	Plaxton-style global mesh network.	XOR metric for distance between points in the key space.	Butterfly network with connected ring of predecessor and successor links, data managed by servers.
Lookup Protocol	key,value pairs to map a point P in the coordinate space using uniform hash function.	Matching Key and NodeID.	Matching suffix in NodeID.	Matching Key and prefix in NodeID.	Matching Key and Node-ID based routing.	Routing through levels of tree until a peer is reached with no downlinks; vicinity search performed using ring and level ring links.
System Parameters	*N*-number of peers in network *d*-number of dimensions.	*N*-number of peers in network.	*N*-number of peers in network *B*-base of the chosen peer identifier.	*N*-number of peers in network *b*-number of bits ($B = 2^b$) used for the base of the chosen identifier.	*N*-number of peers in network *b*-number of bits (($B = 2^b$) of NodeID.	*N*-number of peers in network.
Routing Performance	$O(d.N^{\frac{1}{d}})$	$O(logN)$	$O(\log_B N)$	$O(\log_B N)$	$O(\log_B N) + c$ where c = small constant	$O(logN)$

Routing State	$2d$	$logN$	$\log_B N$	$B \log_B N + B \log_B N$	$B \log_B N + B$	$logN$
Peers Join/Leave	$2d$	$(logN)^2$	$\log_B N$	$\log_B N$	$\log_B N + c$ where c = small constant	$logN$
Security	Low level. Suffers from man-in-middle and Trojan attacks.					
Reliability/Fault Resiliency	Failure of peers will not cause network wide failure. Multiple peers responsible for each data item. On failures, application retries.	Failure of peers will not cause network-wide failure. Replicate data on multiple consecutive peers. On failures, application retries.	Failure of peers will not cause network-wide failure. Replicate data across multiple peers. Keep track of multiple paths	Failure of peers will not cause network-wide failure. Replicate data across multiple peers. Keep track of multiple paths to each peer.	Failure of peers will not cause network wide failure. Replicate data across multiple peers.	Failure of peers will not cause network wide failure. Load incurred by lookups routing evenly distributed among participating lookup servers.

3.2 Chord protocol in a more detailed view

Chord [4] uses consistent hashing [10] to assign keys to its peers. Consistent hashing is designed to let peers enter and leave the network with minimal interruption. This decentralized scheme tends to balance the load on the system, since each peer receives roughly the same number of keys, and there is little movement of keys when peers join and leave the system. In a steady state, for N peers in the system, each peer maintains routing state information for about only O(logN) other peers (N number of peers in the system). This may be efficient but performance degrades gracefully when that information is out-of-date.

The consistent hash functions assign peers and data keys an m-bit identifier using SHA-1 [20] as the base hash function. A peer's identifier is chosen by hashing the peer's IP address, while a key identifier is produced by hashing the data key. The length of the identifier *'m'* must be large enough to make the probability of keys hashing to the same identifier negligible. Identifiers are ordered on an identifier circle modulo *2m*.Key *k* is assigned to the first peer whose identifier is equal to or follows k in the identifier space. This peer is called the successor peer of key k, denoted by successor(k). If identifiers are represented as a circle of numbers from 0 to 2m − 1, then successor(k) is the first peer clockwise from k.

The identifier circle is termed as the Chord ring. To maintain consistent hashing mapping when a peer n joins the network, certain keys previously assigned to *n*'s successor now need to be reassigned to *n*. When peer *n* leaves the Chord system, all of its assigned keys are reassigned to n's successor. Therefore, peers join and leave the system with $(\log N)^2$ performance (i.e. exchanged messages). No other changes of keys assignment to peers need to occur. In Figure 3.2(adapted from [4]), the Chord ring is depicted with m = 6. This particular ring has ten peers and stores five keys. The successor of the identifier 10 is peer 14, so key 10 will be located at NodeID 14. Similarly, if a peer were to join with identifier 26, it would store the key with identifier 24 from the peer with identifier 32.

Each peer in the Chord ring needs to know how to contact its current successor peer on the identifier circle. Lookup queries involve the matching of key and NodeID. For a given identifier, queries could be applied around the circle via these successor pointers until they encounter a pair of peers that include the desired identifier; the second peer in the pair is the peer the query maps to. An example is presented in Figure 3.2, whereby peer 8 performs a lookup for key 54. Peer 8 invokes the find successor operation for this key, which eventually returns the successor of that key, i.e. peer 56. The query visits every peer on the circle between peer 8 and peer 56. The response is returned along the reverse of the path.

As m is the number of bits in the key/NodeID space, each peer n maintains a routing table with up to m entries, called the finger table. The i^{th} entry in the table at peer n contains the identity of the first peer s that it least 2^{i-1} positions after n on the identifier circle, i.e. s = successor(n + 2^{i-1}), where $1 \leq i \leq m$. Peer s is the i^{th} finger of peer n (n.finger[i]). A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant peer.

Figure 3.2 shows the finger table of peer 8, and the first finger entry for this peer points to peer 14, as the latter is the first peer that succeeds (8+20) mod 26 = 9. Similarly, the last finger of peer 8 points to peer 42, i.e. the first peer that succeeds (8 + 25) mod 26 = 40. In this way, peers store information about only a small number of other peers, and know more about peers closely following it on the identifier circle than other peers. Also, a peer's finger table does not contain enough information to directly determine the successor of

an arbitrary key k. For example, peer 8 cannot determine the successor of key 34 by itself, as successor of this key (peer 38) is not present in peer 8's finger table.

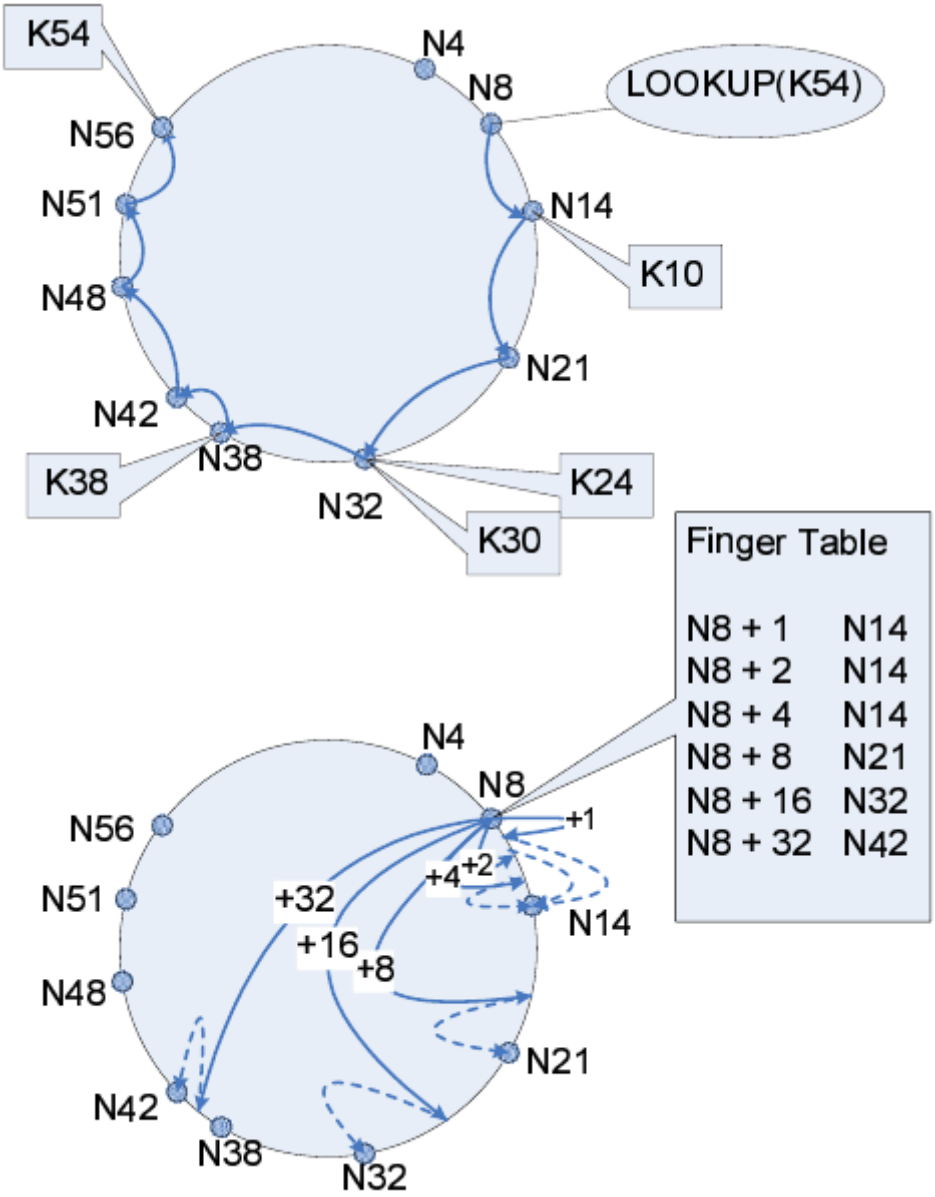

Figure 3.2 Chord Ring of 10 peers and 5 key-value pairs.

When a peer joins the system, the successor pointers of some peers need to be changed. It is important that the successor pointers are up to date at any time because the correctness of lookups is not guaranteed otherwise. The Chord protocol uses a stabilization protocol [4] running periodically in the background to update the successor pointers and the entries in the finger table. The correctness of the Chord protocol relies on the fact that each peer is aware of its successors. When peers fail, it is possible that a peer does not know its new successor and it has no chance to learn about it. To avoid this situation, peers maintain a successor list of size *r*, which contains the peer's first *r* successors.

When the successor peer does not respond, the peer simply contacts the next peer on its successor list. Assuming that peer failures occur with a probability p, the probability that every peer on the successor list will fail is p^r. Increasing *r* makes the system more robust. By tuning this parameter, any degree of robustness

with good reliability and fault resiliency may be achieved. The following applications are examples of how Chord could be used:

- Cooperative mirroring or Cooperative File System (CFS) [21], in which multiple providers of content cooperate to store and serve each others' data. Spreading the total load evenly over all participant hosts lowers the total cost of the system, since each participant needs to provide capacity only for the average load, not for the peak load. There are two layers in CFS. The DHash (Distributed Hash) layer performs block fetches for the peer, distributes the blocks among the servers, and maintains cached and replicated copies. The Chord layer distributed lookup system is used to locate the servers responsible for a block.
- Chord-based DNS [22] provides a lookup service, with host names as keys and IP addresses (and other host information) as values. Chord could provide a DNS-like service by hashing each host name to a key [10]. Chord-based DNS would require no special servers, while ordinary DNS systems rely on a set of special root servers. DNS also requires manual management of the routing information (DNS records) that allows clients to navigate the name server hierarchy; Chord automatically maintains the correctness of the analogous routing information. DNS only works well when host names are hierarchically structured to reflect administrative boundaries; Chord imposes no naming structure. DNS is specialized to the task of finding named hosts or services, while Chord can also be used to find data object values that are not tied to particular machines.

3.3 Chord protocol on top of a dynamic network topology

The PrEstoCloud proposed approach aims at the provision of a generic framework that will facilitate the design and development of autonomic and decentralized services in Mesh networks (see Figure-4.1). The introduction of the different layers of the proposed approach is necessary due to the need to address the following challenges: a) efficiently utilize available network resources in a dynamic environment, b) provide services independently from the underlying topology, c) ensure reliability of services in case of network topology changes and d) reduce the management complexity and increase flexibility to application developers. In order to address these challenges, autonomic functionalities have to be incorporated. The following self-* properties have been defined [23] and should be supported by an autonomic system: self-configuration, self-optimization, self-awareness and self- healing.

Existing protocols that satisfy partially the challenges described above were considered during the design of the proposed approach. There is no existing work on how to combine existing protocols for achieving autonomic service provisioning and how different protocols could interact using predefined interfaces. Taking into account these considerations, the proposed approach is focusing on a) defining concrete layering for enabling autonomic service provisioning in Mesh networks, b) specifying the discrete functionality of each layer and the interfaces between them and c) resolving conflicts between existing protocols, specifically in the field of the overlay topology construction.

The creation and maintenance of an overlay topology that logically interconnects all the participating nodes in the physical network is critical in our approach. Any node that connects to the ad-hoc network has to join to the overlay network. The overlay network is formulated during the topology stabilization phase in an autonomic manner and hides any details of the underlying physical infrastructure, e.g. link establishment or torn down, node failures, node mobility, etc. In case of multiple changes in the physical topology, the overlay network is able to adapt quickly to the new environment (re-stabilization). Furthermore, recovery from

failures can be easily achieved based on information that is available in the network. All these tasks are realized without the intervention of the network administrator.

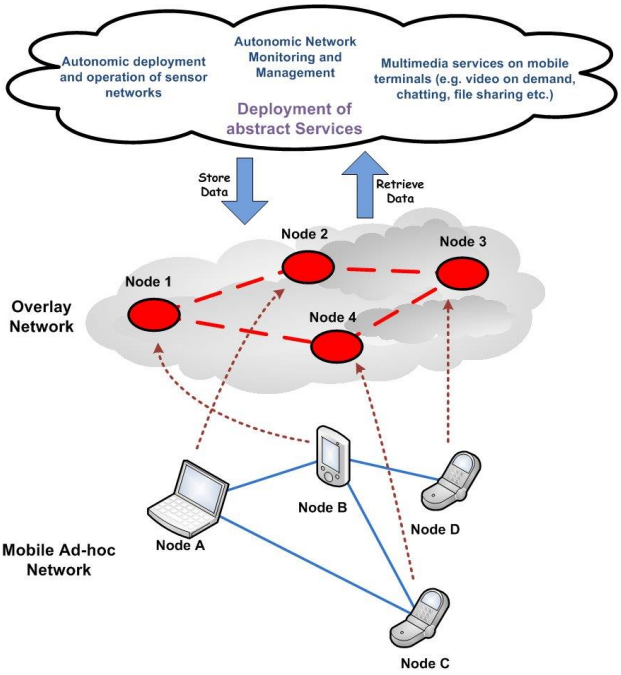

Figure 3.3 Services that rely on an operational DHT in dynamic environment

After the overlay network is established, participating nodes are able to store and retrieve data using typical p2p protocols. Every node that wishes to store a keyvalue- pair, or query a value based on a key, can achieve it by using a Distributed Hash Table (DHT) [24] that operates on-top of the overlay topology. In a similar way, several applications can be built taking under consideration the existence of a high level API put(key,value) and get(key) that would interact with a DHT protocol that operates on-top of a non-reliable Mesh environment.

Provided services are designed based on the assumption of collaboration and dissemination of information among the participating nodes. These services can be fully decentralized as data and functionality is allocated in different nodes at the overlay network. Some functions may be delegated to more than one nodes for higher reliability. In case of changes or failures, roles may be re-assigned autonomously and performance guarantees may be assured for the services provision.

We propose a four-layered scheme based on the functionality requirements imposed by the provided services and the underlying physical networking environment. As shown in Figure-4.2, the following four layers are defined; i) *Neighbor-to-Neighbor layer*, ii) *Routing layer*, iii) *Topology Maintenance layer*, and iv) *DHT layer*. Each layer has a discrete role, implements different mechanisms and specifies its messages types. The proposed layered approach is independent from the selection of p2p protocols, topology formulation

mechanisms and routing protocols. Therefore, any combination of different protocols may be selected and proper adaptations may be proposed.

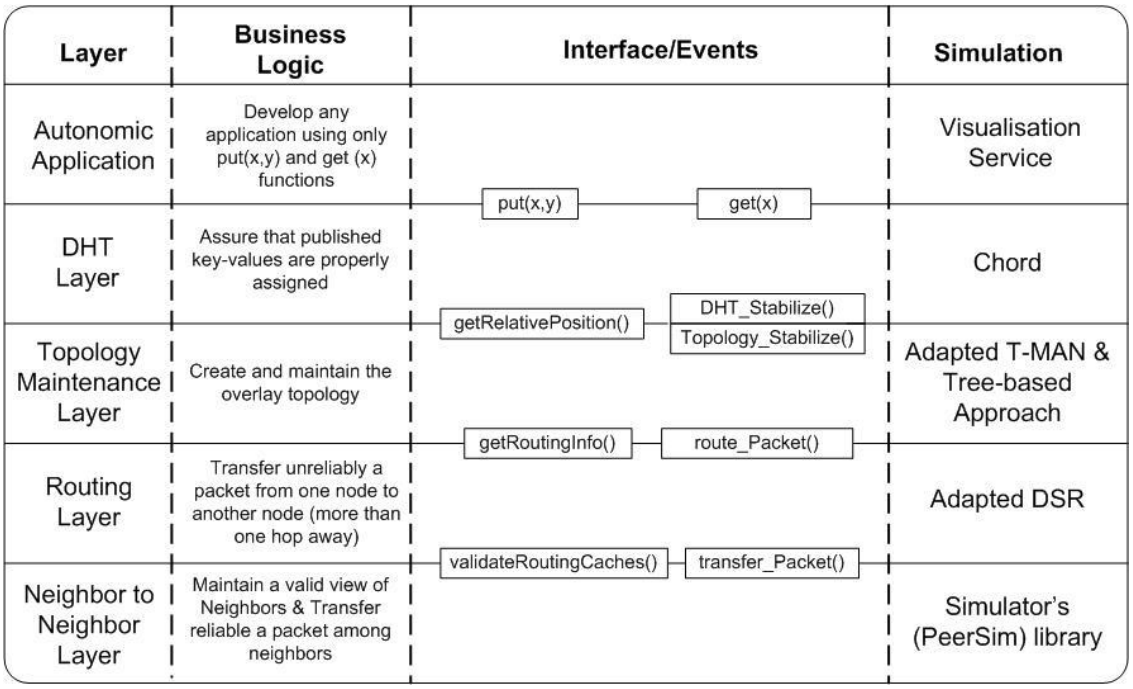

Figure 3.4 Four-Layered Approach

The Neighbor-to-Neighbor layer is responsible for delivering an upper-layer frame from a neighbor to another neighbor. No information from the upper layer is necessary for the delivery. Two types of messages are used; i) *MAC_SEND* in order to achieve one way frame delivery from neighbor X to neighbor Y and ii) *MAC_ACK* in order to achieve acknowledgment for successful message-delivery from neighbor Y back to neighbor X. Also, this layer is responsible for maintaining (i.e. initializing and keeping up-to-date) the routing cache of the Routing layer since, when neighbor-to-neighbor links are created or destroyed, the related routing information has to be updated.

The Routing layer is responsible for delivering an upper-layer frame from a node X to another node Z. It is assumed that node X is not aware how node Z can be reached. The layer is also agnostic of the reason that node X wants to communicate with node Z. This layer relies on routing protocol for frame forwarding across the network. As we stated in section 3.3, in case of Mesh Environments it is suggested the use of a dynamic routing protocol (will be covered in chapter 4).

The Topology Maintenance layer is responsible for formulating a virtual topology of the participating nodes. In our case, the desired topology is a ring (imposed by the use of Chord). Consequently, this layer undertakes the task of identifying the relative position of each node in the overlay topology without being based in centralized or semi-centralized techniques.

The DHT layer is responsible for maintaining a distributed hash table that is bootstrapped over the stabilized overlay topology. For this purpose any existing DHT protocol may be used. These protocols are (semi or fully) decentralized and -in addition to storage and retrieval functionality- may succeed load balancing, reduce bandwidth consumption and improve data reliability across the network. The following interfaces have been defined for the communication among the different layers:

- The Neighbor-to-Neighbor layer provides to the Routing layer routing information for existing neighbors that is stored in the routing cache of each node, through the *validateRoutingCaches()*

function. The Neighbor-to-Neighbor layer provides also medium-level acknowledgments to the Routing layer for neighbor-to-neighbor communication, through the *transfer_Packet()* function.

- The Routing layer provides routing functionality to upper layers through the *routePacket()* function. Additionally, it exposes topology information derived directly from the routing caches to the Topology Maintenance layer, through the *getRoutingInfo()* function. It is up to the Topology Maintenance layer to utilize this information for optimizing its mechanisms or not.
- The Topology Maintenance layer provides information to the DHT layer regarding the relative position of a node in the overlay network (e.g. the predecessor and successor in case of a ring topology) through the *getRelativePosition()* function. In case of changes in the network topology, stabilization procedures take place in both layers. The *Topology_Stabilize()* function is used for re-ordering the overlay topology (e.g. ring in our case) and triggers the *DHT_Stabilize()* function that is used for the re-assignment of key-value pairs that are assigned in the overlay network nodes.

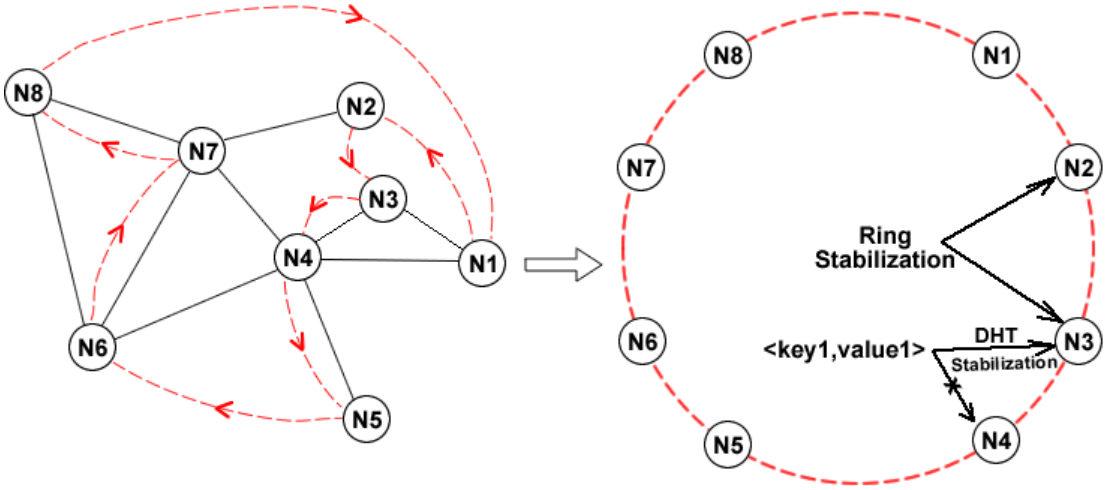

Figure 3.5 Overlay Topology stabilization & DHT entries stabilization

In Figure-4.3, a snapshot of the physical network topology (solid lines) and the logical overlay topology (dashed lines with arrows) is depicted. Initially, node 3 does not exist in the network and the key-value pairs have already been assigned to the network nodes by applications that run on the existing nodes (i.e. applications that use DHT). Then, node 3 is physically connected with node 1 and node 4 and the corresponding overlay topology is updated. It is the responsibility of Topology Maintenance layer to find the successor for each node. However, it is not the Topology Maintenance layer's responsibility to re-assign key-values according to the DHT's assignment algorithm.

The Topology Maintenance layer must inform the DHT layer that the relative position for the node in the overlay topology (e.g. ring in case of Chord) has changed. Then it is up to DHT layer to reassign key-value pairs. This re-assignment will be addressed as DHT re-stabilization while the updated knowledge for the relative position in the overlay topology is called Topology stabilization.

4. PrEstoCloud Device Stack

4.1 Layers of the PDK

Based on Figure 3.4 the following stack has been proposed to handle efficiently edge resources.

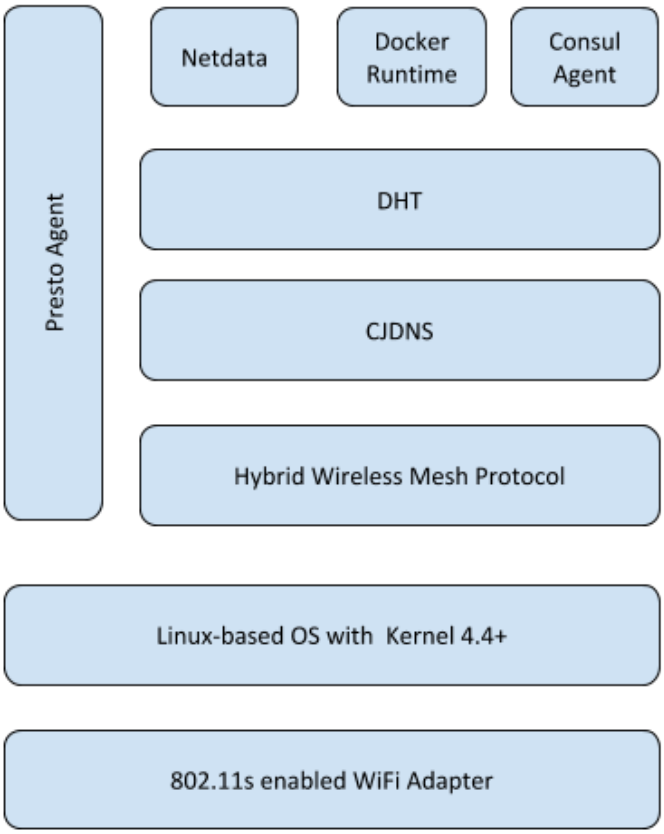

Figure 4.1 Granular View of the PrEstoCloud stack

An overview of each layer will be provided following a bottom up approach. In the lowest part of the stack we have a hardware dependency. More specifically, an 802.11s-enabled card should exist. The reason for that is that **802.11s** is one of the latest standards that are accepted by the 802.11s standardization group that offers native mesh networking capabilities. At this point it should be clarified that, in the frame of PrEstoCloud, mesh-networking should not be confused with ad-hoc networking. In the ad-hoc networking paradigm only single hop-connectivity is supported; hence all participating nodes should be reachable. In the mesh paradigm, multi-hop links are supported i.e. one node can be linked with two nodes that do not have reachability among them. This difference is also depicted on Figure 4.2. In the frame of PrEstoCloud ad-hoc capabilities are not enough since they support only single-hop communications.

The mesh networking standard is not supported by the majority of the commercial wi-fi adapters. The reader is prompted to visit the Linux Kernel Wireless Drivers' page[8] where the capabilities of each driver is listed. As it can be seen (Figure 4.3) a limited set of drivers are developed that include the specific networking capability. In the frame of our testbed we used rt2800usb[9] driver because of its compatibility in IoT devices.

[8] https://wireless.wiki.kernel.org/en/users/drivers

[9] https://wireless.wiki.kernel.org/en/users/drivers/rt2500usb

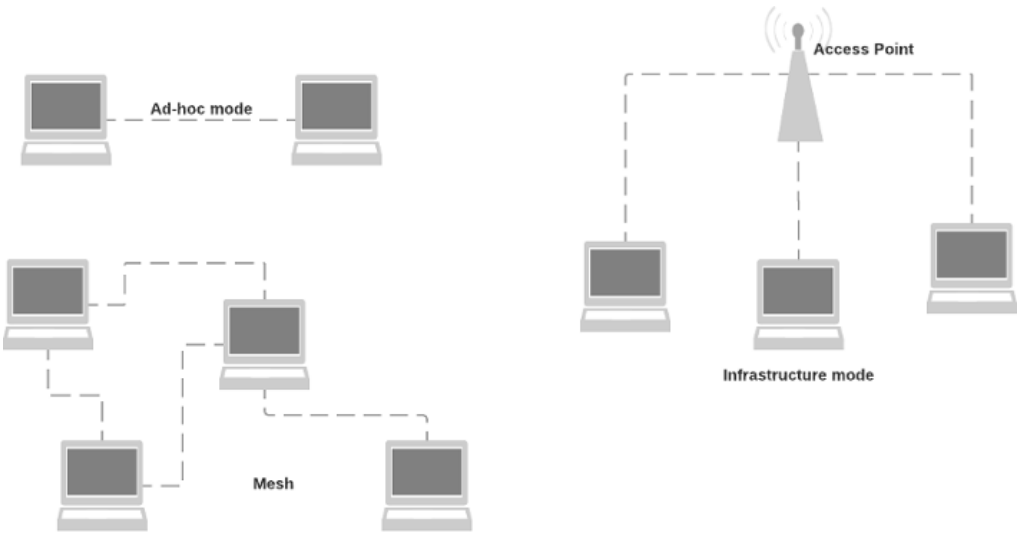

Figure 4.2 Mesh mode vs Ad-hoc mode

Driver	Manufacturer	cfg80211	AP	IBSS	mesh	monitor	PHY modes
adm8211	ADMtek/Infineon	yes	no	no	no	?	B
airo	Aironet/Cisco	no	?	?	?	?	B
ar5523	Atheros	yes	no	no	no	yes	A(2)/B/G
at76c50x-usb	Atmel	yes	no	no	no	no	B
ath5k	Atheros	yes	yes	yes	yes	yes	A/B/G
ath6kl	Atheros	yes	no	yes	no	no	A/B/G/N
ath9k	Atheros	yes	yes	yes	yes	yes	A/B/G/N
ath9k_htc	Atheros	yes	yes	yes	yes	yes	B/G/N
ath10k	Atheros	yes	yes	yes	no	yes	A/B/G/N/AC
atmel	Atmel	no	?	?	?	?	B

Figure 4.3 Mesh support by existing drivers

In addition to the usage of a mesh-enabled hardware device, a proper OS kernel has to be used that is able to interact with the mesh-capable device. The kernel that is widely used is called open80211s[10] and most of the modern kernels are built with this module already integrated. Since the IoT devices that will be used in

[10] https://github.com/o11s/open80211s/wiki/HOWTO

our pilots are Raspberry[11]-based we decided to use Linux kernel 4.4+ which is used in all compatible operating systems i.e. Raspbian[12], Ubuntu Core[13], Ubuntu Mate[14].

The ability to have mesh-level communication is a **prerequisite** for edge device to edge device communication. However, **layer-3 communication per se is not guaranteed by 802.11s protocol since the protocol is a pure layer-2 link** management protocol. In order to achieve IP-based communication a **routing layer** must be established. A routing layer cannot relay on a traditional centralized gateway that maintains routing tables since the topology in a mesh environment is rapidly changing. On top of that, the IP assignment cannot be static i.e. in case a new node arrives it should not consult the existing ones which IPs are reserved in order to perform an initial assignment. This would not scale. These problems are addressed as dynamic routing and IP assignment problem.

Both of these problems have been resolved through the incorporation of a **reactive routing protocol.** Such protocol will be offered through the combination of HWMP (see section 4.2.5) with CJDNS (see section 4.3) . Both of these will be analyzed below. The idea is that each node is auto-generating an IPv6 address along a cryptographic key-pair. The public-key along the IP addressed are exchanged using layer-2 based communication. Upon all exchanges, the mesh participants maintain a local routing table. In case a node wants to communication with another node that is not layer-2 reachable it initiates a find-route request which is propagated through its layer-2 peers. During the message propagation a route is identified and even stored in the intermediate routes. Although this approach has the penalty of layer-2 "flooding" it requires zero configuration and zero-maintenance during operation. Also it is immune to topological changes and topology splits/joins.

On top of this layer, a set of layer-7 services are installed. This include **a)** a **Docker runtime engine**[15] that is used to manage the dynamic deployment of JPPF Tasks, **b)** the **Netdata**[16] monitoring probe that is used to extract compute and network measurements from the JPPF Task execution, **c)** the **Consul**[17] service discovery agent that is used to announce the nodes existence and also maintain the consistent key-value store and **d)** the **PrEstoCloud Agent** which is the daemon that has a twofold role since on the one hand it proxies the programmability of all installed components (i.e. join mesh, deploy JPPF task, set key/value, get key) and on the other hand it can be used by any JPPF Task in order to interact with the DHT.

4.2 Mesh Networking

A mesh network is defined as two or more nodes that are interconnected via IEEE 802.11 links which communicate via mesh services and constitute an IEEE 802.11-based wireless distribution system (WDS). A mesh link is shared by two nodes who can directly communicate with one another via the wireless medium. A pair of nodes that share a link are neighbours. Any node that supports the mesh services of control, management, and operation of the mesh is a mesh point (MP). If the node additionally supports access to client stations (STAs) or non-mesh nodes, it is called a mesh access point (MAP). A mesh portal (MPP) is an MP that has a non-802.11 connection to the Internet and serves as an entry point for MAC service data units

[11] https://www.raspberrypi.org/

[12] https://www.raspberrypi.org

[13] https://www.ubuntu.com/core

[14] https://ubuntu-mate.org/raspberry-pi/

[15] https://docs.docker.com/install/linux/docker-ce/debian/#prerequisites

[16] https://my-netdata.io

[17] https://www.consul.io/

(MSDUs) to enter or exit the mesh (Figure 4.4). An MPP and MAP may be collocated on one device. The draft standard[18] additionally defines options for power-constrained MPs to be lightweight, in which nodes are able to communicate only with their neighbours and do not use the distribution system (DS) or provide congestion control services. It additionally defines a non-forwarding MP for leaf nodes that can fully operate within the mesh even if no MAPs are available (which a STA could not do). A mesh network can have one operating channel or multiple operating channels. A unified channel graph (UCG) is a set of nodes that are interconnected on the same channel within a mesh network.

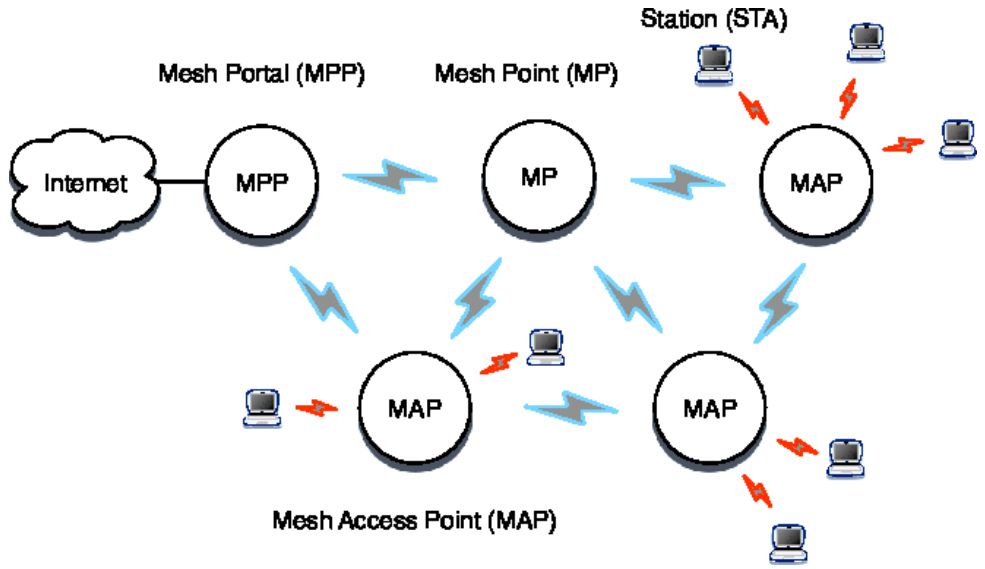

Figure 4.4 IEEE 802.11s terms: A mesh portal connects to the wired Internet, a mesh point just forwards mesh traffic, and a mesh access point additionally allows stations to associate with it.

4.2.1 Channel Selection

After initialization, a node uses the Simple Channel Unification Protocol where the MP performs active or passive scanning of the neighbours. If no neighbouring MPs are found, the MP can establish itself as the initiator of a mesh network by selecting a channel precedence value based on the boot time of the MP plus a random number. If two disjoint mesh networks are discovered (i.e., they are on different channels), the channel is chosen according to the highest precedence value. If the mesh is in the 5 GHz band, the mesh is required to conform to the regulatory requirements of the dynamic frequency selection (DFS) and radar avoidance to conform with FCC UNII-R regulation.

4.2.2 Topology Discovery and Link State

Mesh points that are not yet members of the mesh must first perform neighbour discovery to connect to the network. A node scans neighbouring nodes for beacons that contain at least one matching profile, where a profile consists of a mesh ID, path selection protocol identifier, and link metric identifier. If the beacon contains a mesh capacity element that contains a nonzero peer link value, the link can be established through a secure protocol (see Figure 4.5).

[18] https://standards.ieee.org/findstds/standard/802.11s-2011.html

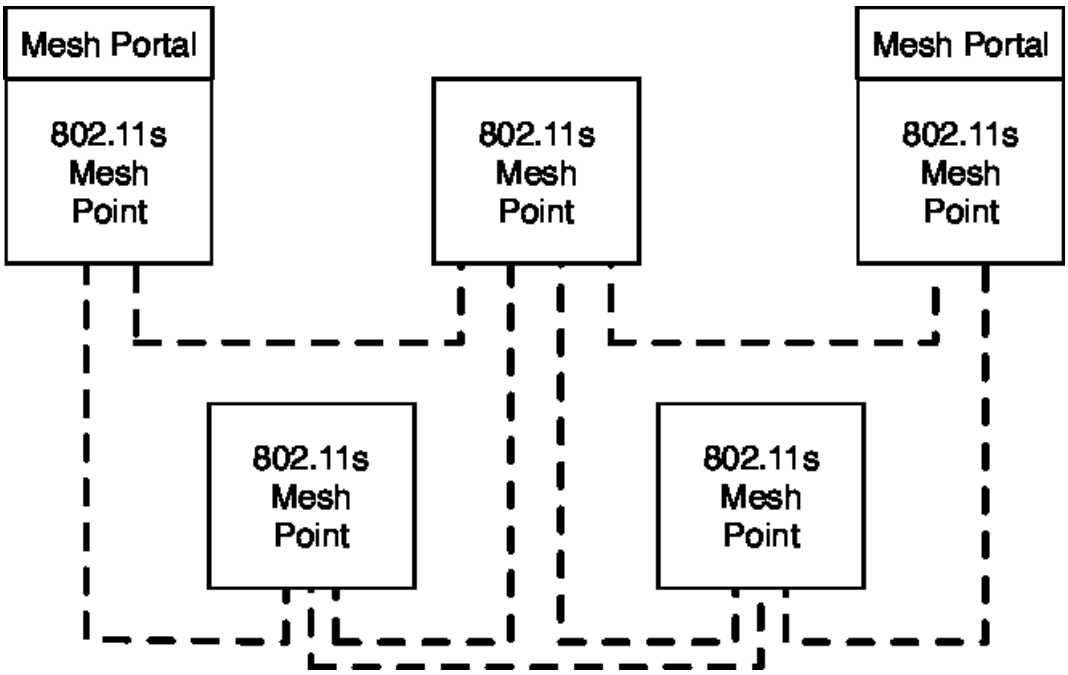

Figure 4.5 Reference model for WLAN mesh interworking.

Mesh portals bridge the wireless and wired networks. MPPs function as if on a single loop-free logical layer 2 and interconnected layer 3 for both the internal mesh and the external LAN segments. For layer 2, the MPPs use the IEEE 802.1D bridging standard, and at layer 3, routing must be performed in a similar fashion to IP gateway routers.

4.2.3 Path Selection and Routing

Within a mesh, all mesh stations use the same path metric and path selection protocol. For both, 802.11s defines a mandatory default scheme. Because of its extensible framework, they can be replaced by other solutions. The default metric, termed airtime metric, indicates a link's overall cost by taking into account data rate, overhead, and frame error rate of a test frame of size 1 kbyte. The default path selection protocol, Hybrid Wireless Mesh Protocol (HWMP)(see section 4.2.5), combines the concurrent operation of a proactive tree-oriented approach with an on-demand distributed path selection protocol (derived from the Ad Hoc On Demand Distance Vector [AODV] protocol [25]). The proactive mode requires a mesh station to be configured as a root mesh station. In many scenarios this will be a mesh station that collocates with a portal. As such, the root mesh station constantly propagates routing messages that either establish and maintain paths to all mesh stations or simply enable mesh stations to initiate a path to it. Mesh stations also rely on AODV when a root mesh station is unavailable. When no path setup messages are propagated proactively, however, the initial path setup is delayed.

4.2.4 Medium Access Control

For medium access, mesh stations implement the mesh coordination function (MCF). MCF consists of a mandatory and an optional scheme. For the mandatory part, MCF relies on the contention-based protocol known as Enhanced Distributed Channel Access (EDCA), which itself is

an improved variant of the basic 802.11 distributed coordination function (DCF). Using DCF, a station transmits a single frame of arbitrary length. With EDCA, a station may transmit multiple frames whose total transmission duration may not exceed the so-called transmission opportunity (TXOP) limit. The intended receiver acknowledges any successful frame reception. Additionally, EDCA differentiates four traffic categories with different priorities in medium access and thereby allows for limited support of quality of service (QoS).

To enhance QoS, MCF describes an optional medium access protocol called Mesh Coordinated Channel Access (MCCA). It is a distributed reservation protocol that allows mesh stations to avoid frame collisions.

With MCCA, mesh stations reserve TXOPs in the future called MCCA opportunities (MCCAOPs). An MCCAOP has a precise start time and duration measured in slots of 32 µs. To negotiate an MCCAOP, a mesh station sends an MCCA setup request message to the intended receiver. Once established, the mesh stations advertise the MCCAOP via the beacon frames. Since mesh stations outside the beacon reception range could conflict with the existing MCCAOPs, mesh stations also include their neighbours' MCCAOP reservations in the beacon frame. At the beginning of an MCCA reservation, mesh stations other than the MCCAOP owner refrain from channel access. The owner of the MCCAOP uses standard EDCA to access the medium and does not have priority over stations that do not support MCCA. Although this compromises efficiency, simulations reveal that high medium utilization can still be achieved with MCCA in the presence of non-MCCA devices [26] Z. After an MCCA transmission ends, mesh stations use EDCA for medium contention again.

4.2.5 Hybrid Wireless Mesh Protocol

The IEEE 802.11s standard suggests HWMP to provide both on-demand routing for predominantly mobile topologies and proactive tree- based routing for predominantly fixed infrastructure networks (the protocol is not bound to HWMP since a functional equivalent protocol can be used). The hybrid protocol is used when an MP does not have an on-demand route to another MP and sends the first packet to the root. Subsequent packets can be sent along a shorter path that is found directly.

4.2.5.1 On-Demand Routing

With an on-demand routing protocol, the network is not required to use routes through the root node (or even have a root node). Specifically, IEEE 802.11s MPs can use a route request (RREQ) and route reply (RREP) mechanism to discover link metric information from source to destination. To maintain the route, nodes send periodic RREQs where the time between two different RREQs transmitted at the same source is known as a refresh-round. Sequence numbers are used per refresh-round to ensure loop-free operation. To avoid updating poor routes too quickly, hysteresis is used to maintain operation of the better route if the updated RREQ from the original route is lost or the RREQ from along another route is delivered first in a particular round. Each best candidate route is cached for later use if loss occurs on a newly selected route.

4.2.5.2 Tree-Based Routing

When an MPP exists within the topology, the network can use proactive distance vector routing through the root to find and maintain routes. The root announcement is broadcast by the root MPP with a sequence number assigned to each broadcast round. Each node updates the metric as the announcements are received and rebroadcast. The MP chooses the best parent and caches other potential parents. Periodic RREQs are sent to parents to maintain the path to the root. If the connection to the parent is lost (three consecutive RREQs), the MP will notify its children, find a new parent, and send a gratuitous RREP to the root, which all intermediate nodes use to update their next-hop information about the source.

4.3 CJDNS as Zero-Configuration layer-3

Cjdns[19] is a networking protocol, a system of digital rules for message exchange between computers. The philosophy behind cjdns is that networks should be easy to set up, protocols should scale up smoothly and security should be ubiquitous. Cjdns implements an encrypted IPv6 network using public key cryptography for network address allocation and a distributed hash table for routing. The New Scientist reports that "Instead of letting other computers connect to you through a shared IP address which anyone can use, cjdns only lets computers talk to one another after they have verified each other cryptographically. That means there is no way anyone can be intercepting your traffic.

[19] https://github.com/cjdelisle/cjdns

The cjdns program talks to other programs on the computer through a TUN device which the computer sees as a regular network interface that accepts IP datagrams. Any program that uses IPv6 can communicate in a cjdns-based network without any modification. Cjdns can communicate over wireless and Ethernet connections as well as tunnel over the internet.

Cjdns addresses are the first 16 bytes (128 bits) of the double SHA-512 of the public key. All addresses must begin with the byte 0xFC, which in IPv6 resolution, is a private address (so there is no collision with any external Internet addresses).

The address is generated initially when a node is set up, through a brute-forced key generation process (keys are repeatedly generated until the result of double SHA-512 begins with 0xFC). This process is unique, as it guarantees cryptographically bound addresses (the double SHA-512 of the public key), sourced from random data (private key is random data, public key is the scalar multiplication of this data).

The routing engine stores its routing table in a distributed hash table similar to Kademlia. When forwarding a packet, rather than looking up an entry using the traditional Kademlia approach of asking a node whose id is similar to that of the target, cjdns forwards the packet to that node for further processing. In order to allow a node to be in touch with many nodes despite being directly connected only to as few as one, there is a switch layer which underlies the routing layer. The switch is inspired by MPLS protocol but without the universal uniqueness nor longevity of MPLS labels but instead with added ability to determine the source of an incoming packet from its label and ability to determine whether a given node is part of the path represented by a label, and ability to switch a label without any memory lookups. In the simplest terms: a switch label is like driving directions to a destination.

It is designed so that every node is equal; there is no hierarchy or edge routing. Rather than assigning addresses based on topology, all cjdns IPv6 addresses are within the FC00::/8 Unique local address space (keys which do not hash to addresses starting with 'FC' are discarded). Although nodes are identified with IPv6 addresses, cjdns does not depend upon having IPv6. Currently, each node may be connected to a few other nodes by manually configuring links over an IPv4 or IPv6 network (the Internet). The ultimate goal is to have every node connected directly by physical means; be it wire, optical cable or radio waves.

A CryptoAuth session between two given nodes is set up with a two-packet handshake. Each of the two packets contains the permanent and temporary keys of the sending node which are piggybacked on top of normal data packets. The data in those packets is encrypted using the permanent keys. Once the temporary keys have been exchanged, the permanent keys are no longer used in that session and the temporary keys are discarded when the session ends so that the data sent during that session cannot be decrypted later. Finally, it should be clarified that since the handshake is piggybacked on top of the first two packets, the maximum allowable packet size differs from packet to packet.

4.3.1 Routing considerations

Routing is designed such that each packet requires very little handling by an individual router, or node. Each node will respond to 'search queries' asking it for other nodes nearby to it. This allows the sending node to determine and add routes to its own routing table. Once the sending node has determined a route, it sends its packet to the first node on said route. For each hop, the receiving node reads the packet's header to determine where to next send the packet. Before the packet is forwarded to the next hop, the node performs a bit shift on the packet's headers, making it ready for use by the next node.

The Source routing used by cjdns has advantages for performance and extensibility. Nodes can use experimental routing algorithms with existing meshes, and new releases of cjdns can change the default routing algorithm without creating protocol incompatibilities. The major security problem of source routing, IP address spoofing, is prevented by the end-to-end nature of cjdns encryption.

4.3.2 Security considerations

The belief that security should be ubiquitous and unintrusive like air is part of the core philosophy behind cjdns. The routing engine runs in user space and is compiled by default with stack-smashing protection, position-independent code, non-executable stack, and remapping of the global offset table as read-only

(relro). The code also relies on an ad-hoc sandboxing feature based on setting the resource limit for open files to zero, on many systems this serves block access to any new file descriptors, severely limiting the code's ability to interact with the system around it.

4.4 Layer-7 components

After layer2 and layer-3 (auto)configuration, a set of layer-7 services will be executed in each node. These services are not **only related to the operation** of the DHT but also with the global configuration of the edge resource. More specifically, the services that will be pre-installed on each device will be:

- a) The **Netdata[20] monitoring probe** which will be responsible to
- b) The **Container Runtime Engine** which will be responsible for deploying and undeploying containers
- c) The **Consul[21] DHT** as a base key/value store
- d) The **PrEstoCloud Agent** which coordinates the execution of all the above plus it provides a REST-based management interface for external components

We will briefly elaborate each one of the layer-7 components.

4.4.1 Netdata monitoring probe

Netdata is a system for distributed real-time performance and health monitoring. It provides unparalleled insights, in real-time, of everything happening on the system it runs, using modern interactive web dashboards (see Figure 4.6). The monitoring framework is fast and efficient, designed to permanently run on all systems (physical & virtual servers, containers, IoT devices), without disrupting their core function. Therefore, it is already ported on arm-based architectures. Based on its benchmarking, it responds to all queries in less than 0.5 ms per metric, even on low-end hardware while it supports dynamic thresholds, hysteresis, alarm templates, multiple role-based notification methods. Furthermore, it is extensible since you can monitor anything you can get a metric for, using its Plugin API. Moreover, the library is auto-configurable since it can collect up to 5000 metrics per server out of the box. Finally, several time-series back-ends are supported out of the box, including Prometheus[22] which is PrEstoCloud's choice.

[20] https://github.com/firehol/netdata

[21] https://consul.io

[22] https://prometheus.io

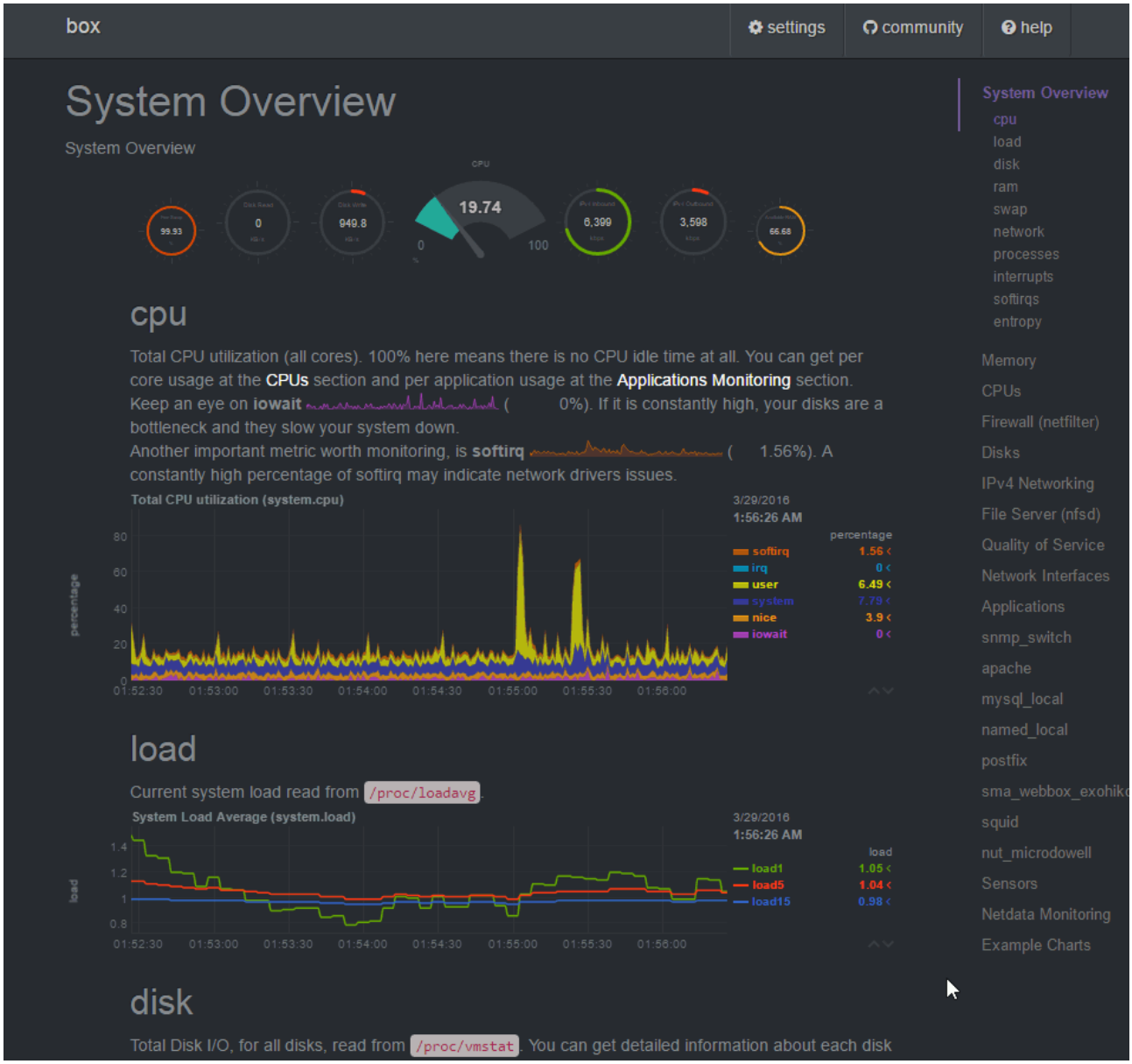

Figure 4.6 Netdata monitoring probe configuration

The endpoint where monitoring streams are reported is configured by the PrEstoCloud agent.

4.4.2 Container runtime engine

Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment which consists from edge resources in our case. Containerization provides a clean separation of concerns, as developers focus on their application logic and dependencies, while IT operations teams can focus on deployment and management without bothering with application details such as specific software versions and configurations specific to the app. For those coming from virtualized environments, containers are often compared with virtual machines (VMs). You might already be familiar with VMs: a guest operating system such as Linux or Windows runs on top of a host operating system with virtualized access to the underlying hardware. Like virtual machines, containers allow you to package your application together with libraries and other dependencies, providing isolated environments for running your software services. As you'll see below however (Figure 4.7), the similarities end here as containers offer a far more lightweight unit for developers and IT Ops teams to work with, carrying a myriad of benefits.

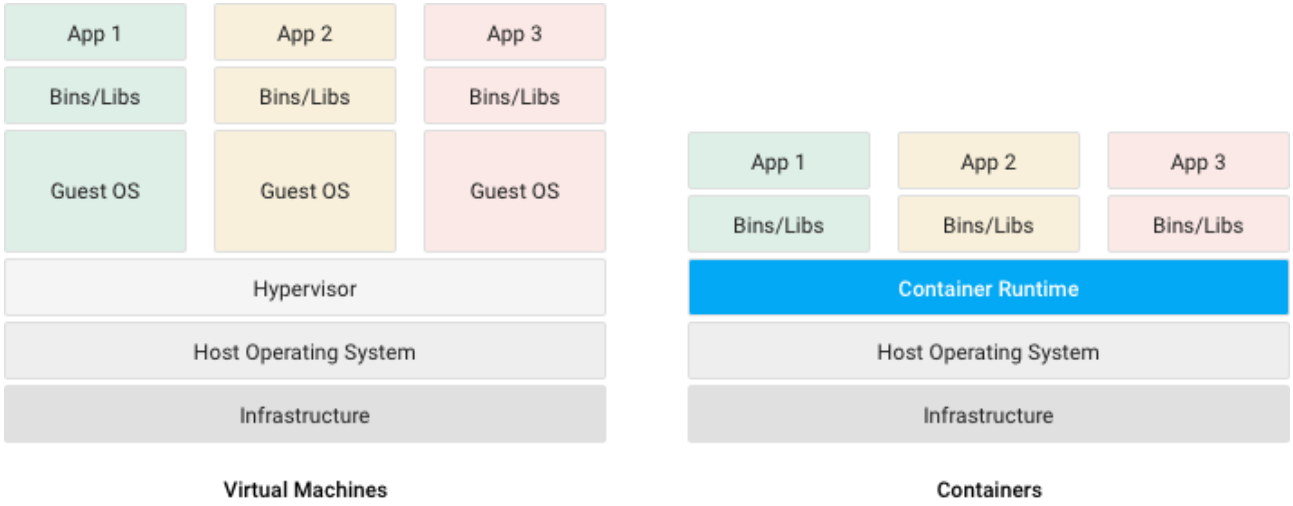

Figure 4.7 Containers vs VMs

Instead of virtualizing the hardware stack as with the virtual machines approach, containers virtualize at the operating system level, with multiple containers running atop the OS kernel directly. This means that containers are far more lightweight: they share the OS kernel, start much faster, and use a fraction of the memory compared to booting an entire OS. To this end, in the frame of PrEstoCloud Docker Engine[23] will be used as the default Container engine. **It should be clarified that all PrEstoTasks that will be executed in the edge devices will be wrapped as containers**.

4.4.3 Consul DHT

Consul is a tool offering Distributed Hash Table capabilities developed mainly for service discovery and configuration. The is distributed, highly available, and extremely scalable and it provides the following key features:

- Key/Value Storage - A flexible key/value store enables storing dynamic configuration, feature flagging, coordination, leader election and more.
- Service Discovery - Consul makes it simple for services to register themselves and to discover other services via a DNS or HTTP interface. External services such as SaaS providers can be registered as well.
- Health Checking - Health Checking enables Consul to quickly alert operators about any issues in a cluster. The integration with service discovery prevents routing traffic to unhealthy hosts and enables service level circuit breakers.

Finally, Consul is built to be datacenter aware, and can support any number of regions without complex configuration. Furthermore, it runs on Linux and supports IoT deployments. Its behavior is also controlled by the PrEstoCloud Agent.

4.4.4 PrEstoCloud Agent

The PrEstoCloud Agent is responsible to manage the lifecycle of all the components above. Furthermore, it proxies some of their functionalities acting as a point of unification. The table below provides an abstract view of the method

Table 4-1 Summary of API calls of the PrEstoCloud Agent

Method	Description

[23] https://www.docker.com

/getNodeId	It returns the descriptive code of the edge resources as configured by the CJDNS. Practically, it is a global-scope IPv6 address that will not change even upon the reboot of the edge device.
/getDeviceContext	It returns the immutable meta-data of the edge device such as architecture (e.g. arm), storage, memory, cpu-type, cpu-speed etc.
/getAdjacentNodes	It returns the identifiers of the neighbourhood nodes in the format that is reported by the 802.11s driver.
/getPublicKey	It returns the cryptographic public key that is assigned to the edge resource during the CJDNS bootstrappling
/setClusterHead	It enforces the edge resource to consider a specific node as a "Cluster Head". A Cluster Head is a node that is responsible for some "centralized tasks" such as measurements' collection, workload-prediction etc.
/getClusterHead	It returns the current node identifier that is considered clusterhead by the resource.
/becomeClusterHead	It instructs the edge resource to become a cluster head. This practically means that the resource will start beaconing this fact to the entire mesh.
/deployContainer	It is used to store a container in the local container registry of the edge resource.
/getDeployedContainers	It is used to retrieve the containers that are already registered in the device's registry
/deleteDeployedContainer	It instructs the edge resource to delete one container from its local registry
/startDeployedContainer	It instructs the edge resource to initiate a container that already exists in its registry
/getRunningContainers	It returns the list of the running containers in the edge resource
/stopRunningContainer	It instructs the edge resource to stop a running container.
/getMeasurementsForMetric	It returns a list of timestamped values that represent for a set if metrics
/putKeyValuePair	It triggers a transactional "put" in the DHT
/getValuesForKey	It returns a list of values from the DHT

4.5 Testbed

For the sake of experimentation several edge resources have been employed. Figure 4.5 illustrates some of the devices that are being used.

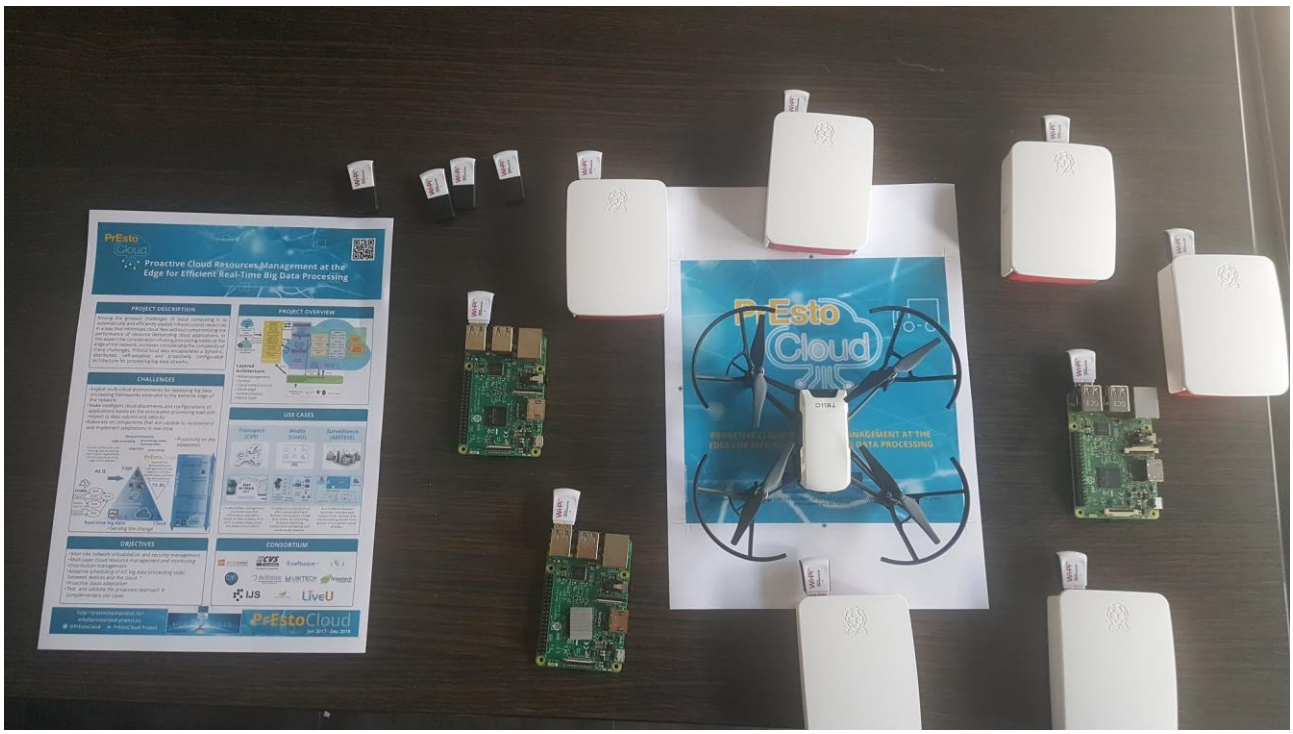

Figure 4.8 Edge resources used during PrEstoCloud experiments

More specifically, the following devices have been used:

- 10 Raspberyy Pi devices (Model 3B & Model 3B+)
- 1 DJI Tello Drone[24]
- 5 Intel NUC[25]s
- several laptops
- 20 WiPi Mesh cards[26]

In the first phase of development, special emphasis has been given in the established on of a fully dynamic environment where temporal storage is provided. In the second phase of the project, detailed measurements regarding the efficiency of the logical topology maintenance and the read/writes to the DHT will be conducted taking under consideration various topology and mobility scenarios.

[24] https://store.dji.com/product/tello

[25] https://www.intel.com/content/www/us/en/products/boards-kits/nuc.html

[26] https://export.farnell.com/element14/wipi/dongle-wifi-usb-for-raspberry/dp/2133900?COM=referral-noscript

5. Conclusions

The scope of this deliverable was to elaborate on the Spatiotemporal Processing capabilities that are offered by the PrEstoCloud framework. These capabilities relate to the **need that PrEstoCloud Tasks have for storing and retrieving datasets** from other tasks **during execution of jobs on the edge devices**. More specifically, when performing distributed computations, there is a need for reading and writing data to a data storage structure which is accessible by all compute nodes. This data structure must expose an API (i.e. read/write functional primitives) which will be used by the business logic of the computations per se. In case the computations are performed in a data center there are many data structures that can be used. Existing big data frameworks depend on underlying storage engines that operate on data centers using reliable resources that are reachable with low-latency. Reliability refers to both compute and networking aspects.

The goal of PrEstoCloud is to provide a holistic "Proactive Cloud Resources Management framework targeting the Edge". Unfortunately, the **operational assumptions** of compute/network reliability are **not valid** in the case of edge resources. In the edge, the operational environment consists of resource limited devices that formulate temporal connections using mesh connectivity principles. Such connections can be **established** or **broken** at any time based on the mobility profile of the edge devices. The strategy of "offloading" data to the backhaul part of the network is not viable because of the huge delays that would be raised and to the unnecessary utilization of the network capacity.

PrEstoCloud will offer a sophisticated solution of temporal storage based on the principles of a Distributed Hash Table. A DHT is a data structure that is created and maintained by many network participants without any centralization. This structure is totally distributed structure since it does not have the necessity of a leader. However, this structure requires a pre-existing routing layer in order to formulate an overlay which is used for reading/writing data. In a Mesh Environment there is not pre-existing routing layer. Moreover, the additions and removals of nodes from the mesh are totally random; hence there is no central routing table which can be used to define layer-3 paths from one node to another node.

The provided solution resolves all issues above through the incorporation of a **PrEstoCloud Device Stack** (PDS). PDS is a software package that upon installation, in an edge device, performs all appropriate configurations so as an edge resource to be able to accept computional tasks and in parallel to be able to interact with the DHT that is member of. More specifically, PDS is responsible to enroll an edge device to a layer-2 Mesh and in parallel to assign to it a routable IPv6 address. The first is achieved using the 802.11s protocol while the latter is achieved using the CJDNS IPv6 protocol.

Upon joining a mesh with a routable IP, a DHT protocol will be bootstrapped using the Chord algorithm. Through the deployed stack, any booted node will broadcast its existence to its adjacent nodes. Through this broadcasting all nodes that execute Chord protocol will restabilize. During re-stabilization, all reads/writes that are performed are executed without disruption. Any JPPF Task that will be deployed to the edge device that participates in the DHT will be able to read and write data using a socket to the localhost. In this way, PrEstoCloud offers maximum functional transparency since a developer is not aware about the location of the edge resource and its configuration. Finally, it should be noted that the **CAP theorem states that no distributed system can have Consistency, Availability, and Partition-tolerance**. Our implementation falls under the category of **AP** systems.

References

[1] PrEstoCloud, (2017b). D2.3 – Requirements for the PrEstoCloud platform. Public deliverable, available at: http://prestocloud-project.eu/new/deliverables/.

[2] R. Dingledine, M. J. Freedman, and D. Molnar, "Accountability measures for peer-to-peer systems," in Peer-to-Peer: Harnessing the Power of Disruptive Technologies, D. Derickson, Ed. O'Reilly and Associates, November 2001

[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in Processings of the ACM SIGCOMM, 2001, pp. 161–172

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan,"Chord: A scalable peer-to-peer lookup protocol for internet applications," IEEE/ACM Transactions on Networking, vol. 11, no. 1, pp. 17–32, 2003

[5] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," Lecture Notes in Computer Science, vol. 2218, 2001

[6] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D.Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," IEEE Journal on Selected Areas in Communications, vol. 22, no. 1, pp. 41–53, January 2004

[7] P. Maymounkov and D. Mazi` res, "Kademlia: A peer-to-peer information system based on the xor metric," in Processings of the IPTPS, Cambridge, MA, USA, February 2002, pp. 53–65

[8] D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: a scalable and dynamic emulation of the butterfly," in Processings of the ACM PODC'02, Monterey, CA, USA, July 2002, pp. 183–192

[9] Guido Urdaneta, Guillaume Pierre and Maarten van Steen. A Survey of DHT Security Techniques. ACM Computing Surveys 43(2), January 2011

[10] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, May 1997, pp. 654–663

[11] P. Maymounkov and D. Mazi` res, "Kademlia: A peer-to-peer information system based on the xor metric," in Processings of the IPTPS, Cambridge, MA, USA, February 2002, pp. 53–65

[12] Y. Wang, X. Yun, and Y. Li, "Analyzing the Characteristics of Gnutella Overlays", Fourth International Conference on Information Technology, pp.1095-1100, April 2007

[13] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", In Proceedings of SIGCOMM 2001, San Deigo, CA, August 2001

[14] C. Plaxton, R. Rajaraman, and A. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures,1997

[15] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distribution: Evidence and implications," in Proceedings of the IEEE INFOCOM, 1999

[16] **F**. Dabek, B. Zhao, P. Druschel, and I. Stoica, "Towards a common api for structured peer-to-peer overlays," in Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS 2003), Berkeley,California, USA, February 20-21 2003

[17] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker, "Spurring adoption of dhts with openhash, a public dht service," in Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS 2004), Berkeley, California, USA, February 26-27 2004

[18] A. Malatras, State-of-the-art survey on P2P overlay networks in pervasive computing environments, Journal of Network and Computer Applications,Elsevier 2015

[19] Gouvas, P., Zafeiropoulos, A., Liakopoulos, A., Mentzas, G., Mitrou, N., (2010), Integrating Overlay Protocols for Providing Autonomic Services in Mobile Ad-hoc Networks, IEICE Transactions on Communications, Vol. E9x-B, No.8, August 2012

[20] "Secure hash standard," NIST, U.S. Dept. of Commerce, National Technical Information Service FIPS 180-1, April 1995

[21] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with cfs," in Proceedings of the eighteenth ACM symposium on Operating systems principles, 2001, pp. 202–215

[22] L. P. Cox, C. D. Murray, and B. D. Noble, "Pastiche: making backup cheap and easy," SIGOPS Oper. Syst. Rev., vol. 36, no. SI, pp. 285–298, 2002

[23] B. Jennings, S. Van der Meer, S. Balasubramaniam, D. Botvich, M.O. Foghlu, W. Donnelly and J. Strassner, "Towards Autonomic Management of Communications Networks", IEEE Communications Magazine, vol.45, no.10, pp.112-121, October 2007

[24] DHT-Based Mobile Service Discovery Protocol for Mobile Ad Hoc Networks, Proceedings of the 4th international conference on Intelligent Computing, pp.610 – 619, 2008

[25] C. Perkins, E. Belding-Royer, and S. Das, "Ad Hoc On-Demand Distance Vector (AODV) Routing," IETF RFC 3561, Jul. 2003

[26] Y. Chen, and S. Emeott, "MDA Simulation Study: Robustness to Non-MDA Interferers," IEEE 802 Plenary Meeting, Submission 11-07-0356-00-000s, Orlando, FL,Mar. 2007