# ARES project CONSOLIDER-INGENIO 2011 CSD2007-00004 Workpackage 2 - Task 4 (WP2.T4) Development of a middleware for service discovery over ubiquitous networks

Juan Vera del Campo, Josep Pegueroles

Universitat Politècnica de Catalunya. Departament Enginyeria Telemàtica.
1-3 Jordi Girona, C3 08034 Barcelona (Spain)
{juanvi, josep.pegueroles}@entel.upc.es

October 20, 2011

**Abstract.** This report summarizes the advances carried out for task WP2.T4 within the ARES project to improve the service and document discovery in ubiquitous networks. The report describes the works that have been performed by the ISG group to justify the task WP2.T4 "Development of a middleware for service discovery over ubiquitous networks".

## 1 Introduction

The objective of Work-package 2 - Task 4 (WP2.T4) of the ARES project was the development and provision of a software prototype complex enough to test the mechanisms and technologies proposed in WP2 and WP4.

Specifically, the ARES project seeks to enhance secure service discovery, by means of providing to P2P applications over ubiquitous networks with methods to announce and discover services regardless of the physical network and the temporal situation of services and users. The developed solution, that was outlined in WP2.T2 and WP4, considers the following services: denial-of-service prevention, identity impersonation, authorization and access control, privacy of users' profiles, and confidentiality of service data. In addition, during the development of the project, we identify a new treat against users of a P2P network. Those nodes that aid in the discovery, localization and provision of copyrighted material, even if it is not their intentions, may be prosecuted and find liable of a copyright infringement. In the development of the final prototype, we took into account this new thread.

The final prototype that we present as output of WP2.T4 is divided in three different modules:

- Multi Service Discovery System (MSD). A prototype of a service discovery middleware that proxies existing service discovery protocols adding security services and interoperability in ubiquitous networks.
- Secure Cooperative File System. A prototype of a distributed filesystem over P2P networks. This system is both a service to be found with the MSD and a testbed to deploy the searching algorithms provided by the MSD to localize documents.
- Two ways authentication for mobile devices. A prototype of a mechanism to authenticate users for remote services when they are using untrusted interfaces, as public computers.

## 2 Multi Service Discovery System

The Multi-Service Discovery platform (MSD) is a middleware that lets applications to discover and use services in heterogeneous, mobile and insecure network environments. The framework is highly modularized and can be used in a large range of devices, from mobile phones to desktop computers.

Over the years, many academic and industry-supported service discovery (SD) protocols have been proposed for specific networks. While efficient for the targeted environment, existing SD protocols prove to be inefficient or not applicable in different network settings as they have usually been designed based on specific network characteristics. A comprehensive solution must also address security, and mobility issues. Context-awareness and semantic knowledge, which have been seldom investigated in current SD protocols, also prove fundamental for the effective execution of applications in such service-rich environments. Indeed, the description of the environment, as bandwidth and cost of the network, is as important as the service description itself.

The MSD is not just a service discovery protocol, but a complete framework that allows inter-operation of different SDs and networks technologies, and the final connection between services and users. Efficient resource discovery, interoperability with existent and future SDs, interoperability between different network technologies and deployments, context-awareness of service description, secure transactions and management of the mobility of the devices are the features of this system. This system, for instance, will allow a user with a Bluetooth mobile to print a photograph on the fly in a near printer located in an Ethernet network. In

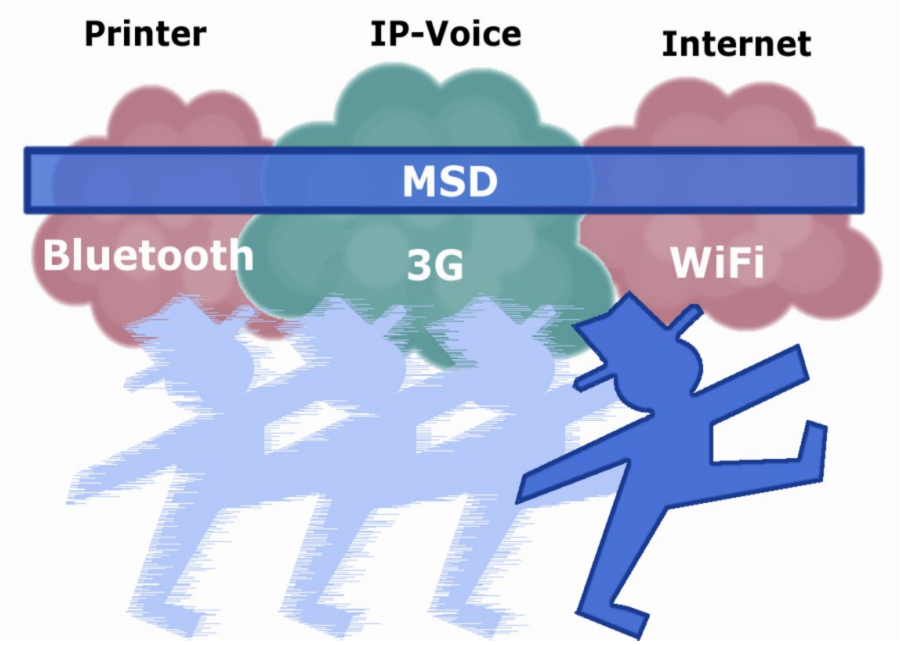



Fig. 1: The MSD in ubiquitous environments

the example, several printers can coexist and intruders could exploit it to register a new "malicious" printer in order to capture data from the fair user. To avoid this, authenticated registration of services should be warranted. Using the MSD platforms, the discovering of available printers, the discarding of the malicious one and the routing of data through distinct networks will be completely transparent to the client application, as figure 1 shows.

### 2.1 MSD description

The MSD provides a mechanism to users to discover and access all services services, documents and resources that are available in their current context. The MSD is generally transparent to the service provider and acts as a proxy between clients and providers. Despite this, provider can deploy MSD-aware services that make use of the more advanced abilities of the system.

*Network* The network module sets an abstract P2P layer between the MSD and the actual network interface. There is no need of a common address space or even a common network configuration. That is, the MSD can be set in a local WiFi network, in the wild of Internet or in a Bluetooth network. The MSD sends and receives packeted units called Messages. These messages can be unicasted, multicasted or multiplexed inside Connections. The MSDs identify each other by means of a Universal Unique Identifier (UUID). There can be MSDs in any network acting as bridges and routing messages between the network interfaces they have got.

*Groups* The MSD platform organizes users, services, providers and resources in groups. Currently, groups are not supported by the underlaying service discovery protocols and thus they are only supported by MSD-aware services. Groups as a security mechanism for authentication, authorization and confidentiality will be discussed in depth in Sec. 2.2.

*Proxies* The proxies' module is in charge of discovering the services offered in an EN. There is a proxy for each service discovery protocol integrated in the system. Current prototype includes proxies for the SLP and SDP protocols. In each local network a special group is formed and its leader runs the plugins for the SDs.

*Cache* The system cache stores all the information collected about the available services by the proxies' module. The information in the cache is shared between all the MSDs, and therefore an efficient sharing scheme is provided.

We define an entity running an MSD, or MSD for short, as a piece of software that includes all core modules listed above. In general, each MSD can take the role of client, server or bridge, depending on the needs. A single device is allowed to run several MSDs at once. This operation way is intended to run in a peer-to-peer fashion, without the need of a central repository. When a centralized server is needed (for instance for compatibility reasons with concrete Service Discovery Plugins as SLP) an algorithm executed among all MSD entities with complete functions will result in the election of an MSD manager performing all the centralized functions.

### 2.2 Security

Threats to Service Discovery protocols can be classified in three categories:

- identity theft: impersonating some user to take advantage of his privileges, and impersonating providers to fool clients to use their services.
- private data leakage: unauthorized third parties that collect private data from clients and servers, by inspection of the queries and service descriptions that are sent to the network.
- service eavesdropping: leaking information about the actual use of a service, or the downloading of a resource.

As seen in Fig. 2, security in the MSD platform relies on three security mechanisms: trust management, groups and secret sharing.

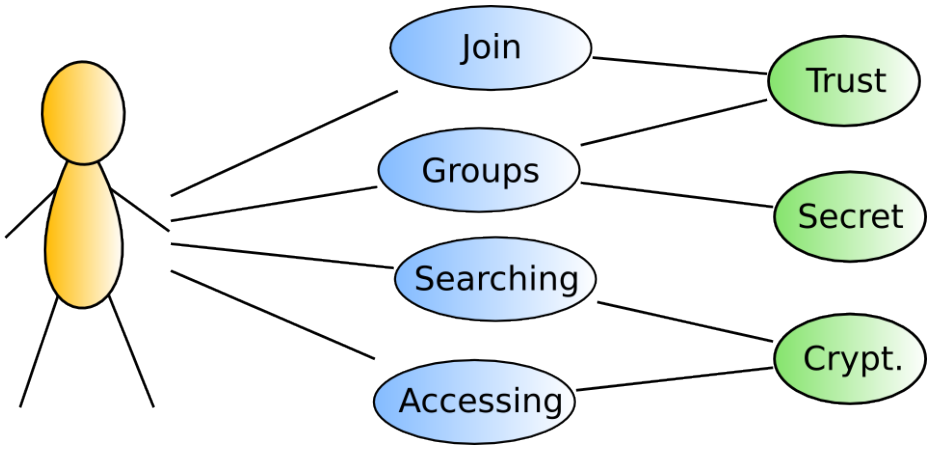



Fig. 2: Security phases and mechanisms of the MSD

*Trust Management* Authentication is the process of verifying someone's claimed identity. The MSD platform warrants that information that two agents exchange, comes from the sources that claim to be the origin. The MSD uses the trust model described in [1]. It is a mixture of a traditional certification agent approach and the web of trust of PGP. In this model, every node holds a credential that presents when joins to the network. If the certification agent of the node is available, other nodes can test the authentication of the new one. If nodes have not a direct connection with the certification agent (the 'offline mode'), a trust level is calculated based on the previous behavior of the node. Then, nodes test if the trust level of the new node is high enough to access to the desired resource.

*Groups* Users, services and nodes in a MSD architecture are organized in groups. The ability to manage groups is a main feature in the MSD

platform and are a central part of the security policy. Grouping nodes makes distributed authorization easier since if members of a group share a secret, privacy and confidentiality could be warranted inside the group members. In the MSD platform, each group is managed by a leader that must be elected among the members. The platform includes secure election and recovery mechanisms to elect a new leader when the previous one disappears [6]. When a new node wants to join to the group, the group leader try to authenticate it and checks its authorization level. If accepted, the node enters the group and gets a secret key

*Shared group keys* can be used to create secure and fast point-to-point connections between members of the group. Besides, they are a simple distributed group authorization. Since only members of the group know the shared key, it can be used as an easy and simple method to prove that an unknown entity is an authorized member of the group. Moreover, Group authentication is cheaper in computational cost and more efficient than individual authentication since keys are pre-known and it uses symmetrical encryption. Finally, a shared key could be used to distribute confidential notifications only to authorized members of a group. Every member of a group shares a secret that is used as the key of the group. This key changes after any risk situation, that is, when a new node enters the group, or nodes leave the group, or even periodical changes. Thanks to this mechanism the MSD can assure forward and backward security. A new node cannot decrypt messages prior of its joining, and leaving nodes cannot decrypt group messages any more. In order to achieve this group security, a secret distribution scheme is included in the group module of the MSD framework [10].

### 2.3 Real World Deployment

You can get the prototype from the a git repository or https://github.com/Juanvvc/sMSD:

```
git clone git@github.com:Juanvvc/sMSD
```

In order to demonstrate the abilities of the MSD platform, we developed a testbed supporting two different scenarios: one showing the deployment of the MSD without security, and the other including the security services [12].

**Insecure Scenario: Remote printing service** User A is holding a mobile phone with Bluetooth networking and a camera. A takes a photograph on the fly and wants to print it on a printer connected to an

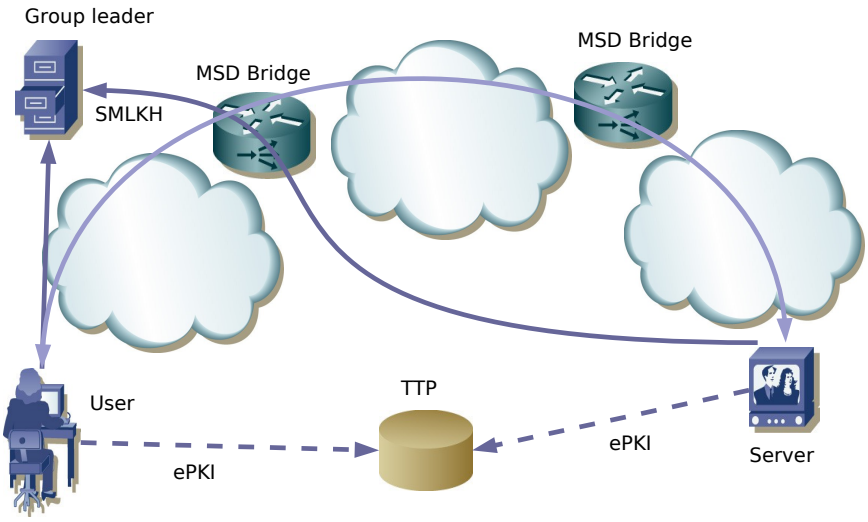



Fig. 3: Security mechanisms of the MSD

Ethernet network. The Ethernet network has a WiFi Access Point. Some devices in the WiFi network also have Bluetooth interfaces. These can act as bridges of the MSD platform. See Fig. 4.

User B publishes its printing service using SLP and the MSD platform detects and registers this service. Then, A takes a photograph and joins to the MSD network looking for available printers. The printing service in B is detected through the bridge M. Finally, A sends its photograph to B and it gets printed.

This scenario faces heterogeneity of service discovery protocols and network technologies. Although the printer publishes its service with SLP and the client only knows how to look for services in SDP, B is able to discover the service A. But discovering services is not enough in this scenario, and the MSD platform offers the possibility of connecting A and B even if they have different network interfaces.

Notice that this scenario does not face security: malicious printers can be registered and the data is sent in clear, so anyone is able to eavesdrop the photograph taken by A.

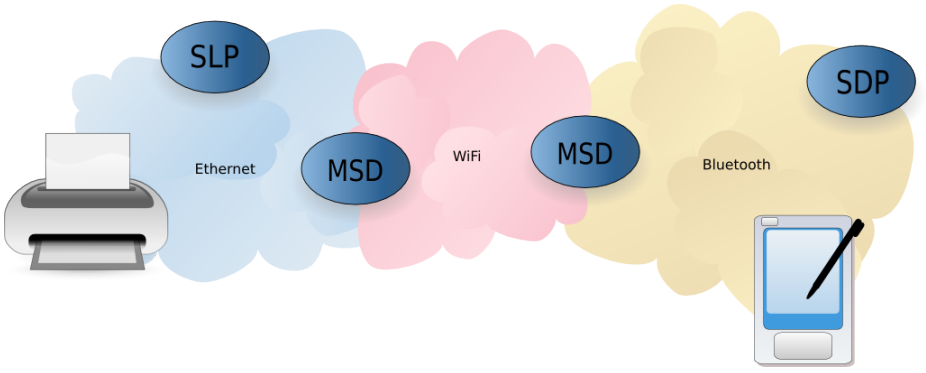



Fig. 4: Remote printing

**Secure scenario: File sharing** The scenario consists in a secure file-sharing peer-to-peer network. Each peer in the network runs the MSD as client and server of a simple file-sharing protocol. Each peer belongs to one or more groups, shares some files and is able to discover and download only the files belonging to the same group that itself.

This scenario has two major groups of interest: pupils and teachers. By no means should exams be under the greedy eyes of pupils and they must not to be allowed to change any question. On the other hand, teachers must authenticate themselves and the communication between them must be complete and confidential. This covers every security service in the MSD platform.

Teacher B holding a PDA with a Bluetooth network interface wants to get an exam E from the distributed exams database. He does not know where the file actually is, but his PDA can search for the desired resource. The network in the campus works in a decentralized fashion with many laptops offering bridging to the Internet. One specific laptop (M) offers an Internet gateway to Bluetooth devices. This bridge is controlled by students. Teacher B should be able to get the exam E from another teacher A using his Bluetooth PDA connection without noticing the students he is accessing an exam. See Fig. 5.

Previous to the MSD operation, different groups of users have to be established. We suppose the existence of two groups: teachers and students. Notice that teachers can prove their membership to the teachers' group by showing that they know the shared key.

Once the MSD is started, teacher A starts the announcement of exams and notes from his computer at home. Exams should only be discovered and accessed by other teachers, while notes are public. Then, the con-

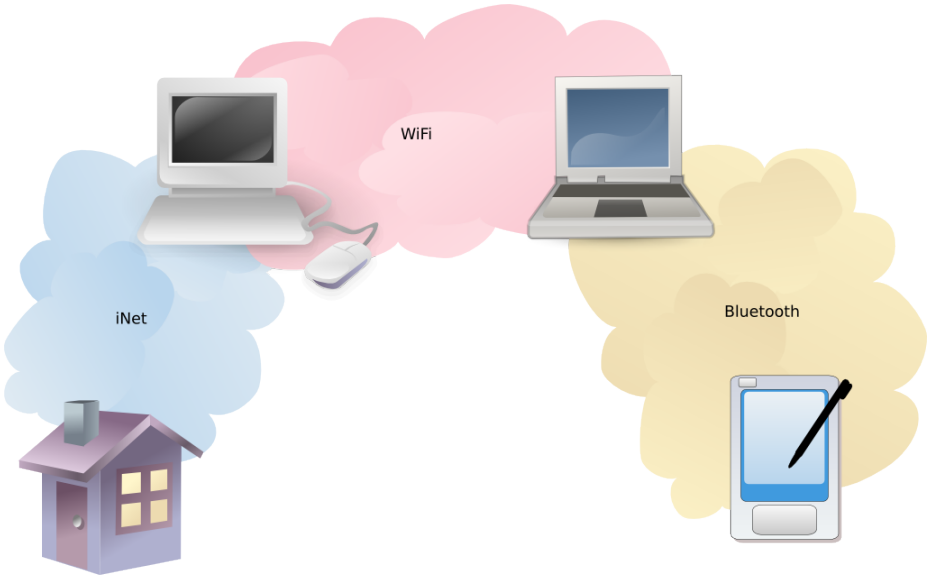



Fig. 5: Secure file sharing

tent of MSD service description and messages of the exams server are encrypted using the teachers group key, while the notes resources are sent in clear through the network. This prevents no one but teachers to know about the exams server. When teacher B starts the seeking of exams, he discovers teacher A's server because he is able to decrypt the service description. This includes the URL where teacher B must connect to download the exams. Besides that, stronger application level authentication can be provided upon accessing the service.

In this scenario, there are several possible treats against security. For example, all students or any other network member with an MSD agent can publish information as a server. However, since they do not know the key of the teachers group they won't be trusted by other teachers. In the other hand, since we are in open networks, the eavesdropping of messages cannot be avoided. However, by means of the ciphering mechanisms used in the MSD platform only valid MSD members of the teachers' group are able to read private messages. Finally, students or malicious members could misbehave as bridges. This can be considered a kind of Denial of Service attack. When detected, the recovery protocol is executed, discovering another bridge if available and then providing a trusted route to the final server.

## 3 Cooperative File System

After the development of a secure way to localize and access services and resources, we focused our efforts on the development of one of these services. In this way, we extended the testbed that was proposed in section 2.3. We thought that a distributed filesystem was very suitable for the task, since it is a very successful service that users demand, but also one where localization of documents owned by different users can be deployed.

Our proposal aims to add security to the **Cooperative File System** (CFS [2]), that is a distributed file system over a **Distributed Hash Table** (DHT). Figure 6 shows the architecture of CFS.

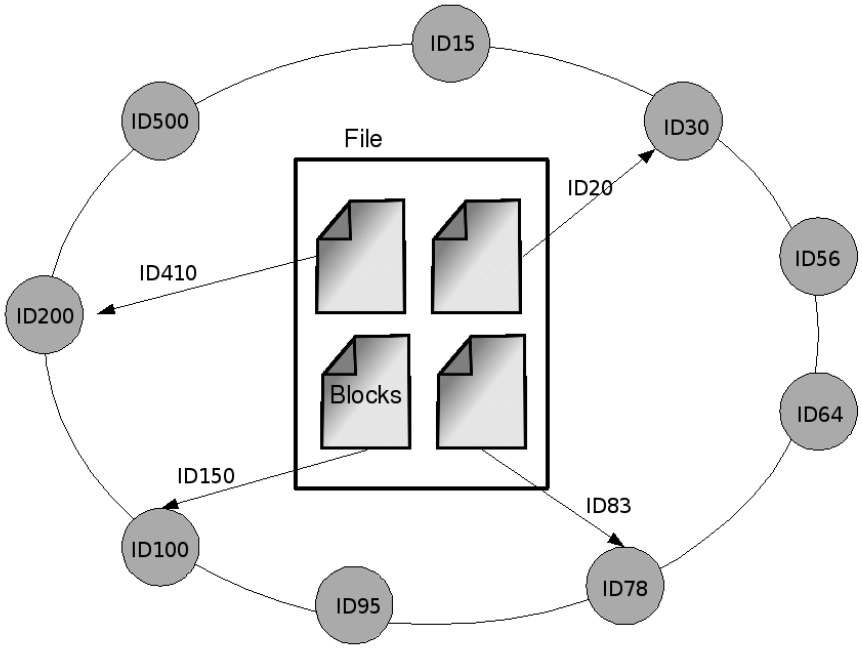



Fig. 6: Cooperative File System.

A DHT is a structured peer-to-peer network where nodes choose a random identifier $ID_i$. Each piece of data is store under a key that works in the same way that a node identifier. Each node in the network finds and creates a link at least to the node that has the very next $ID_n > ID_i$ in

increasing order. Then, the node $ID_i$ is in charge of storing a piece of data that is identified with a key $ID_i \leq ID < ID_n$. The process goes on until that the last node links to the first one and the whole network creates a structure with the shape of a ring. In order to PUT or GET the piece of data that was identified with $ID$, a node sends clockwise a message in the ring to the node in charge of $ID$. To improve the routing of messages inside the ring and minimize the number of hops that a message takes to reach its destinations, nodes of the network also link to many other nodes in the ring, according to a certain, foreknown algorithm. Current implementation of DHTs include Kademlia [8] or Chord [13], and the main difference between them is the specific algorithm to choose far links.

CFS was implemented over Chord. It gets a file $F$ with a filename $f$, divides the file in blocks, store the blocks $B_i$ and calculates $H_i$ and $ID_f$.

$$F = \{B_1 \cup B_2 \cup B_3 ... \cup B_n\} \tag{1}$$
$$H_i = hash(B_i) \tag{2}$$
$$F_h = \{H_1, H_2, H_3, ..., H_i\} \tag{3}$$
$$ID_d = hash(f) \tag{4}$$

Then CFS saves each block $B_i$ in the DHT identified as $H_i$ and stores the list of $H_i$ in a special block saved under $ID_f$. Files in CFS are persistent and the system warrants that every file will be retrieved, regardless of its unpopularity. This is a main advantage to store personal data and backups over other distributed filesystems that were proposed in the literature. Unfortunately, CFS authors did not consider security in the design of the system.

### 3.1 Prototype description

The Secure Cooperative File System (SCFS, [3, 4]), was implemented in a prototype accessible in [5] under the GPL license. The prototype was coded in Python on a Linux system, it uses a simplified Kademlia as the algorithm for the DHT, blocks have a 2048 bytes and there is a 30% of redundancy in the IDA algorithm. Identifiers of files, nodes and users have 128 bits. AES is used as the encryption algorithm, and the first 128 bits of SHA as a hashing algorithm. The reader is highly advised to download the prototype and perform a run test before proceeding any further.

You can get the prototype from the a git repository or https://github.com/Juanvvc/scfs:

```
git clone git@github.com:Juanvvc/scfs
```

The prototype runs on any Linux system that includes a Python interpreter. It doesn't need any special library apart from cherrypy, that will be included along with the prototype.

Please, refer to the README of the prototype for updated information.

**Command line interface** We provide a command line interface to manage the distributed filesystem and run the network nodes. This is the *dfsh.py* script, and it includes well known shell commands such as "rm" and "cp" in addition to node management commands such as "server" or "configure". The next text shows the parameters of this script.

```
Usage: dfsh.py [options] COMMAND command-opts

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -c FILE, --conf=FILE  Read configuration from FILE
  -p PWD, --pass=PWD    PWD for the configuration file
  -d LEVEL, --debug-level=LEVEL
                        Debug LEVEL for information
  -q, --quiet           Do not log messages
  -v, --verbose         Log messages
  -l FILE, --logfile=FILE
                        Log messages in FILE
  -S                    Development mode
```

First, the user will need to set up the environment for her node. She will run at least once the *configure* command, that asks for a username, a password and some information about the namespace. Then, the user runs a new local node.

```
./dfsh.py configure
./fssh.py server
```

Currently, we do not provide with any bootstrap process to join the network. If you want to set up a node or connect to an existing p2p network with several nodes on different computers, you may need to manually configure your configuration file. See section 3.1 for further details.

Next, the user is ready to save any local file into the distributed file system. In this example, we will use the root of the filesystem that the user associated to himself during the configuration.

```
./dfsh.py  cp digital2.jpg dfs:///digital.jpg
```

Current implementation has not efficiency as one of its goals. Sending a large file to the filesystem may be a very long process. In order to run the test under a reasonable time, the reader is advised to store in the filesystem only small files, under 100KB in size. In order to get this file, run:

```
./dfsh.py cp dfs:///digital.jpg digital2.jpg
```

The user is able to delete distributed files and create directories in a similar way.

**Configuration files** Next we describe the configuration files that SCFS uses. These configuration files are associated to a user, namespace or directory, as we will see in section 3.2.

```
[Main]
nick = juanvi
uid = juanvi
root = dfs://juanvi@juanvi/root

[DHT]
datadir = dhtdata
maxbuffer = 4096
descpermetapart = 12
block = 1024

[Ring]
id = 28790746847835588597651173935682614952 5
ip = 341.341.1.1
port = 12345
known = 341.341.1.2:12345, 341.341.1.3:12345
```

The configuration files are organized in different sections:

**Main** includes the description of the user and the URL of the directory that this configuration file manages. You will find additional information about the various namespaces in section 3.2.




Fig. 7: Login form

**DHT** configures some DHT parameters, including the local directory where data will be saved. Currently, we include several implementations of DHT as a playground to check several parameters of the network. This section includes the size of the blocks of the filesystem or the size of the buffer. This parameters will be described in section 3.2.

**Ring** includes the description of the local node and a reference to remote nodes. For example, the identification of the local node or the local port should be included in this section.

This configuration file should be carefully managed by the users. They must be kept privately, and they should be provided to the different interfaces to access the system. The user can manually copy her configuration files to a new device to access to her private files, but she should change the configuration of the **Ring** section.

These configuration files can be encrypted.

**The web interface** The prototype includes a web interface that the client can use to access to her private files. You can access to this interface running *webserver/webserver.py* and pointing your browser to "http://127.0.0.1:9090". A page like the one in Figure 7 will be presented.

Here, the user can enter her pair username/password or enter a previously created configuration file. A new user can register a new profile




Fig. 8: View of the user's files

and join the distributed system, as well. Once logged in the system, the user is allowed to perform common actions for a filesystem: creation of a new directory, uploading of a file, downloading from the filesystem and deleting a file. From the page shown in figure 8, the user is able to access to her global configuration file and to the local configuration file of her directories.

The web interface is not intended to be run on a remote server, but being a local service inside the computer that the user controls. The reader should take into account that configuration files never should leave the control of the user.

The web is only an interface to the filesystem, but you will need at least a node in the network to store the data. We refer to section 3.1 for instructions to run a network of several nodes.

### 3.2 Secure Cooperative File System

This section details the mechanisms of the Secure Cooperative File System (SCFS, [3, 4]). A graphical outline of the whole process is shown in Figure 9.

**Assumptions and Definitions** Table 1 includes the definition of the main concepts of SCFS. The reader can find a relation of symbols in table 2.

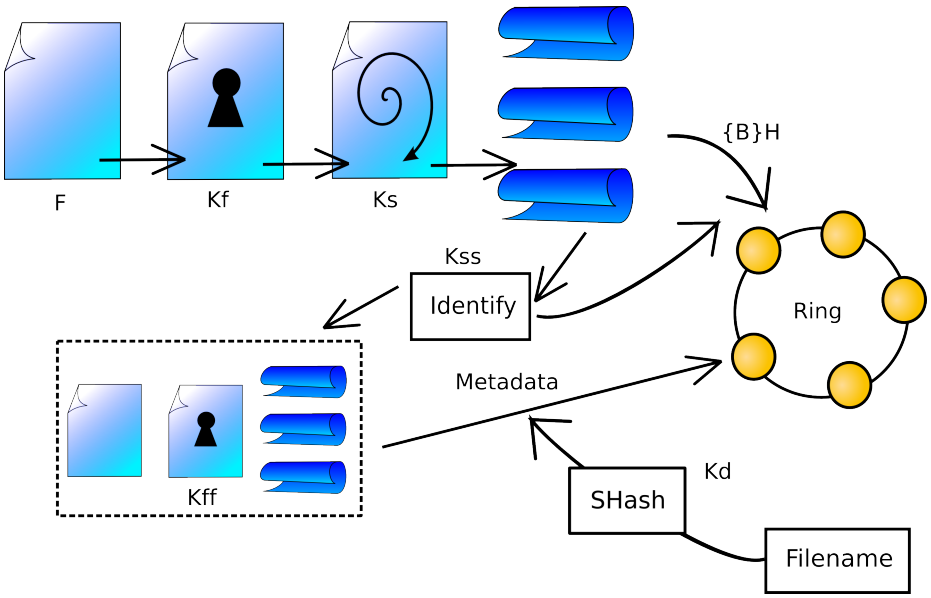



Fig. 9: Description of the process.

For example, Bob and Alice join to a SCFS network. The both desire to store a file that has "revenues.xls" as filename. Bob has $U_{bob} = bob$ as his identifier, and Alice $U_{alice} = alice$. Bob has $cfs : //bob/root$ as the root directory, and Alice has $cfs : //alice/root/$. In order to access to the personal root directory of Bob, she will need her global key $K_d$. Since their user identifiers and $K_d$ are different, their namespaces will be different as well, as the next section will show.

Next, we detail each one of the steps of figure 9 to protect the file.

**Securing the whole file** The first step to secure a file is encrypting its contents. Each file is encrypt with a symmetric algorithm with $K_f$ In our prototype we choose AES as the encryption mechanism.

After the encryption process, a shuffling and redundancy creator algorithm takes place. SCFS uses the algorithm IDA described in [11]. This step has a double objective. The first one is preventing that consecutive bytes in the original file were consecutive in the stored blocks. This shuffling prevents some kind of statistical attacks against files with a known structure or common header, such as PDF files. Since users can choose the kernel space of vectors for the IDA algorithm, the distribution of data inside the final blocks can be determined only by the original data owner.

| | |
|---|---|
| User | A client of the system. Each user has a unique identifier $U$. Collision of identifiers of users are not managed in this work |
| File | An ordered array of data |
| Directory | An unordered set of files and directories |
| Filename | A human readable identifier for a file or directory. It is not unique in the system, since two different users can store different files under the same filename |
| Root Directory | The filename of the directory that holds every file and document of the user. In SCFS it is not mandatory for documents to be included to a directory and that users can use arbitrary, binary identifiers for their documents. In order to provide a user friendly environment, we assume that users have a root directory for all their documents and files. |
| Block | Each file is divided in blocks that are created with the IDA algorithm. Only $k$ blocks of $N$ are needed to recreate the original file |
| Metadata Blocks | Special blocks of data that have enough information to restore the whole file |
| iNode | The first block of metadata that holds a list to the rest of metadata blocks |
| Config file | It is a file that stores the identification of keys used in each file and the personal information of the user, as her root directory. |
| Nodes | Each one of the devices that a client uses to join to the network. Each node has a different identifier. |

Table 1: Definitions

Key $K_{ss}$ is used to generate the vector space that the IDA algorithms need. The second objective of the algorithm is creating data redundancy. In order to enhance the availability of the service, the algorithm takes the original file and created $m$ blocks in such a way that it will need $k < m$ blocks to restore the original file. The dimension of $B_i$ is chosen in such a way that it matches the size of a block in the DHT. In this sense, the system is protected against malicious nodes that randomly changes nodes, overloaded nodes than cannot manages all their parallel connections and local network failures.

$$S = \{F\}_{K_f} \tag{5}$$

$$B = \{S\}_{IDA} = \{B_1, B_2, ..., B_i\} \tag{6}$$

$$dim(F) = dim(S) < dim(\bigcup B) \tag{7}$$

Clearly, this algorithm consumes both time, bandwidth and disk space in remote nodes since the dimension of the final data in $B$ are greater

| | |
|---|---|
| $U$ | The identifier of each user. $U$ is the same for every node that a user controls. |
| $F$ | The original file of the user |
| $ID_f$ | The final identifier of a file |
| $S$ | The encrypted file of the user |
| $B_i$ | Each one of the blocks of a file |
| $B$ | The set of blocks of the file |
| $H$ | The set of identifiers of blocks |
| $K_d$ | The key used to secure the filename |
| $K_f$ | The key used to encrypt the file |
| $K_{ff}$ | The key used to encrypt the metadata |
| $K_s$ | The key used to generate a vector space for the IDA algorithm |
| $K_{ss}$ | The key used generate pseudo random identifiers |
| $K$ | It is the set of keys used to secure each file. This set contains five conceptual keys associated to a single file. The complete set $K$ can be generated from a master key or stored individually for each file. |

Table 2: Symbols used

than the dimension of the original file $F$. Users can choose the amount of redundancy to apply to each file, or even if this algorithm runs or not at all over files. Authors expect that clients will use most of the time the default redundancy of 30%.

The encryption and shuffling steps take place locally to the user and the set $B$ is not yet published in the ring.

**Securing file blocks** After the previous step each block $B_i$ has the same size than the DHT blocks. Since the data in each $B_i$ were encrypted and shuffled, it makes little sense to any casual attacker than sniffs the blocks.

The CFS stores and retrieves blocks of data associated to a identifier. We use the **secure hash of the block contents** to identify a block.

Using hashes as block identifiers has an important drawback. A file of 20MB stored in blocks of 2048B need about 160KB to store the block identifiers. Since SCFS uses special blocks to store file identifiers, this approach needs at least 80 blocks just to store the file structure. Furthermore, since hashes will be protected with a key $K_sss$, the client must save $K_{ss}$ somewhere. Anyway, hashes have the advantage that they could be used to ensure integrity of data or event prevent unauthorized overwriting, as will be stated in next sections.

The set of block identifiers of a file is $H$.

**Metadata** In this moment the user's node has a local buffer with the blocks of the file $B$, a set $H$ to identify each block and a configuration file, as stayed in section 3.2. Many of these data are stored in a special block that we call iNode, in the same sense that iNodes in the traditional filesystems. iNodes store configuration, identification of the set $K$ and the whole contents of the table of block identifiers $H$. Metadata are then managed in the same way that a file object. It is padded, encrypted with $K_{ff}$ and divided in blocks using IDA.

Metadata keep lots of sensitive information. SCFS accepts two different mechanism to protect these special blocks:

- Metadata blocks are not published in SCFS but stored locally in the nodes controlled by the client. In this way, since attackers have no access to the list of blocks of a file, they cannot even gather them to begin with any cryptanalysis. Unfortunately, keeping this data locally means that clients must use always the same node or devise an extern mechanism to distribute these file to other nodes.
- Metadata blocks are published in SCFS as regular blocks. Since they have the same length and entropy as any other block, they cannot be distinguish from regular blocks. Identifiers are assigned in the same way as block identifiers and each block keeps a reference to the next block of the chain. This prevents shuffling metadata as in regular file objects. The first metadata are identified in a special way, described below.

In the case of publishing metadata blocks, the first block is referred with a special identifier. Metadata blocks are then randomly introduced in the local buffer. In this way an eavesdropper cannot discriminate between file blocks and metadata blocks. Our prototype implements both methods.

*Securing File Identifications.* Every file in SCFS is related to a URI that identifies only one file. Furthermore, since nodes in the network saves part of files, it must be not possible to get the URI out from any number of parts of the file.

Since users store in SDFS their private data with an arbitrary name, the file system must provide with a separate namespace of files for each one of the users. In the other hand, users may want to share their file to any other user of the network, so there must be a common namespace in the whole network.

Our filesystem supports a human readable identification of files. In this sense, a human readable and writable string must be the main interface of

the user to the system. The complexity and security involved in mapping that string to a DHT identifier must be hidden to the user by the system.

If an attacker is able to identify the first block of metadata and somehow access, to its contents, she may be able to retrieve the complete metadata and then the complete file. In this sense, securing the identification of the first metadata block is crucial for the security of the whole system. The next section will study several mechanisms to identify this first block of metadata from the human readable filename.

*Public and private identifiers.* There are two possible namespaces in SCFS: user's namespace and public namespace. In both namespaces, users must identify their files with a unique filename. Using directory paths as part of the identifier is an easy mechanism to warrant this unicity in the private namespace, but it is a really hard problem in the public namespace that is out of the scope of this research.

It is up to the users to ensure that filenames are unique in the public namespace. The filename is then hashed with SHA and cut down to 132 bits in order to get the file identifier. Every user that knows the filename can get the metadata and then every block of the file. As stated in section 3.2, this does not mean that the attacker has access to the plain data of the file, but she is on the right way.

$$ID_f = hash(filename) \tag{8}$$

Each user has a private key $K_d$. The filename is hashed as in the public namespace and then encrypted using AES. The hashing ensures a 132 bits input to the encryption, and then maximizing the entropy of the encrypted string. The encryption step warrants that the userspace is unique and secured, since none can identify the file even knowing the author and the filename.

$$ID_f = hash_{K_d}(filename) \tag{9}$$

**Publishing the file** At this moment, the node has local buffer with a set of blocks $B$ that are identified with $H$ and the iNode of the system identified with $ID_f$. The node publishes the contents of the local buffer in the DHT under its keys. The reader should notice that since blocks, metadata and the iNode were randomly introduced in the local buffer, an attacker is not able to distinguish between them and then a cryptanalysis is really more difficult.

**Reading process** The reading process is the inversion of the writing process. From a filename users create a $ID_f$, maybe using a user key $K_d$ to maintain confidentiality. From $ID_f$, the user gets the iNode of the file and then the list of metadata block. From the metadata, the user is able to recreate $H$. Then, she downloads the blocks, reorders and the decrypt the file.

Typical DHTs warrant than a block can be retrieved for a week. This time may be enlarged with a ping from the user. In this sense, data may be indefinitely stored on the network and the DHT structure warrants that it is accessible nearly forever. Moreover, the IDA algorithm warrants that the file can be restored even if only $k$ blocks are accessible in the network. Furthermore, files can be retrieved from everywhere in the network, so users do not have to contact to the same neighbors each time that they access to the network with different nodes.

It is easy for an attacker to identify all or some of the blocks of a user. He only has to eavesdrop identifiers from the petitions of the user. In this sense, it is easy for an attacker to save junk under the same identifier than valid users. Even fair users may store her own data under the same identifier, and their data will collide with previously stored information.

DHTs let storing several blocks of data under the same identifier. When a user asks for a key, she gets every block that was published with the same identifier. If the iNode includes the hash of the block, as stated as one of the possible options of section 3.2, it is easy to discard random and malicious collisions. Sending the hash of the block to the final node and letting to compute the hash to itself saves space, but augments the bandwidth of the system. The redundancy introduced by the IDA algorithm can help in this phase.

**Additional mechanisms** Apart from the mechanisms of the previous sections, our implementation of SCFS includes additional mechanisms to enhance the security of the system. In this section we will study these additional mechanisms that takes place in phases other that reading and writing data.

Attackers may join a large number malicious nodes in order to perform Sybil attacks. In this way, the attacker gains a large influence in a region of the Kademlia ring and she may be able to put the whole system down or perform denial of service attacks to some users. In order to prevent this kind of attack, the original creators of CFS proposed that the identifiers of the nodes must depend on the network address of the node. Furthermore, the random identifiers of each block disperse blocks in the whole ring,

while the information dispersal algorithm warrants that the file can be retrieved even if a segment of the ring is not available.

Attackers may collect user information by means of sniffing the communications of users in the network. Files are encrypted, shuffled and split down in blocks as explained in section 3.2, but metadata have a slightly weaker encoding process, especially the first of them. An attacker that gets the first iNode needs to decrypt a single block of data to get the list of every blocks in the system. That first iNode is encrypted using AES, but it contents known information that may simplify the cryptanalysis. An attacker may identify this first iNode because it will be the first block than a user demands. In this way, SCFS asks for several random blocks apart from the iNode.

There is no deletion service in SCFS. A deletion service need to authenticate the owner of the file in order to prevent deletion from unauthorized users. In order to enhance privacy of the users, SCFS does not include any authentication mechanisms. In order to prevent size growing with time, blocks are really deleted in nodes if user do not access to the block after a month. In order to show interest for a block without the need of downloading it and increasing the bandwidth, users must send a "ping" to the data block. Nodes count these pings as an access and will mark the block as no removable for an extra month.

Apart from deletion, both malicious and fair users may overwrite blocks of legitimate users. Since identifiers of blocks are random numbers, collisions are possible. Kademlia lets different blocks under the same key, and when a user demands that key she will receive the whole collection. SCFS takes advantage of this by means of saving the hash of the block in the iNodes. In this sense, when a user demands a key and receives several blocks, she is able to discriminate which one is the valid to create the original file. This mechanisms does not prevent to malicious users to write blocks under the same key, but their blocks won't be used to recreate the original file.

## 4 Mobile interface

As discussed in section 2.2, when the user authenticates in the proposed system, he joins one or several groups and perform his searches. Thus, authentication is an important security service that must be offered by service discovery systems. According to the groups of the user, the base network or the requirements of the service, the initial process of authentication may be different. We worked in the most restrictive scenario,

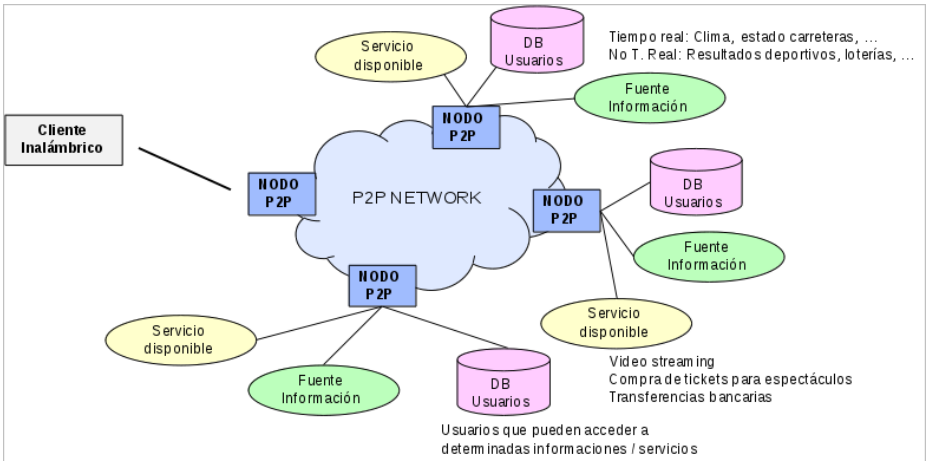



Fig. 10: Mobile deployment

forcing mutual authentication of both parts using a mobile phone system [7].

We propose an authentication system [7] based on three different parameters [9].

*Something that the user knows* The first step to authenticate a user is ask him for something that only he should know. In our system, this is carried out using a simple name/password to log in the system. The main objective of this step is avoiding random attacks, since it cannot be used to authenticate the server, and we judge as not enough to authenticate the user.

*Something that the user has* The second step for our authentication system is the user proving that he owns a certain object that the server administrator gave to him. We use a challenge-response protocol with an additional channel. As figure 11 shows, the mobile device M is used to solve a challenge set by the server S. S shows on the screen of C a text that the user introduces in the mobile phone. The text is signed with S's private key, and then M can prove its authenticity. The introduction of the text in M can be manual, or using any of the automatic systems that improve the usability of the system. M sends the response using an alternate channel that C cannot control, as an SMS or a GPRS connection. This protocol is able to authenticate both the mobile phone and the server.

*Something that the user is* Finally, the last step is proving something that the use is. Usually, this was performed using biosensors to check the fingerprints, voice recognition or the iris of the user. The major difficulty of this test is the limited capabilities that the mobile phone that the user owns has. In the described scenario, this check must be performed by the mobile phone. If the user passes the test, the mobile phone authenticates the user. Since the second test authenticated the mobile phone, using a chain of trust, the server authenticates the user as well.

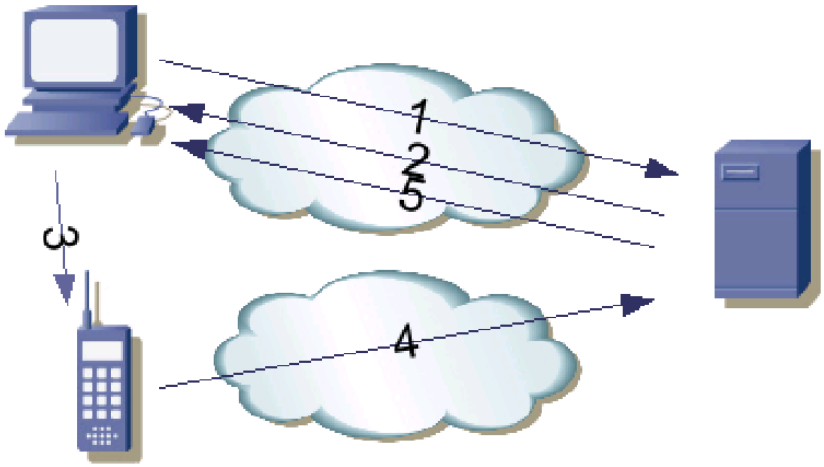



Fig. 11: Challenge response protocol

## 5 Conclusions

While designing and developing this middleware, we got new experience in the field of resource discovering and access for heterogeneous networks and devices.

There is no point in discovering a resource that cannot be accessed, so any middleware looking for resources must provide a way to access them as well. Unfortunately, many services in the past were designed in a single scope network. That is, they do not provide SOAP, CORBA or proxy protocols. In order to access these services, the resource discovering middleware must provide a transparent point-to-point connection through bridges, emulating traditional input-output streams. Besides, the middleware should support modern access such as SOAP. In this sense, we think that asynchronous message-oriented communications matches both criteria, and this kind of communications was developed into the MSD.

In the other hand, the MSD should be able to handle a wide spread of devices with different network and processing capabilities. Services must provide flow-control techniques in order to overcome the problem of flooding little devices with flows of information that cannot handle. Besides, the flow-control mechanism should be intelligent enough to be able to adapt itself to the changing conditions in network load and route configuration through bridges. The TCP dynamic-window flow-control is a good inspiration protocol.

The MSD scenario defines devices in a wide sense, from mobile phones to large desktop computers. Java is a fantastic language to develop new applications for a big number of devices, since it has good serialization tools to handle RMI and proxies. Unfortunately, most current services are not written in Java. Rewriting all those services is neither desirable not affordable, so the middleware should not be bounded to a single language as Java. With this idea in mind, we developed the MSD protocols over standard ones such as HTTP, XML and SSL. In this way, developers using the MSD can choose any language to develop clients and services in the GN.

In ad-hoc networks, flooding the network with discovery messages is an efficient, quick and simple technique to search available resources. In the other hand, this mechanism is extremely inefficient in large and global networks. Besides, different network and context circumstances could convert an efficient searching protocol in the worst possible ever. In

this sense, the MSD must be able to use different searching mechanism, and adapt them to the changing circumstances.

Finally, regarding security, in pervasive computing the trust management is a critical problem. Clients must trust or not in never seen nodes according to their policy and needs. In this sense, the MSD joins traditional credentials and trust level based on context to provide a distributed trust management that handles authentication. In order to distribute authorization as well, the MSD uses the group mechanisms. Groups are an efficient way to handle security, and in this sense they were chosen as the main mechanism for the security module.

## References


1. F. Almenez, A. Marin, C. Campo, and C. Garcia R. A Pervasive Trust Management Model for Dynamic Open Environments. In *First Workshop on Pervasive Security and Trust at MobiQuitous 2004, Boston, USA.*, 2004.
2. Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM Press.
3. Juan Vera del Campo, Juan Hernández-Serrano, and Josep Pegueroles. Análisis de seguridad en un sistema de archivos distribuido. *X Reunión Española sobre Criptografía y Seguridad de la Información (RECSI)*, 2008.
4. Juan Vera del Campo, Juan Hernández-Serrano, and Josep Pegueroles. Scfs: Design and implementation of a secure distributed filesystem. *SECRYPT*, 2008.
5. Juan Vera del Campo, Juan Hernández-Serrano, and Josep Pegueroles. Scfs: lewis.upc.es/svn/dfs, 2008.
6. J. Hernadez-Serrano, J. Pegueroles, and M. Soriano. GKM over large MANET. In *IEEE International Workshop on Self Assembling Wireless Networks*, 2005.
7. Joan Tomàs i Buliart, Juan Vera del Campo, Josep Pegueroles, and Miguel Soriano. Diseno de una plataforma segura de e-gobierno. 2007.
8. P. Maymounkov and David Mazières. Kademlia: a peer-to-peer information system based on the xor metrid. In *IPTPS*, pages 53–65, 2002.
9. Alfred J. Menezes, Paul C. van Oor, and A. Vanstone. *Handbook of applied cryptography.* CRC Press, 1996.
10. Pegueroles, Wang-Bin, Soriano, and Rico-Novella. Group Rekeying Algorithm using Pseudo-Random Functions and Modular Reduction. *Lecture Notes in Computer Science*, 3032:875–882, 2004.
11. Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335 – 348, April 1989.
12. rick.upc.es. MSD Prototype in github, 2011.
13. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.