jDHTUQ for *peer-to-peer* DHT networking

(This document is under construction, but contains information that can be useful for understanding the functioning and use of JDHTUQ)

Index

Introduction

jDHTUQ is a *peer-to-peer* DHT system based in Chord algorithm, but built to generalize the implementation of *peer-to-peer* DHT system. It have two fundamental services, put and get of resource.

jDHTUQ is enfoqued to:

- Layered Architecture
- Lower coupling
- Easily adaptable to any routing algorithm
- Easily adaptable to any resources management
- Independent communication module configurable
- Implementation of two forms of communication, one in data struture and another on the network
- Xml properties files
- Xml-based communication
- One implementation of Chord algorithm
- One implementation of a resources management (DHash)
- Gui-oriented education

The project is conformed by a communication module (contains log4j from apache), interface of storage services, interface for lookup services and two implementations, Chord like lookup service and DHash like storage service.

The communication module is designed for be customizable and reusable. It have two general implementations, communication on a data structure (exist graphics user interface where you see the lookup and transfers) and on a network (implements for two branch, protocol UDP and TCP).

The interface for storage services and lookup services defines all services necessary for implementation of resources management and overlays network (like Chord, Kademlia, Pastry, Biseroy, etc).

Chord layer is an implementation of overlay network and exposes all services from the interface for lookup services. DHash layer is an implementation of resources management and exposes all services from interface for storage services. This is made for that components are loosely coupled. For this, is possible to implement any overlay network that implements the interface lookup services and use the same layer of resources management.

The Aplication jDHTUQ

2.1. Installation

Download from https://sourceforge.net/projects/jdhtuq/files/ jDHTUQ-1.0.0.zip and unzip, execute jDHTUQ-1.0.0.jar or (if operative system is windows) jDHTUQ-1.0.0.bat for to show console and to see the logs.

2.2. Set of components

The jDHTUQ-1.0.0.zip contains a GUI and libraries (Figure 1):

Figure 1: jDHTUQ files

The folder lib contains all libraries need for well function of jDHTUQ. This libraries are: chord-1.0.0.jar, dhash-1.0.0.jar, storageService-1.0.0.jar, lookupService-1.0.0.jar, communication-1.0.0.jar and log4j-1.2.15.jar (Figure 2).

Figure 2: Libraries files

Some libraries have configuration files, the following are the libraries with their own configuration files:

Library	Configuration File
chord-1.0.0.jar	chord_properties/chord.xml
	chord_properties/communication.xml
dhash-1.0.0.jar	dhash_properties/dhash.xml
	dhash_properties/communication.xml
log4j-1.2.15.jar	communication_properties/logger.xml

The properties for each file will be covered later.

2.3. Executing aplication

Exist two modes for execute aplication: Data structure and network. Both modes depend on which classes handle the communication. This is defined through of configuration files.

The communication was use in Chord layer for lookup and for storage service in the DHash layer. Both configurations MUST be in the same mode.

Before of execute the aplication, you MUST setup the communication mode. The configurations files are dhash_properties/communication.xml and chord_properties/communication.xml.

2.3.1. Data structure mode

Next we will show the correct configurations for data structure mode (Figure 3):

```
1<?xml version="1.0" encoding="UTF-8"?>
2
3<communication xmlns="http://www.DHT-UQ.org/communication"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://www.DHT-UQ.org/communication communication.xsd ">
6
7    <instance class="co.edu.uniquindio.utils.communication.transfer.structure.CommunicationManagerStructure" />
8
9    <time waitingResult="2000" />
10
11   <params>
12   </params>
13
14</communication>
```

Figure 3: Configuration for data structure mode

The core of the configuration is in the line 7, the property instance.class MUST to be co.edu.uniquindio.utils.communication.transfer.structure.CommunicationManagerStructure for both, for DHash and Chord. The data structre mode not required params.

Next you can execute the application from jDHTUQ-1.0.0.jar or jDHTUQ-1.0.0.bat (only on windows system), when you do this must be display the following window:

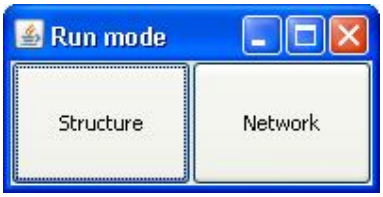


Figure 4: Main window

Selected *Structure* and must display the main window for data structure mode:

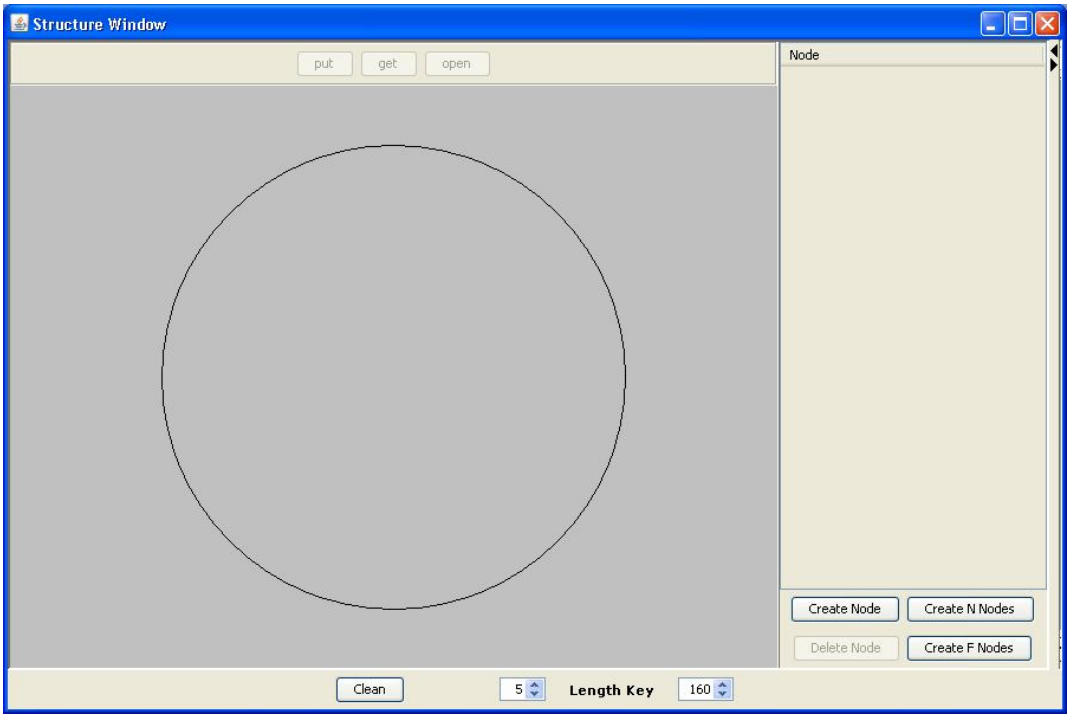


Figure 5: Main window in data structre mode

2.3.2. Network mode

For network mode exist two class to communication management, the first on protocol UDP and the second on protocol TCP. Commonly protocol UDP is use for lookup services and protocol TCP for storage services.

The classes for communication management in network mode are:

- co.edu.uniquindio.utils.communication.transfer.network.CommunicationManagerTCP
- co.edu.uniquindio.utils.communication.transfer.network.CommunicationManagerUDP

Both communications use protocol UDP Multicast for discovery and join of nodes.

The Figure 6 shown the correct configurations for Chord layer in chord_properties/communication.xml and the Figure 7 for DHash layer in dhash_properties/communication.xml. The parameters in both configurations are required and it are customizables.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<communication xmlns="http://www.DHT-UQ.org/communication"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.DHT-UQ.org/communication communication.xsd ">

    <instance class="co.edu.uniquindio.utils.communication.transfer.network.CommunicationManagerUDP" />

    <time waitingResult="2000" />

    <params>
        <param name="BUFFER_SIZE_MULTICAST">1024</param>
        <param name="IP_MULTICAST">224.0.0.2</param>
        <param name="PORT_MULTICAST">2000</param>
        <param name="PORT_UDP">2003</param>
        <param name="BUFFER_SIZE_UDP">1024</param>
    </params>

</communication>
```

Figure 6: Configuration for network mode in Chord layer.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<communication xmlns="http://www.DHT-UQ.org/communication"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.DHT-UQ.org/communication communication.xsd ">

    <instance class="co.edu.uniquindio.utils.communication.transfer.network.CommunicationManagerTCP" />

    <time waitingResult="2000" />

    <params>
        <param name="BUFFER_SIZE_MULTICAST">1024</param>
        <param name="IP_MULTICAST">224.0.0.1</param>
        <param name="PORT_MULTICAST">2000</param>
        <param name="PORT_TCP_RESOURCE">2001</param>
        <param name="PORT_TCP">2002</param>
    </params>

</communication>
```

Figure 7: : Configuration for network mode in DHash layer.

Only one node must to be executed in the same machine. Verifies that are enable the communication on protocol TCP, UDP and Multicast UDP in this machine.

Next execute the application from jDHTUQ-1.0.0.jar or jDHTUQ-1.0.0.bat (only on windows system), must display the window in Figure 4.

Selected *Network* and must display the main window for network mode:

Figure 8: Main window in network mode

Using Graphics User Interface

3.1. Data Structure GUI

The data structure GUI is composed by two main components, routing and storage, and hashing utilities.

3.1.1. Routing and storage

Next we will show the window components:

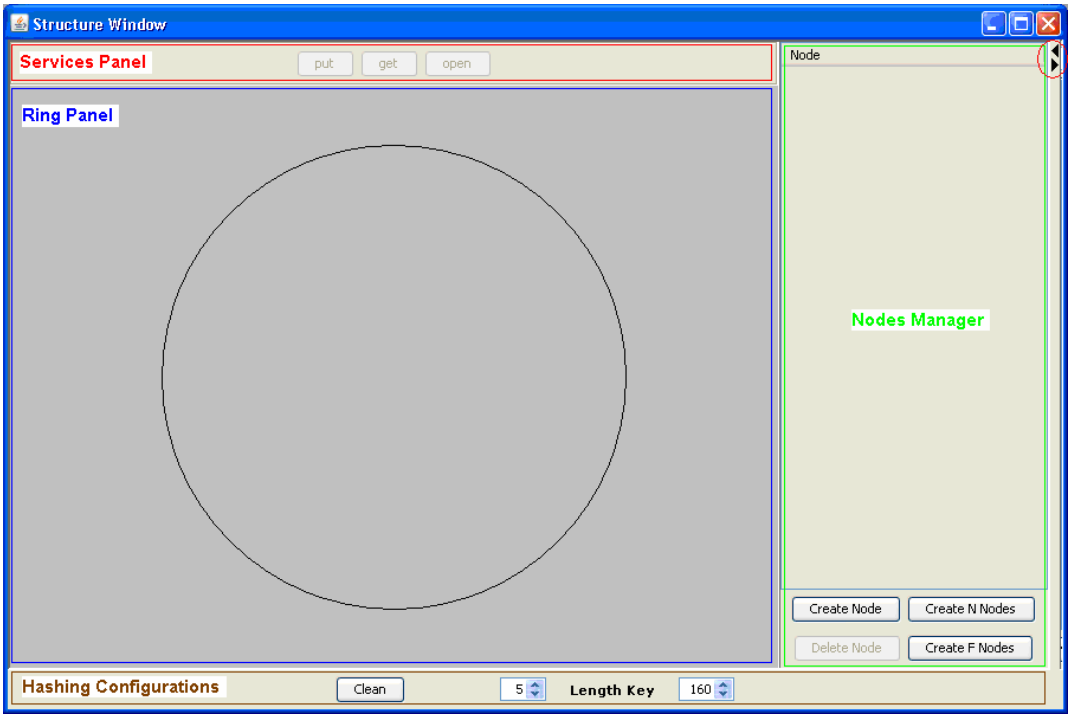


Figure 9: Data structure window and its components.

3.1.2. Specifing length key

The length key is the length of number hashing in BITS. Represents the size of fingers table (for lookup service Chord) and therefore impact of stabilization time of network. To configure the length key use the spinner in *Hashing Configurations* Figure 9 (brown color).

Note: For hashing is used SHA-1. If is lower length key to 160 bits, NOT guaranteed that all hashing generated will be diferents.

3.1.3. Creating nodes

There are four ways to create nodes: *Create Node, Create N Nodes* and *Create F Nodes*. This operations are in *Nodes Manager* Figure 9 (green color).

Create Node creates a node with the specified name. If not name is specified, the node name is generated automatically like number incremental. If you want, you can simulates behavior in the network using internet address for node name.

Create N Node creates a determinate nodes number. Must set a number greater that 0. The nodes names is generated automatically like number incremental.

Create F Node creates a determinate number based in text file with nodes names separated by ENTER.

All nodes created are shown like list in *Nodes Manager* Figure 9 (green color) linking node number and name, also are shown in *Ring Panel* Figure 9 (blue color) like a point in circle with node number (Figure 10). The order of list is with respect to hashing.

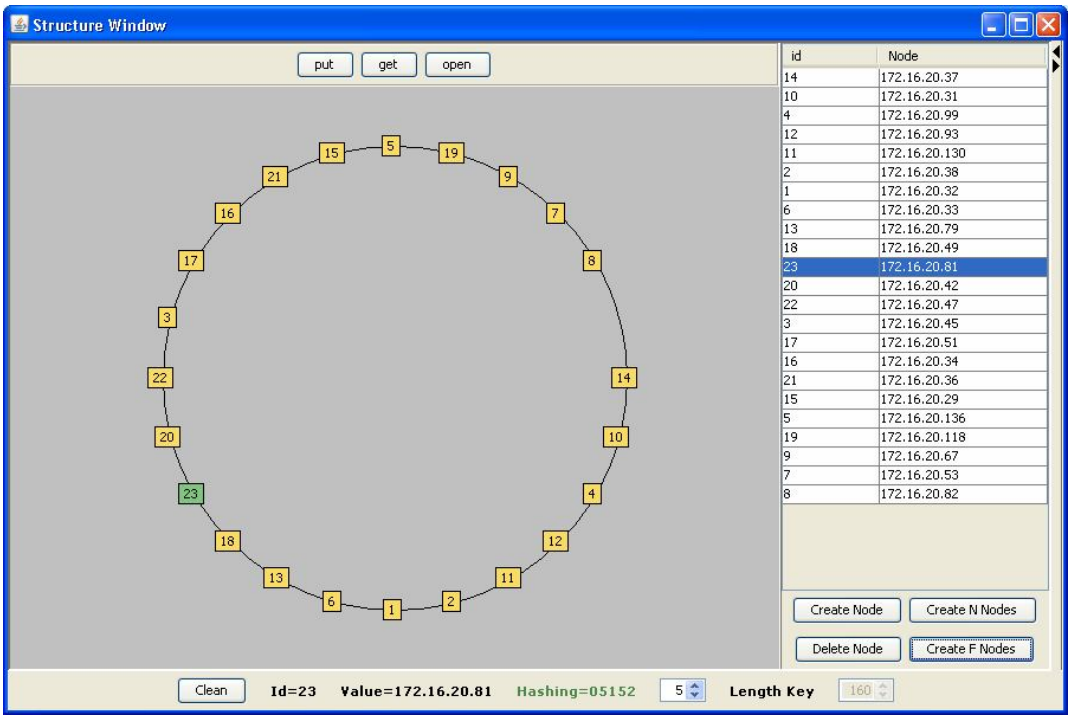


Figure 10: Creates 23 nodes by text file

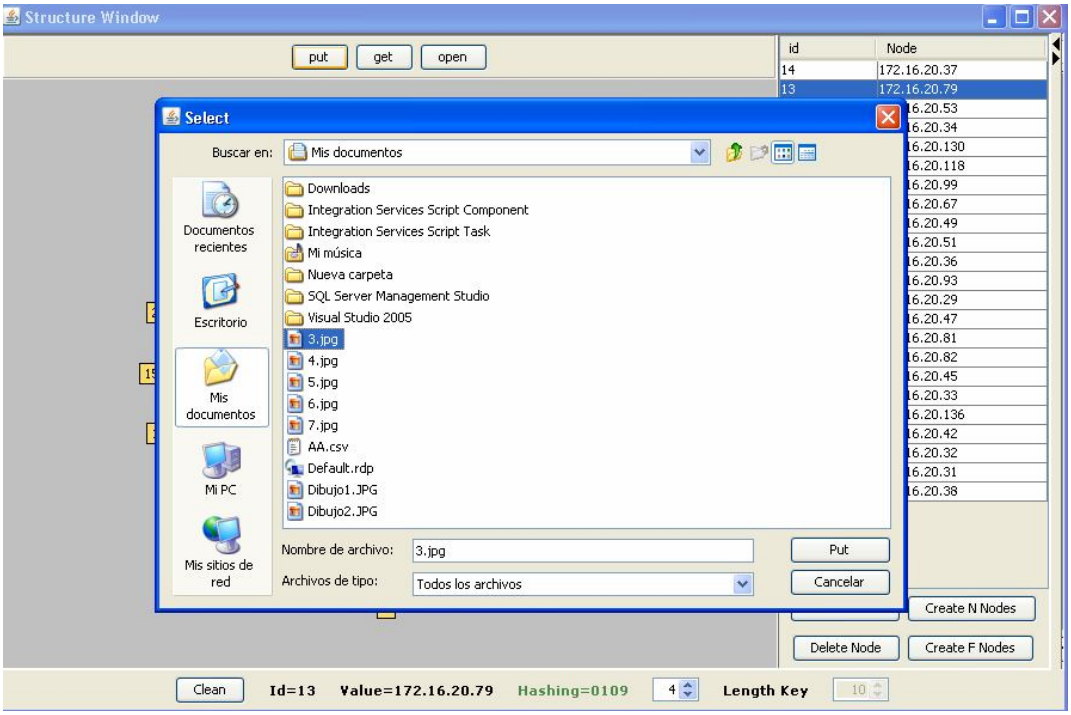
You can select any node in *Ring Panel* Figure 9 (blue color) or in *Nodes Manager* Figure 9 (green color). In *Hashing Configurations* Figure 9 (brown color) show the hashing (green color) of selected node. You can configure how many digits displayed using spinner (next of hashing).

3.1.4. Deleting nodes

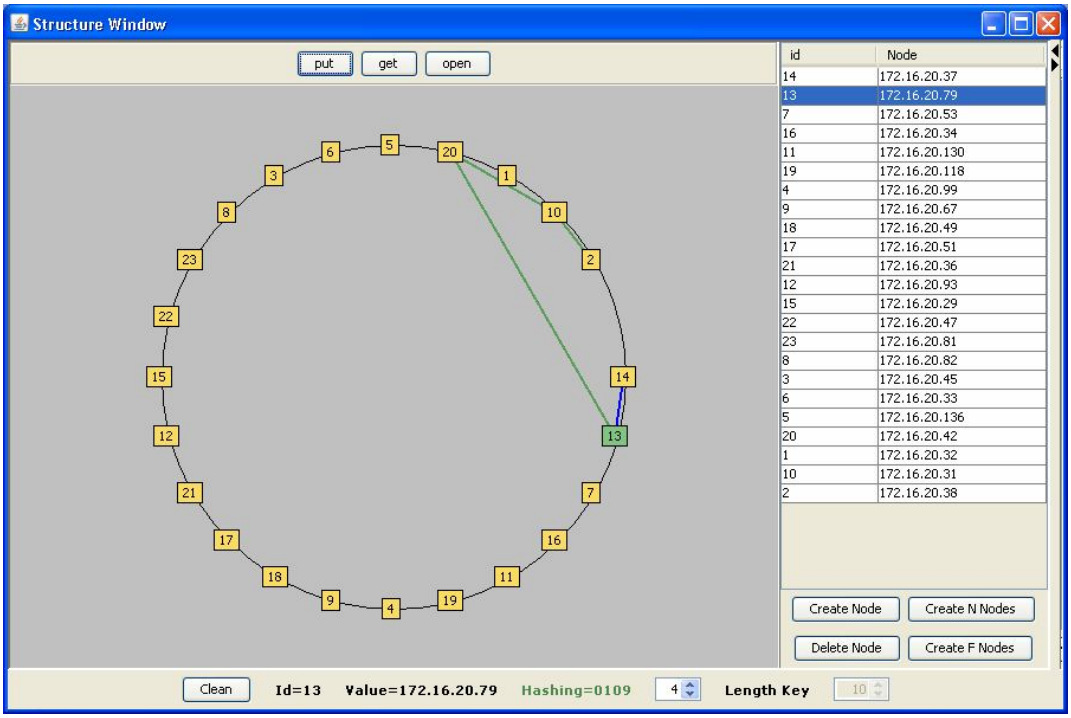
For deleting nodes, you can selected any and click in *Delete Node* in the *Nodes Manager* Figure 9 (green color).

3.1.5. Putting resources

For putting resource, select button Put in the Services Panel Figure 9 (red color). It displays file chooser, select the file to put and select Put. In the Ring Panel is displayed in green color the routing jumps for lookup and blue color the resource transfer (Figure 11).



a) Select file *3.jpg* and click in *Put*.



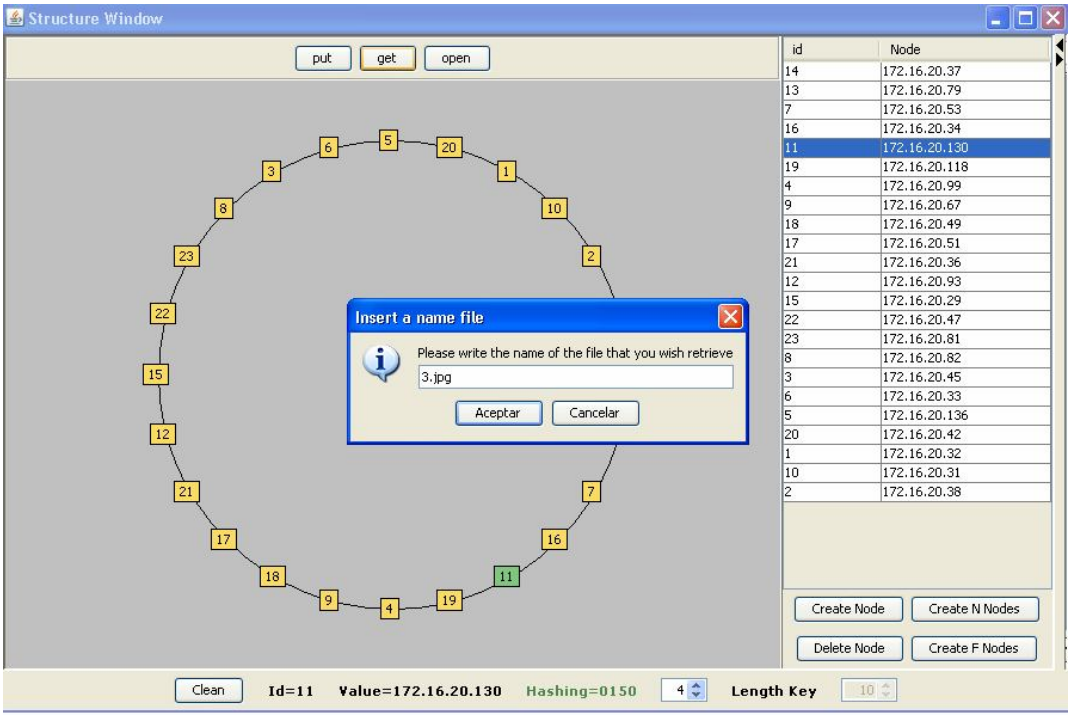
b) Displayed jumps routing for lookup (green) and transfer (blue)

Figure 11: Putting resource

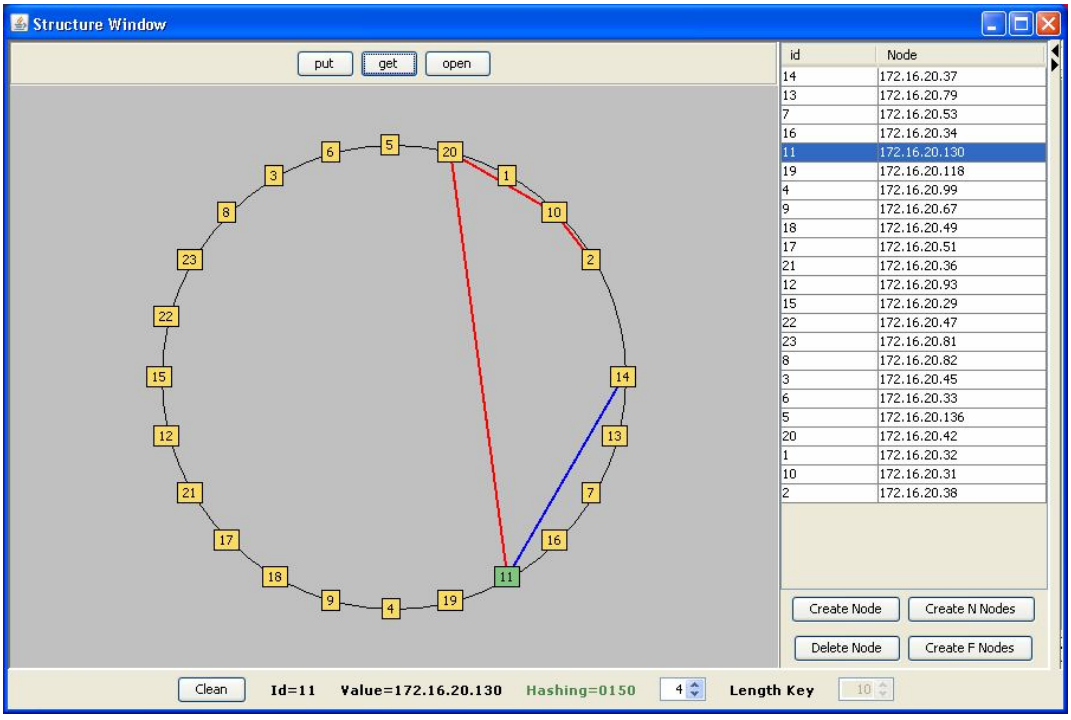
The resources are stored in *dhash/node_name*. In this folder, you must find in root all resources put that will correspond to the node. To see this folder, selects the node and click in open button.

3.1.6. Getting resources

For getting resource, select button Get in the Services Panel Figure 9 (red color). It displays dialog, sets resource name (file name, sensitive case) and select Get. In the Ring Panel is displayed in red color the routing jumps for lookup and blue color the resource transfer (Figure 12).



a) Writes file name (case sensitive) and select *Aceptar*



b) Displayed jumps routing for lookup (red) and transfer (blue)

Figure 12: Getting resource

The resources obtained from get are stored into *dhash/node_name/gets*.

3.1.7. Hashing utilities

To see hashing utilities select the arrow or drag the part referred of color red (Figure 13), must sees like Figure 14.

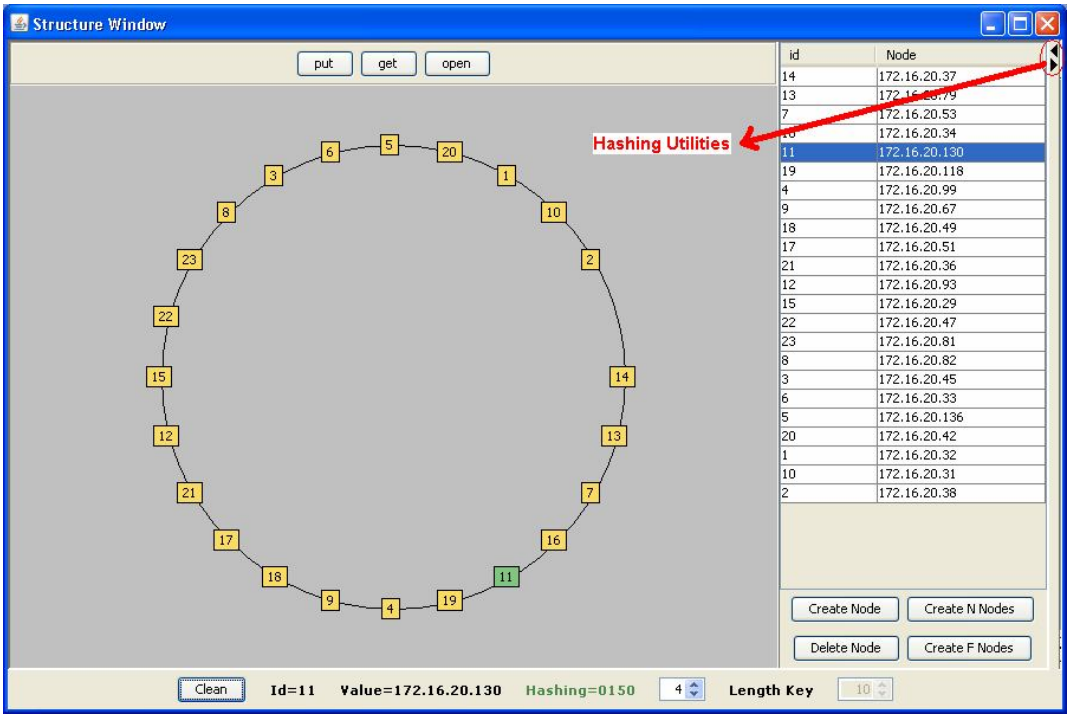


Figure 13: How to see hashing utilities

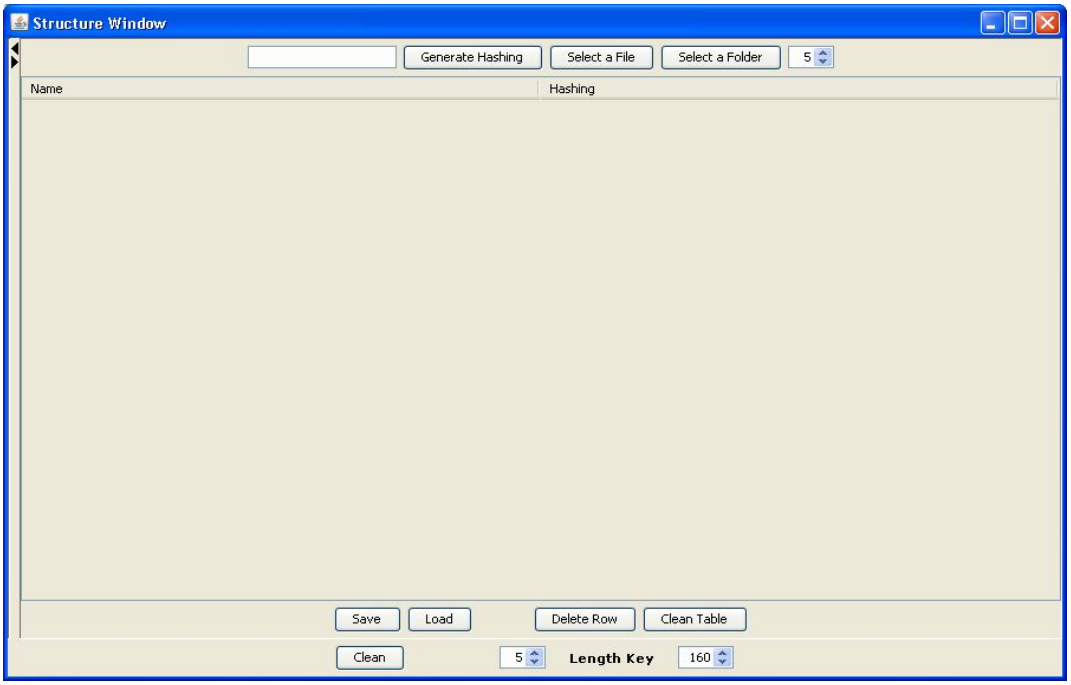


Figure 14: Hashing utilities

You can generate hashing in 3 ways:

- *Generate Hashing*: Generates hashing from a String sets in the input.
- *Select a File*: Generates hashing of file name.
- *Select a Folder*: Generates hashing from a folder. Finds all files into this folder and generates hashing of file name.

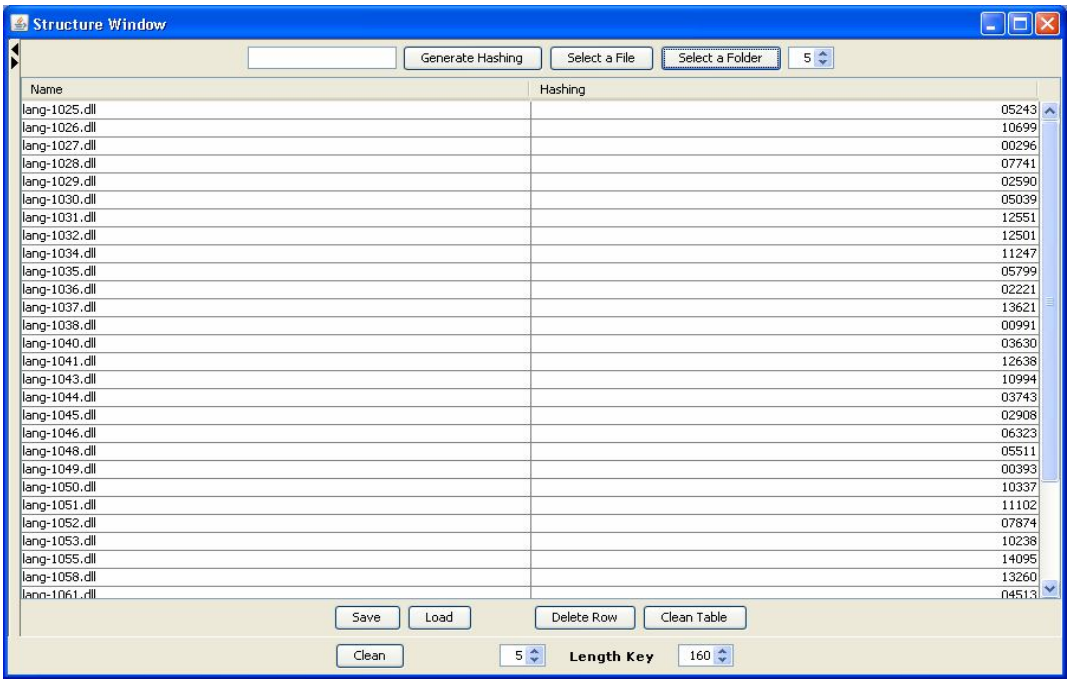


Figure 15: Generated hashing from folder

You can modified length key for generated hashing based in this. Also you can modified the amount of digits displayed with the spinner aside of *Select a Folder* button.

For save the contents in table, selects *Save* button. The format of file generated is:

name1_:_hashing1
name2_:_hashing2
name3_:_hashing3
...
...
nameN_:_hashingN

For example: lang-1025.dll_:_05243.

3.2. Network GUI

The network GUI is like show in Figure 8. This works in the same form like data structure GUI. The only difference is the *exit* button, this button is used for exit the node correctly.

Note: You MUST used the exit button when you wants that the node exit correctly.

Lookup Services API

The *lookupServices.jar* contains all services required for to make lookups in a *peer-to-peer* network. Represents routing layer.

4.1 How to implements

For to implement *Lookup Services* you must to make following:

I. Create a class that implements the interface *OverlayNode*. This class must to implement all required for lookup. (Required)

Example:

```
public class ChordNode implements OverlayNode {
```

II. Create a class that extend from *OverlayNodeFactory*. This class must to implement all required for to create nodes *OverlayNode*. (Required)

Example:

```
public class ChordNodeFactory extends OverlayNodeFactory {
```

III. Verify that the class co.edu.uniquindio.utils.hashing.Key implements the methods appropriate comparison, if not, create a class that extend from *Key* and overriden this methods. (Optional)

IV. Verify that the implementation for hashing (co.edu.uniquindio.utils.hashing.HashingGeneratorImp) is useful. If not, create a class that extend from co.edu.uniquindio.utils.hashing.HashingGenerator. (Optional)

4.2 How to use

For to use Lookup Services observes the following code:

```
12 public class LookupMain {
13     public static void main(String[] args) {
14
15         try {
16
17             // Init hashing generator
18
19             HashingGenerator
20                     .load("co.edu.uniquindio.utils.hashing.HashingGeneratorImp");
21
22             // Instancing factory
23
24             OverlayNodeFactory overlayNodeFactory = OverlayNodeFactory
25                     .getInstance("co.edu.uniquindio.chord.node.ChordNodeFactory");
26
27             // Creating nodes
28
29             OverlayNode node1 = overlayNodeFactory.createNode();
30             OverlayNode node2 = overlayNodeFactory.createNode("MyNode");
31             OverlayNode node3 = overlayNodeFactory.createNode(InetAddress
32                     .getLocalHost());
33
34             // Stabilization time...
35
36             Thread.sleep(5000);
37
38             // Key lookup
39
40             Key key = new Key("image.jpg");
41
42             // Node found after lookup
43
44             Key foundNode;
45
46             foundNode = node1.lookUp(key);
47
48         } catch (OverlayException e) {
49             e.printStackTrace();
50         } catch (UnknownHostException e) {
51             e.printStackTrace();
52         } catch (InterruptedException e) {
53             e.printStackTrace();
54         }
55     }
56 }
```

Figure 16: Code for to use *Lookup Service*

Line 19: Initialized hashing generator. The parameter is the class name of the implementation of *HashingGenerator* (Optional)

Line 24: Create an instance of overlay node factory. The parameter is the class name of the implementation of *OverlayNodeFactory*.

Line 29-31: Create 3 overlay nodes.

Line 36: Waiting for the network stabilizes (Only if the overlay network requires to ensure success in the search) (Optional)

Line 40: Create an instance of Key to search.

Line 46: Lookup of Key. Return the node that is responsible for the key.

Chord Like Lookup Service

Chord is a *peer-to-peer* system described in the paper: A Scalable peer-to-peer Lookup Protocol for Internet Applications. This is an implementation of Lookup Service API and it is reusable for components that have access across Lookup Service API.

5.1 Properties File

Chord module contains two properties file in XML, *chord.xml* and *communication.xml*.

5.1.1 chord.xml

chord.xml is based in the following XSD named *chord.xsd*:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.DHT-UQ.org/chord"
    elementFormDefault="qualified" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.DHT-UQ.org/chord">

    <xsd:element name="chord">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="time" minOccurs="1" maxOccurs="1"></xsd:element>
                <xsd:element ref="successorList" minOccurs="1"
                    maxOccurs="1"></xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="time">
        <xsd:complexType>
            <xsd:attribute name="stableRing" use="optional"
                default="2000">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:long">
                        <xsd:minInclusive value="100"></xsd:minInclusive>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="successorList">
        <xsd:complexType>
            <xsd:attribute name="amount" use="optional" default="3">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:int">
                        <xsd:minInclusive value="1"></xsd:minInclusive>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Figure 17: XSD for *chord.xml*

The XSD (Figure 17) describes all terms on properties.

XML example:

```
<?xml version="1.0" encoding="UTF-8"?>
<chord xmlns="http://www.DHT-UQ.org/chord"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.DHT-UQ.org/chord chord.xsd ">

    <time stableRing="2000" />

    <successorList amount="3" />

</chord>
```

Figure 18: Properties file *chord.xml*

The following describes the properties:

Name	**Description**
time.stableRing	Stabilizing time in miliseconds of *ChordNode*. Time interval for invoke *stabilized*, *fixFingers*, *checkPredecessor* and *fixSuccessors*.
successorList.amount	Size of successor list.

5.1.2 communication.xml

This properties file is described after. It is used for the communication module configuration.

Storage Service API

The *storageServices.jar* contains all services required for to storage resources in a *peer-to-peer* network. Represents storage layer.

6.1 How to implements

For to implement *Storage Services* you must to make following:

I. Create a class that implements the interface *StorageNode*. This class must to implement all required for puts and gets of resources. (Required)

Example:

```
public class DHashNode implements StorageNode {
```

II. Create a class that extend from *StorageNodeFactory*. This class must to implement all required for to create nodes *StorageNode*. (Required)

Example:

```
public class DHashNodeFactory extends StorageNodeFactory {
```

III. Verify that the implementation for digest (co.edu.uniquindio.utils.hashing.DigestGeneratorImp) is useful. If not, create a class that extend from co.edu.uniquindio.utils.hashing.DigestGenerator. (Optional)

IV. Verify that the implementations resources (co.edu.uniquindio.storage.resource.FileResource and co.edu.uniquindio.storage.resource.ObjectResource) is useful. If not, create a class that extend from co.edu.uniquindio.storage.resource.Resource. (Opcional)

6.2 How to use

For to use Lookup Services observes the following code:

```
15 public class StorageMain {
16     public static void main(String[] args) {
17 
18         try {
19 
20             // Init digest generator
21 
22             DigestGenerator
23                     .load("co.edu.uniquindio.utils.hashing.DigestGeneratorImp");
24 
25             // Instancing factory
26 
27             StorageNodeFactory storageNodeFactory = StorageNodeFactory
28                     .getInstance("co.edu.uniquindio.dhash.node.DHashNodeFactory");
29 
30             // Creating nodes
31 
32             StorageNode node1 = storageNodeFactory.createNode();
33             StorageNode node2 = storageNodeFactory.createNode("MyNode");
34             StorageNode node3 = storageNodeFactory.createNode(InetAddress
35                     .getLocalHost());
36 
37             // Stabilization time...
38 
39             Thread.sleep(5000);
40 
41             // Resources to put
42 
43             Resource resourceToPut1=new FileResource(new File("image.jpg"));
44             Resource resourceToPut2=new ObjectResource("object1",new Object());
45 
46             // Making put
47 
48             node2.put(resourceToPut1);
49             node3.put(resourceToPut2);
50 
51             //Making get
52 
53             Resource resourceToGet=node1.get("image.jpg");
54 
55         } catch (UnknownHostException e) {
56             e.printStackTrace();
57         } catch (InterruptedException e) {
58             e.printStackTrace();
59         } catch (StorageException e) {
60             e.printStackTrace();
61         }
62     }
63 }
```

Figure 19: Code for to use *Storage Service*

Line 22: Initialized digest generator. The parameter is the class name of the implementation of *DigestGenerator* (Optional)

Line 27: Create an instance of storage node factory. The parameter is the class name of the implementation of *StorageNodeFactory* (Required)

Line 32-34: Create 3 storage nodes (Required)

Line 39: Waiting for the network stabilizes (Only if the overlay network requires to ensure success in the search) (Optional)

Line 43-44: Create an instance of FileResource and ObjectResource (Optional, can implement another type of resource)

Line 48-49: Putting the resources (Optional)

Line 53: Getting the resource named ‘image.jpg’ (Optional)

DHash Like Storage Service

DHash is an implementation of Storage Service. Managements all onto resources (persist, remove, storage, mapped).

7.1 Properties File

DHash module contains two properties file in XML, *dhash.xml* and *communication.xml*.

7.1.1 dhash.xml

dhash.*xml* is based in the following XSD named *dhash.xsd*:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.DHT-UQ.org/dhash"
    elementFormDefault="qualified" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.DHT-UQ.org/dhash">

    <xsd:element name="dhash">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="overlay" minOccurs="1" maxOccurs="1"></xsd:element>
                <xsd:element ref="replication" minOccurs="1" maxOccurs="1"></xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="replication">
        <xsd:complexType>
            <xsd:attribute name="amount" default="1" use="optional">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:int">
                        <xsd:minInclusive value="1"></xsd:minInclusive>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="overlay">
        <xsd:complexType>
            <xsd:attribute name="factoryClass" type="xsd:string"
                use="required">
            </xsd:attribute>
            <xsd:attribute name="observerClass" type="xsd:string"></xsd:attribute>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Figure 20: XSD from *dhash.xml*

The XSD (Figure 20) describes all terms on properties.

XML example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dhash xmlns="http://www.DHT-UQ.org/dhash"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.DHT-UQ.org/dhash dhash.xsd ">

    <overlay factoryClass="co.edu.uniquindio.chord.node.ChordNodeFactory"
        observerClass="co.edu.uniquindio.dhash.node.ReAssignObserver" />

    <replication amount="1" />

</dhash>
```

Figure 21: Properties file *dhash.xml*

The following describes the properties:

Name	**Description**
overlay.factoryClass	Class name that creates overlay nodes. Have to be a *OverlayNodeFactory*
overlay.observerClass	Class name that is notifies by Overlay layer when is needs. (For Chord layer, this is notified when predecessor changes and *ReAssignObserver* reassigned all resource to new predecessor). This class have to be an Observer
replication.amount	Amount replication nodes. **Note:** The amount replication nodes real is calculate by *min*(replication.amount, successorList.length). The successor list is based in Overlay layer.

7.1.2 communication.xml

This properties file is described after. It is used for the communication module configuration.