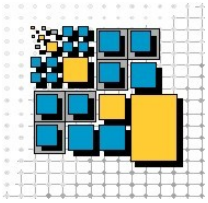

Otto-Friedrich-Universität Bamberg
Distributed and Mobile Systems Group

Open Chord version 1.0.4
User's Manual

Sven Kaffille, sven.kaffille@wiai.uni-bamberg.de
Karsten Loesing, karsten.loesing@wiai.uni-bamberg.de

Lehrstuhl für Praktische Informatik
Otto-Friedrich-Universität Bamberg
Feldkirchenstr. 21
D-96052 Bamberg
GERMANY

22.10.2007

Contents

1 What is Open Chord?

As Peer-to-Peer (P2P) computing becomes more important for distributed applications, which have to be reliable, load balanced and scalable, the underlying technologies must also provide these properties. This is also important for data storage in P2P networks. In recent years new data structures – so-called distributed hash tables (DHT) – have been shown to provide a reliable, load balanced and scalable mechanism to store data in a P2P network.

DHT differ from P2P networks developed before (e.g. Gnutella) in their structural organization. While DHTs are based on structured networks, where for example the nodes in the network are organized in a ring, former P2P networks had no structure, which ordered the nodes of the networks in a particular way. Therefore these networks could not provide guarantees for data stored in the network to be found. The structure of DHTs allows to exactly locate desired data with help of a unique key associated to this data similar to conventional hash tables.

In a DHT every peer of the underlying P2P network takes responsibility for certain data values, that it must store and provide to the other participants of the P2P network. The structure of the P2P network (e.g. ordering of nodes on a ring) is exploited to quickly find and store desired data. Open Chord is an open Java-based[1] implementation of the Chord DHT described by Stoica et al. in [1]. It provides an interface for Java applications to take part as a peer within a DHT and to store and retrieve arbitrary data from this DHT. So Java-based P2P applications can benefit from properties of DHTs. Open Chord is called open, as it is distributed under GNU General Public License (GPL) [2], so that it can be used and extended for own purposes for free and as desired.

In the following section this manual first describes what is required to and how to install Open Chord. For this purpose the compilation process and configuration of Open Chord is explained. The third section explains how the Application Programming Interface (API) of Open Chord can be used by any Java Application. To get a better understanding of what Open Chord is doing the architecture of Open Chord and its interfaces and their usage are described. As Open Chord comes with a command line interface, that can be used to test an instance of an Open Chord DHT or create one DHT consisting of several nodes within one Java Virtual Machine, section four deals with the Open Chord console. Section five describes current limitations of Open Chord.

1.1 Features of Open Chord

Open Chord provides the following features to a Java application:

- Easy to use interfaces for synchronous and asynchronous utilization of a Chord [1] DHT.
- Possibility to store every serializable Java object within the DHT.
- Creation of custom keys to associated data with.
- Transparent maintenance of Chord DHT routing.
- Transparent replication of stored data.
- A remote communication protocol based on Java sockets.
- A local communication protocol for testing and presentation purposes.

[1] Sun Microsystems, "Java 2 platform, standard edition (j2se) 5.0," http://java.sun.com/j2se/1.5.0/download.jsp.
[2] Free Software Foundation, "Gnu general public license," http://www.gnu.org/copyleft/gpl.html

2 Setting up Open Chord

This section describes what is required to use Open Chord and how to compile and configure it.

2.1 Requirements

As Open Chord is Java-based it can be run on any Operating System, for which a Java Virtual Machine for Java 5.0 is available. In order to be compiled Open Chord requires:

- Java 2 Platform Standard Edition Development Kit 5.0,
- the Apache Ant[3] build tool,
- and a library of Apache log4j logging framework, that must be placed in the `lib` directory and can be obtained from the Apache Software Foundation[4]

In order to be executed Open Chord just requires a Java 2 Platform Standard Edition Runtime Environment 5.0. Log4j is only required to compile Open Chord and does not need to be available at runtime.

2.2 Installation

In order to install Open Chord it has to be downloaded from its website[5]. There the sources and any other files required can be found in zip archive. This archive has to be extracted in any desired directory.

The contents of the directory after extraction should contain the following directories and files:

- `bin`: This directory contains scripts to start the Open Chord Console. The instructions in these scripts work only in the directory structure described here.
- `build`: This folder contains the compiled source files after Open Chord has been compiled.
- `config`: Within this folder the configuration file (Java property file) for Open Chord and a sample configuration file for log4j reside.
- `dist`: If a jar file of Open Chord is created during compilation, this directory will contain it. should be created during the compilation process.
- `docs`: This directory contains this manual and the Open Chord API description.
- `lib`: This directory contains third party libraries used by Open Chord. Currently the only required library is log4j.
- `src`: This directory contains the source files of Open Chord.
- `build.xml`: This is the build file used to compile Open Chord with help of the Ant build tool.
- `license.txt`: This file contains a copy of GNU General Public License.

[3] http://ant.apache.org/
[4] http://logging.apache.org/
[5] http://www.lspi.wiai.uni-bamberg.de/dmsg/software/open_chord/

2.3 Compilation

Open Chord can be compiled with help of the Apache Ant build tool, that can be obtained from the Apache Software Foundation for free. For this purpose Open Chord is distributed with an Ant build file (`build.xml`). This build file contains the following targets[6]:

- `clean`: Deletes the `build` and `dist` directories.
- `init`: Creates `build` and `dist` directories if they do not exist.
- `compile`: Compiles Open Chord to the `build` directory.
- `dist`: Compiles Open Chord to the `build` directory and creates a jar file within `dist` directory.
- `documentation`: Creates Javadoc API documentation in the `docs directory`.

To compile Open Chord at the command line of your Operating System change to the directory, where the Open Chord build file (`build.xml`) is located. After Open Chord has been successfully compiled, change to the bin directory and execute the script to start the Open Chord Console for your Operating System (e.g. for Microsoft Windows execute `console.bat`). Type `ant compile` to create the class files for Open Chord. If you want to create a jar file, that can be used as a library for Java applications type `ant dist`. In order to use Open Chord as a library for other applications make sure, that the configuration file for Open Chord is available from the classpath of these applications.

2.4 Configuration

Open Chord can be configured with a Java property file, which is located in the `config` directory as mentioned above. In this section the possible properties that can be set to adjust Open Chord are explained.

The properties to configure can roughly be divided into three categories. Properties to configure logging of Open Chord, properties to configure maintenance and replication[7] within DHT, properties to configure handling of incoming requests.

The properties to configure logging of Open Chord are:

- `de.uniba.wiai.lspi.util.logging.logger.class`:
  This property specifies the fully qualified name (FQN) of the logger implementation that is used to log messages of Open Chord. Currently a Logger, that just logs on the console (`de.uniba.wiai.lspi.util.logging.SystemOutPrintlnLogger`), and a logger that wraps log4j (`de.uniba.wiai.lspi.util.logging.Log4jLogger`) are available. The latter is the standard logger if no logger has been specified. If log4j or a specified custom logger cannot be found on the classpath logging is turned off. So Open Chord can even run if log4j or the specified custom logger are not available.

- `de.uniba.wiai.lspi.util.logging.off`:
  If this property is set to `true`, no logging is performed by Open Chord. This is automatically set to true if the standard logger or the logger specified with help of the property above cannot be found.

- `log4j.properties.file`:
  This property defines the name of the property file, that is used to configure log4j. If the file is

[6] Please refer to the Ant User's Manual for details on how to use Ant. http://ant.apache.org/manual/index.html
[7] For information on how maintenance and replication is conducted in Chord refer to [1]

available from the classpath it can just be the file name. If the file is located in a directory not on the classpath, this must be the full path to the file in a format suitable for your operating system.

- `de.uniba.wiai.lspi.Chord.data.ID.number.of.displayed.bytes`:
  This property defines how many bytes of an identifier of a peer or data item within the DHT should be displayed in logging output.

The properties to configure maintenance and replication of the Open Chord DHT are:

- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.successors`:
  This property must be set to an integer value that represents the number of replicas that are created from a data value.
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.StabilizeTask.start`:
  This property must be set to an integer value, that specifies the number of seconds to wait until the task to stabilize the Open Chord network is started, after Open Chord has been initialized.
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.StabilizeTask.interval`:
  This property specifies (with help of an integer value) the timespan in seconds between successive executions of the task to stabilize the Open Chord network.
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.FixFingerTask.start`:
  This property must be set to an integer value, that specifies the number of seconds to wait until the task to fix the routing table of an Open Chord peer is started, after Open Chord has been initialized.
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.FixFingerTask.interval`:
  This property specifies (with help of an integer value) the timespan in seconds between successive executions of the task to fix the routing table of an Open Chord peer.
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.CheckPredecessorTask.start`:
  This property must be set to an integer value, that specifies the number of seconds to wait until the task to check the predecessor of an Open Chord peer is started, after Open Chord has been initialized.
- `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.CheckPredecessorTask.interval`:
  This property specifies (with help of an integer value) the timespan in seconds between successive executions of the task to check the predecessor of an Open Chord peer.

The properties to configure the number of concurrently serviceable requests from other peers are:

- `de.uniba.wiai.lspi.chord.com.socket.InvocationThread.corepoolsize`:
  This property must be set to an integer value, that specifies the number of threads that are always available to serve incoming requests from other Open Chord peers.
- `de.uniba.wiai.lspi.chord.com.socket.InvocationThread.maxpoolsize`:
  This property must be set to an integer value, which specifies the maximum number of threads that can be available to serve incoming requests from other Open Chord peers.
- `de.uniba.wiai.lspi.chord.com.socket.InvocationThread.keepalivetime`:
  This property defines in seconds how long threads for incoming requests are allowed to be kept alive when they are idle.

2.5 Execution

You may start Open Chord either by starting the Open Chord console (see section 4) or by initializing an Open Chord peer and invoking API methods (see section 3) in your own project. In the latter case you need to make sure that the classpath is set correctly. Apart from the Open Chord classes, you need to include the `/config` directory and maybe a log4j library in the classpath. The `/config` directory contains a property file for Open Chord which is required to run Open Chord. This properties must be loaded before intializing Open Chord or you may also set the properties specified in `chord.properties` within your code with help of `System.setProperty(String, String)`.

3 Using Open Chord

After Open Chord has successfully been compiled and set up, it can be used from any Java application. This section describes the interfaces of Open Chord and how they can be used. For this purpose first a short overview of the architecture and the available communication protocols is provided. Afterward the interfaces relevant to an application programmer and their usage in typical use cases as e.g. creating and joining a network or inserting and retrieving data from the DHT are explained.

3.1 Architecture of Open Chord

The architecture of Open Chord is divided into three layers. These layers are shown in figure 1 below the layer that represents a Java application, that uses Open Chord. On the lowest layer the implementation of the employed communication protocol based on a network communication protocol (e.g. Java Sockets) is located. On top of this communication layer a communication abstraction layer resides, that provides interfaces, which abstract from the actually used communication protocol.

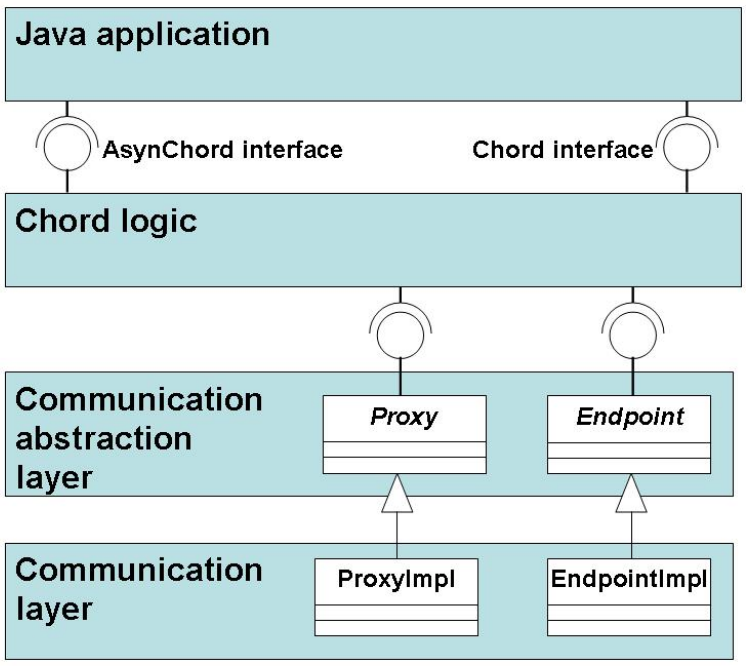

Figure 1: Architecture of Open Chord.

For this purpose two abstract classes have been developed, which represent the communication abstraction layer, and provide factory methods, to create instances of themselves for a specific communication protocol. The communication abstraction layer provides interfaces for synchronous communication between peers. Instances of class represent `de.uniba.wiai.lspi.chord.com.Proxy` (`Proxy`) references to remote peers participating in an Open Chord overlay network. For each node in a Open Chord network an instance of `de.uniba.wiai.lspi.chord.com.Endpoint` (`Endpoint`) provides a connection point for remote peers with help of `Proxy` for a specific communication protocol. The concrete implementations for a communication protocol are determined with help of the URL of a peer[8].

[8]This mapping from communication protocol (with help of URL) to the `Proxy` of `Endpoint` implementation is currently hard-coded within the corresponding factory methods

Based on the communication abstraction layer the logic of the Chord overlay network such as how to find the successor of a peer[9] has been implemented. This layer provides two interfaces to Java applications, which abstract from the implementation of routing within the Chord DHT. One interface `de.uniba.wiai.lspi.chord.service.Chord` (`Chord`) provides synchronous methods to retrieve, store, and remove values within the DHT. The other interface `de.uniba.wiai.lspi.chord.service.AsynChord` (`AsynChord`) can be used for asynchronous retrieval, storage, and removal of data from the DHT. The Chord logic layer also is responsible for replication of data and maintenance of the properties that are necessary to keep the DHT running as described in [1]. This processes are transparent to the application using Open Chord, but must be configured as described in section 2.4.

3.2 Available communication protocols

Currently two communication protocol implementations are available for Open Chord:

1. Local (within one JVM) communication protocol `oclocal`: The implementation of this protocol (with help of sub-classes of `Proxy` and `Endpoint`) can be found in package `de.uniba.wiai.lspi.chord.com.local`. This protocol has been developed in order to facilitate creation of an Open Chord network within a single JVM for testing purposes.
2. Socket-based protocol `ocsocket`: The implementation of this protocol (with help of sub-classes of `Proxy` and `Endpoint`) can be found in package `de.uniba.wiai.lspi.chord.com.socket`. This protocol facilitates reliable communication between Open Chord peers based on TCP/IP sockets.

The protocol, which is used by an Open Chord peer is determined by its `URL`. An implementation of `URL` is provided in package `de.uniba.wiai.lspi.chord.data`. This `URL` currently provides some public constants, which can be used to create Strings representing an appropriate `URL` for one of the protocols described above. The String array `URL.KNOWN_PROTOCOLS` contains the names of the two protocols and the integer constants `URL.LOCAL_PROTOCOL` and `URL.SOCKET_PROTOCOL` represent their corresponding index[10]. So this can be used to create appropriate Strings representing an `URL`, which can be used with the public constructor `URL(String url)`.

3.3 The Chord and AsynChord interfaces

As presented before (figure 1) Open Chord provides two interfaces, which can be used by an application built on-top of Open Chord to retrieve, remove, and store data from/to the underlying DHT. These interfaces (`Chord`, `AsynChord`) provide some common methods that are important to create, join, and leave an Open Chord DHT. These methods are shown in listing 1.

Each peer within an Open Chord DHT is represented by a unique identifier as proposed in [1]. This unique identifier is usually created with help of the `URL` of a peer, when it creates or joins a network or the method, `setURL(URL nodeURL)` is invoked. An identifier of a peer is represented by the class `de.uniba.wiai.lspi.chord.data.ID`. An unique identifier of a peer can also be set by

[9] For details how the Chord DHT works refer to [1].

[10] This should be changed in future releases to allow addition of custom communication protocols without change of implementation of `URL` and factory methods of `Proxy` and `Endpoint`.

the application with help of the method `setID(ID nodeID)`. Setting the `URL` or `ID` of a peer is only allowed **before** an Open Chord network is joined or created.

A new network can be created with help of the methods `create()`, `create(URL localURL)`, and `create(URL localURL, ID localID)`. The first method can only be invoked if an `URL` and an `ID` is set for a peer. An `ID` is automatically generated when an `URL` is set. So when a custom `ID` should be provided, the method `setID(ID nodeID)` must be invoked **after** invocation of `setURL(URL nodeURL)`. The second `create` method is a convenience method to avoid setting of an `URL` before creation of a peer. The third `create` method can be used to create a peer that listens on the provided `URL localURL` and has the `ID localID`.

Listing 1: Common Methods.

```
1  public interface ... {
2    public URL getURL();
3    public void setURL(URL nodeURL)
4      throws IllegalStateException;
5    public ID getID();
6    public void setID(ID nodeID)
7      throws IllegalStateException;
8    public void create()
9      throws ServiceException;
10   public void create(URL localURL)
11     throws ServiceException;
12   public void create(URL localURL, ID localID)
13     throws ServiceException;
14   public void join(URL bootstrapURL)
15     throws ServiceException;
16   public void join(URL localURL, URL bootstrapURL)
17     throws ServiceException;
18   public void join(URL localURL, ID localID, URL bootstrapURL)
19     throws ServiceException;
20   public void leave()
21     throws ServiceException;
22 }
```

The join methods allow a peer to join an existing Open Chord network. These methods correspond to the functionality of the three `create` methods regarding the unique identifier of a peer, but need a further parameter, which is a `URL` of a so-called bootstrap peer, that can be used by the peer, that wants to join, to get access to the existing DHT. How such a `URL` can be found is out of scope of Open Chord and depends on the application, which is built on top of it.

Although Open Chord handles crashes and desertion[11] of peers transparently, it is recommended for a peer to announce to others, when it leaves the Open Chord network. For this purpose the two interfaces provide the method `leave()`. By invocation of this method the peer makes sure that the routing tables of its neighbors immediately reflect the change within the network.

In addition to these methods the `Chord` interface provides the methods shown in listing 2. They can be used to retrieve, insert, and remove data. As data is associated with a unique identifier within a DHT an instance of interface `de.uniba.wiai.lspi.chord.service.Key` (`Key`) must be provided for these methods, with which the data to retrieve, insert, or remove can be associated. Any `Serializable` Java object can be associated with a custom `Key` implementation.

[11] A peer does not crash but leaves the network without notifying other peers

Listing 2: The `Chord` interface.

```
package de.uniba.wiai.lspi.chord.service;
public interface Chord {
  ...
  public Set<Serializable> retrieve(Key key)
      throws ServiceException;
  public void insert(Key key, Serializable entry)
      throws ServiceException;
  public void remove(Key key, Serializable entry)
      throws ServiceException;
}
```

A `Key` implementation must just be able to create a byte representation of itself, which can be obtained by its `getBytes()` method within a byte array. These bytes are used by Open Chord to create a hash value for a given data item with help of a hash function. As more than one instance may be associated with a single `Key` an invocation of `retrieve(Key key)` may return a `Set` of associated instances. For `Key` implementations and classes, whose instances should be stored in the DHT, it is strongly recommended (as with every class used in data structures or the Java Collections API) to overwrite the `equals(Object)` and `hashCode()` method of class `Object` to ensure that Open Chord can correctly manage data storage.

The method `insert(Key key, Serializable entry)` inserts the provided instance `entry` into the DHT associated with the given `key`. This `key` can later on be used to retrieve or remove `entry`. In order to remove an `entry` the `entry` and its `key` must be known, as indicated by the method `remove(Key key, Serializable entry)`.

It is **important** to note that, with the current implementation of Open Chord, class definitions (the byte code of classes), whose instances are used as `Key` or as data associated with a `Key` must be present on every peer within the network as Open Chord does not support class loading from remote peers.

The methods of the `Chord` interface are synchronous and block the thread, that invokes them, until the operation has been performed and a result has been obtained. This may not be desired in all cases, as invocation of the methods described above may take some time. While data is retrieved it may be possible that the same thread, which wants the data, wants to insert, remove or retrieve some other data in parallel. For this purpose Open Chord provides an interface with methods, that allow asynchronous processing of data retrieval, removal, and storage.

These methods can be divided into two kinds. The first kind of methods presented in listing 4 in addition to the provided `Key` or data requires an instance of interface `de.uniba.wiai.lspi.chord.service.ChordCallback`, which is presented in listing 3. The other kind of methods returns instances of `de.uniba.wiai.lspi.chord.service.ChordFuture` (listing 5), which can be used later on to determine if an asynchronous invocation has been completed and to obtain its result (if necessary).

Listing 3: The `ChordCallback` interface.

```
package de.uniba.wiai.lspi.chord.service;

public interface ChordCallback {
  public void retrieved(Key key, Set<Serializable> entries,
     Throwable t);
  public void inserted(Key key, Serializable entry, Throwable t);
  public void removed(Key key, Serializable entry, Throwable t);
}
```

The methods of the first kind require an instance of `ChordCallback`, which is notified, when asynchronously performed retrieval, removal, or storage of data has been completed. These methods return directly after invocation and notification takes place with help of the corresponding callback method `retrieved(...)`, `removed(...)`, or `inserted(...)`. To these method the affected `Key` and `entry` (if required) or result (in case of data retrieval) is provided. If the processing of retrieval, removal, or storage of data has failed a `Throwable` that represents the cause of the failure is provided. This `Throwable` is `null` if invocation completes successfully.

Listing 4: The `AsynChord` interface.

```
1  package de.uniba.wiai.lspi.chord.service;
2  public interface AsynChord {
3    ...
4    public void retrieve(Key key, ChordCallback callback);
5    public void insert(Key key, Serializable entry, ChordCallback
       callback);
6    public void remove(Key key, Serializable entry, ChordCallback
       callback);
7    public ChordRetrievalFuture retrieveAsync(Key key);
8    public ChordFuture insertAsync(Key key, Serializable entry);
9    public ChordFuture removeAsync(Key key, Serializable entry);
10 }
```

The second kind of methods does not require a callback object as a parameter, but instead immediately returns an instance of `ChordFuture`. The methods to insert and remove data both return an instance of `ChordFuture` which can be used to test if the corresponding invocation of `insertAsync(...)` or `removeAsync(...)` has been done with help of the `isDone()` method.

Listing 5: The `ChordFuture` interface.

```
1  package de.uniba.wiai.lspi.chord.service;
2  public interface ChordFuture {
3    public Throwable getThrowable();
4    public boolean isDone() throws ServiceException;
5    public void waitForBeingDone() throws ServiceException,
6        InterruptedException;
7  }
```

The `isDone()` method returns `true` when the invocation has been performed successfully. When the invocation is still being performed `false` is returned. When the invocation has been performed, but has failed, a `ServiceException` is thrown by `isDone()`. If a thread wants to wait until the operation associated with a `ChordFuture` finishes, it can invoke `waitForBeingDone()`, which blocks the calling thread until the end of the operation. This method throws a `ServiceException` if the operation fails and an `InterruptedException` occurs when the thread has been interrupted. When this method returns the associated operation has been performed.

Listing 6: The `ChordRetrievalFuture` interface.

```
1  package de.uniba.wiai.lspi.chord.service;
2  public interface ChordRetrievalFuture extends ChordFuture {
3    public Set<Serializable> getResult() throws ServiceException,
4        InterruptedException;
5  }
```

The method to retrieve data from the DHT returns a `ChordRetrievalFuture` presented in listing 6. This interface extends `ChordFuture` by a method to obtain the result of the retrieval. This method behaves like `waitForBeingDone()` regarding the declared exceptions, but returns the result as soon as the retrieval has been performed to the calling thread. If there is no data associated with the provided `Key` an empty `Set` is returned.

3.4 Examples

3.4.1 Creating a new Chord overlay network

In order to create a new network one of the `create(...)` methods of the `Chord` interface or `AsynChord` interface has to be invoked on an instance of `de.uniba.wiai.lspi.chord.service.impl.ChordImpl`.

Listing 7: Creating an Open Chord network.

```
1  public static void main(String[] args) {
2    de.uniba.wiai.lspi.chord.service.PropertiesLoader.
       loadPropertyFile();
3    String protocol = URL.KNOWN_PROTOCOLS.get(URL.SOCKET_PROTOCOL);
4    URL localURL = null;
5    try {
6      localURL = new URL(protocol + "://localhost:8080/");
7    } catch (MalformedURLException e){
8      throw new RuntimeException(e);
9    }
10   Chord chord = new de.uniba.wiai.lspi.chord.service.impl.ChordImpl
       ();
11   try {
12     chord.create(localURL);
13   } catch (ServiceException e) {
14     throw new RuntimeException("Could not create DHT!", e);
15   }
16   ...
17 }
```

`ChordImpl` implements both interfaces. An instance of it can be created with help of its public constructor. Listing 7 shows an example for creation of a new network. For this purpose a `URL` for the `ocsocket` protocol is created. This `URL` becomes the `URL` of the Open Chord peer. It is recommended to automatically determine the host name and IP-address of a peer with help of `java.net.InetAddress` and to use the hosts IP-address as the host part of the `URL`.

3.4.2 Joining an existing Chord overlay network

Only the first peer of an Open Chord network can create a network. To add peers to an existing network one of the `join(...)` methods of `Chord` or `AsynChord` has to be invoked. Listing 8 shows how an existing network can be joined.

Listing 8: Joining an Open Chord network.

```
1  public static void main(String[] args) {
2    de.uniba.wiai.lspi.chord.service.PropertiesLoader.
       loadPropertyFile();
3    String protocol = URL.KNOWN_PROTOCOLS.get(URL.SOCKET_PROTOCOL);
4    URL localURL = null;
5    try {
6      localURL = new URL(protocol + "://localhost:8181/");
7    } catch (MalformedURLException e){
```

```
8       throw new RuntimeException(e);
9     }
10    URL bootstrapURL = null;
11    try {
12      bootstrapURL = new URL(protocol + "://localhost:8080/");
13    } catch (MalformedURLException e){
14      throw new RuntimeException(e);
15    }
16    Chord chord = new de.uniba.wiai.lspi.chord.service.impl.ChordImpl
        ();
17    try {
18      chord.join(localURL, bootstrapURL);
19    } catch (ServiceException e) {
20      throw new RuntimeException("Could not join DHT!", e);
21    }
22    ...
23  }
```

This works similar to network creation, but additionally a `URL` of a peer, which already is part of the network, that should be joined, is required. This peer is called a bootstrap peer and its `URL` is called a bootstrap `URL`. How this `URL` is obtained depends on the application, that wants to use the DHT. Both `URL`s used in listing 8 must represent an `URL` of the same protocol.

Please note that the `URL` of a peer must reflect the real host name of the host on which the peer is located. In this examples we used `localhost`, so that the examples can only be executed on one machine. It is possible to provide the host name as a parameter to the main method or to determine it with help of the Java API (`java.net.InetAddress`). It is also possible to create more than one chord peer (that uses the `ocsocket` protocol) within one JVM.

3.4.3 Leaving a network

To leave a network the `leave()` method of `Chord` or `AsynChord` has to be invoked.

3.4.4 Inserting and Removing data

In order to work with the DHT at least one implementation of the `Key` interface must be provided. Nevertheless one instance of the DHT can be used with several `Key` implementations. For the following examples a `Key` implementation (listing 9), which uses a `String` as key, is used.

Listing 9: A `Key` implementation.

```
1  public class StringKey implements de.uniba.wiai.lspi.chord.service.
     Key {
2    private String theString;
3
4    public StringKey(String theString){
5      this.theString = theString;
6    }
7    public byte[] getBytes(){
8      return this.theString.getBytes();
9    }
```

```
10   public int hashCode(){
11     return this.theString.hashCode();
12   }
13   public boolean equals(Object o){
14     if (o instanceof StringKey){
15       return ((StringKey) o).theString.equals(this.theString);
16     }
17     return false;
18   }
19 }
```

To synchronously insert data into the DHT a reference to the DHT of type `Chord` is required. On this instance the insert method can be invoked as shown in listing 10. Asynchronous insertion of data can be performed with help of a reference to the DHT of type `AsynChord`.

Listing 10: Inserting data.

```
1 ...
2 Chord chord = ...
3 // create or join
4 ...
5 String data = "Just an example.";
6 StringKey myKey = new StringKey(data);
7 try{
8   chord.insert(myKey, data);
9 } catch(ServiceException e){
10   //handle exception
11   ...
12 }
```

Asynchronous use of the DHT is shown with help of removal of data, which is analogous to asynchronous insertion of data. Two possibilities exist for asynchronous used as described in section 3.3. First removal of data using a `ChordFuture` is shown in listing 11.

Listing 11: Removing data asynchronously.

```
1 ...
2 AsynChord aChord = ...
3 //join or create a DHT
4 ...
5 String data = "Just an Example.";
6 StringKey myKey = new StringKey(data);
7 ChordFuture future = aChord.removeAsync(myKey, data);
8 ...
9 //do other things while operation performed.
10 ...
11 try {
12   boolean finished = future.isDone();
13   while (!finished) {
14     try {
15       future.waitForBeingDone();
16       finished = true;
17     } catch (InterruptedException e){
18       finished = false;
19     }
```

```
20    }
21  } catch (ServiceException e) {
22    //handle exception
23    ...
24  }
25  ...
```

The method `removeAsync(...)` in line 7 of listing 11 returns immediately and provides an instance of `ChordFuture` to the calling thread. This instance (`future`) can later on be used to test if the removal has finished, when it is necessary to know, that the removal has been performed. For this purpose in line 12 it is tested if this is the case. If it is not true, `future.waitForBeingDone()` is invoked to block the thread until the removal has been performed. As the thread may be interrupted while waiting, a while loop (line 13) ensures that the thread really waits until the removal has finished. Insertion of data with this invocation model is performed analogously.

Another possibility to remove or insert data is to use the corresponding method of `AsynChord`, which requires an instance of `ChordCallback` as a parameter. Listings 12 and 13 show an implementation of `ChordCallback` and how an invocation to remove data is made with this model.

Listing 12: A sample callback implementation.

```
1  public class MyCallback implements de.uniba.wiai.lspi.chord.service
       .ChordCallback{
2    public void retrieved(Key key, Set<Serializable> entries,
         Throwable t){
3      ...
4    }
5    public void inserted(Key key, Serializable entry, Throwable t){
6      ...
7    }
8    public void removed(Key key, Serializable entry, Throwable t){
9      if (t == null) {
10       System.out.println("Successfully removed "
11         + entry + " with key "
12         + key);
13     } else {
14       System.err.println("Removal of "
15         + entry + " with key "
16         + key + " failed!");
17       t.printStackTrace();
18     }
19   }
20 }
```

Listing 13: Removing data asynchronously with help of a callback instance.

```
1 ...
2 AsynChord aChord = ...
3 MyCallback callback = new MyCallback();
4 ...
5 String data = "Just an Example.";
6 StringKey myKey = new StringKey(data);
7 aChord.remove(myKey, data, callback);
8 //do other things
9 ...
```

The `ChordCallback` implementation just prints a message to the standard output of the JVM if removal of data is successfully performed. If removal fails, a message is printed to the error output of the JVM and the stack trace of the `Throwable`, that caused the failure, is printed. In order to remove data the `remove(...)` method of `AsynChord` has to be invoked and provided with an instance of the callback. The method returns immediately and the removal is performed asynchronously. As soon as the removal has been performed, the `removed(...)` method of `MyCallback` is invoked and one of the messages described above is being printed. Insertion of data is processed analogously to removal of data.

3.4.5 Data retrieval

In order to retrieve data from the DHT there are also three possibilities as for insertion or removal of data. These work similar to the invocations described in the previous section. Therefore here no code samples are presented and the three possibilities are explained shortly.

1. Synchronous retrieval with `Chord.retrieve(Key key)`: An invocation of this method blocks the calling thread until the retrieval has been performed. A `Set` of the results is returned to the calling thread.

2. Asynchronous retrieval with `AsynChord.retrieve(Key key, ChordCallback callback)`:
   This method returns immediately and the result of the retrieval is provided to the callback instance passed to it through the method `ChordCallback.retrieved(Key key, Set<Serializable> entries, Throwable t)`. The parameter `key` is the key, for which data has been retrieved, `entries` contains the results, and `t` is `null` if the retrieval is successful. When the retrieval has failed, `t` is not `null` and it is an instance of the exception, which has been raised by the retrieval.

3. Asynchronous retrieval with `AsynChord.retrieveAsync(Key key)`:
   This method immediately returns an instance of `ChordRetrievalFuture` which can be used to obtain the result of the retrieval with help of the method `ChordRetrievalFuture.getResult()`. This method works similar to `ChordFuture.waitForBeingDone()`, but returns a `Set` (possibly empty if no data is associated with the key used for retrieval) as soon as the retrieval has finished. If retrieval fails, a `ServiceException` is raised. As the thread may be interrupted, an `InterruptedException` is thrown in that case. For this reason if the thread invoking `getResult()` must wait until retrieval is finished, the invocation of `getResult()` must be guarded by a while loop as described earlier for `ChordFuture.waitForBeingDone()`.

4 The Open Chord Console

For testing purposes Open Chord provides a console, from which a network can be created. For this console a `Key` implementation based on `String` similar to that described in the previous section is being used. Instances of `String` are used as data. There are two possibilities two create/join an Open Chord network. The first one uses the `oclocal` protocol to create a network within one JVM. Therefore many local peers can be created with help of the console. The other possibility is to instantiate a single peer that uses `ocsocket` protocol to create or join a remote Open Chord network.

In order to start the console the following command has to be executed:

```
java -cp OC_DIR/build/classes;OC_DIR/config;OC_DIR/lib/log4j.jar
de.uniba.wiai.lspi.chord.console.Main
```

`OC_DIR` is the directory where the directory structure of Open Chord as described in section 2.2 has been created. If console execution is successful, your screen looks as presented in figure 2. Note that the reference to `log4j` in the classpath (`cp`) is optional as `log4j` is not required to use Open Chord.

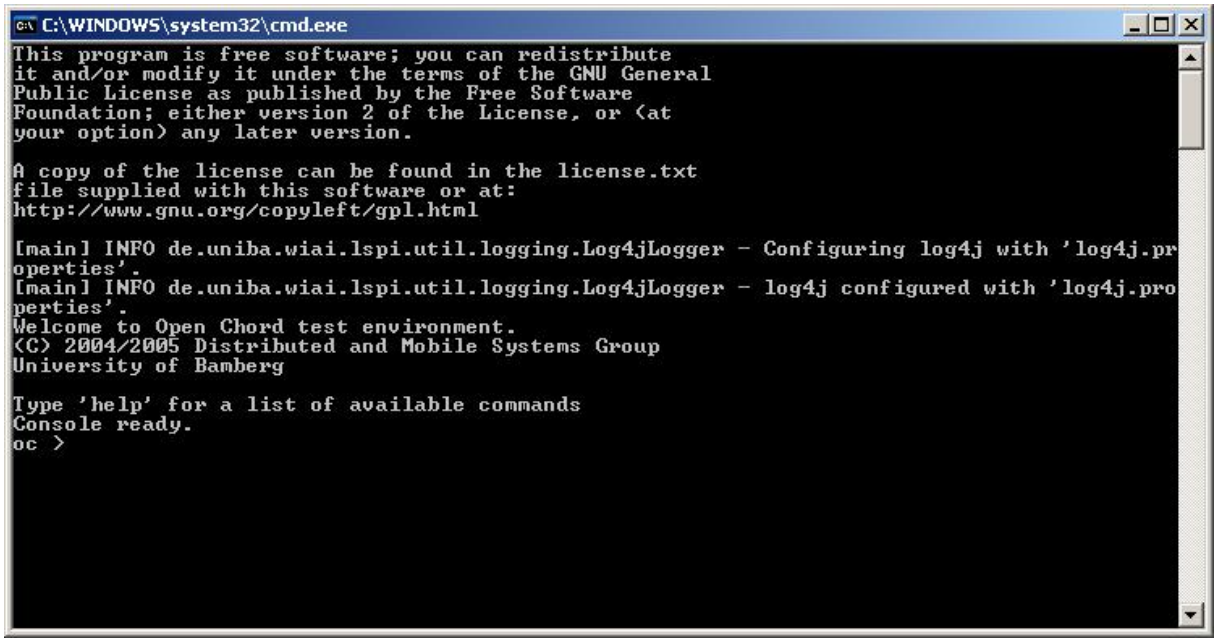

Figure 2: Open Chord console.

When the console is running it waits for commands to be typed in. There are two kinds of commands. The first kind of commands is designed to be used with an Open Chord network that runs within one JVM. The second kind of commands is designed to be used with a single peer, that is connected to other remote peers to create a Open Chord network. The next sections (4.1, 4.2) describe these commands and how they can be used.

There are also some other commands available from the console. To get a list of available commands from the console type `help` at the prompt and press enter. Every listed command also provides help to the user, when it is issued with the parameter `-h` or `-help`. To close the console type `exit`. In order to prevent the console from becoming unreadable if classes use `System.out`, `System.out` is redirected and the output is saved in memory. To display this output type `displaySystemOut`. Output of `System.err` is still printed to the console. Some times it may be useful to repeatedly execute the same commands in the same order. For this purpose a macro can be created. This

macro is a simple text file, which contains a command in each line. This macro file can be loaded and executed with help of the `executeMacro` command.

4.1 Simulating a Chord overlay network in one JVM

With the `oclocal` communication protocol for Open Chord it is possible to create an Open Chord network within one JVM. For this purpose the console provides commands, that can be used to e.g. create a network, insert data, or crash a node. As most of the commands affect only one peer the name of this peer must be provided to these commands. Names of peers have to be unique within the JVM and an `URL` of a peer, that uses the `oclocal` protocol, contains the name of the peer as the host part of the `URL` (e.g. `oclocal://mypeer/`.

These commands are:

- `create`: A new network can be created or an existing network can be joined with this command. To create a new network this command expects one name of a peer provided with help of the `names` parameter (e.g. `create -names mypeer`). If a network exists, this command can be used to create a number of peers at once. The names of the peers must be concatenated with help of '_'. In addition to the names a list of bootstrap peers is required. These list must at least contain one name of an existing peer. If there is more than one bootstrap peer, the names must again be separated by '_'.

  Example:[12]

  ```
  oc > create -names mypeer0
  Creating new chord network.
  oc > create -names mypeer1_mypeer2 -bootstraps mypeer0
  Starting node with name 'mypeer1' with bootstrap node 'mypeer0'
  Starting node with name 'mypeer2' with bootstrap node 'mypeer0'
  oc > create -names mypeer3_mypeer4_mypeer5 -bootstraps mypeer0_mypeer1
  Starting node with name 'mypeer3' with bootstrap node 'mypeer0'
  Starting node with name 'mypeer4' with bootstrap node 'mypeer1'
  Starting node with name 'mypeer5' with bootstrap node 'mypeer1'
  oc >
  ```

  First a new network with a single peer with name `mypeer0` is created. Afterward two peers, that use both `mypeer0` as bootstrap node are simultaneously created. Then three peers are created. The first one uses `mypeer0` as bootstrap peer and the last two use `mypeer1` as bootstrap peer. Bootstrap peers from the list are assigned to the peers at the same position of the list of peers to create. If there are fewer bootstrap peers than peers to create, the last bootstrap peer of the bootstrap peer list is assigned to all peers at the end of the list as a bootstrap peer.

- `insert`: This command can be used to insert a `String` with a certain key into the Chord DHT. This command expects three parameters: `node`, `key`, and `value`. The first one is the name of the peer, from which the insert request is issued. The second parameter is the key, that is used to store the `String`, which is provided with help of the third parameter.

  Example:

[12] If the examples in this section are processed in the same order as in this manual, their output will look similar to the examples!

```
oc > insert -node mypeer1 -key test -value test
Value 'test' with key 'test' inserted successfully from node 'mypeer1'.
oc >
```

- `entries`: This command can be used to print the data and the associated keys stored by all peers in the network. If the name of the peer is provided with help of parameter `node` the entries of this peer are printed to the console.

  Example:

```
oc > entries
Node mypeer5: Entries:
  key = A9 , value = [( key = A9 , value = test)]

Node mypeer0: Entries:

Node mypeer3: Entries:
  key = A9 , value = [( key = A9 , value = test)]

Node mypeer2: Entries:
  key = A9 , value = [( key = A9 , value = test)]

Node mypeer1: Entries:

Node mypeer4: Entries:

oc > entries -node mypeer5
Retrieving node mypeer5
Entries:
  key = A9 , value = [( key = A9 , value = test)]

oc >
```

  In this example it can be noted that data is replicated twice, as the property `de.uniba.wiai.lspi.chord.service.impl.ChordImpl.successors` (see section 2.4) has been set to `2`. Therefore three peers have stored the data inserted in the previous example. The property `de.uniba.wiai.lspi.chord.data.ID.number.of.displayed.bytes` has been set to `1`. For this reason the first byte of the key of data item `test` are displayed as a hexadecimal number.

- `retrieve`: This command can be used to retrieve all data stored for a provided key within the DHT. For this purpose two parameters are required: `node` which determines the peer, from that the retrieval is started, and `key`, that defines the key to lookup.

  Example:

```
oc > retrieve -node mypeer1 -key test
Values associated with key 'test':
test
Values retrieved from node 'mypeer1'
oc >
```

- `remove`: This command removes previously inserted data from the Chord DHT and its parameters are the same as for the insert command.

  Example:

  ```
  oc > remove -node mypeer1 -key test -value test
  Value 'test' with key 'test' removed successfully from node 'mypeer1'.
  oc >
  ```

  After execution of this command, the command `entries` should yield no peer, that holds any data.

- `refs`: This command prints the reference a peer has in its finger table ([1]). It requires one parameter `node`, that defines the name of the node, whose finger table is shown.

  Example:

  ```
  oc > refs -node mypeer3
  Retrieving node mypeer3
  Finger table:
    E5 (0-155)
    19 (156-157)
    21 (158)
    BE (159)

  oc >
  ```

  This command prints the identifier of a peer in the finger table and behind an identifier a range of numbers. This range represents the indexes the node with that identifier has in the finger table.

- `successors`: This command prints out the successors ([1]) of a peer. It requires one parameter `node`, that defines the name of the node, whose finger table is shown.

  Example:

  ```
  oc > successors -node mypeer1
  Successor List:
    19
    21

  oc >
  ```

- `shutdown`: This command can be used to orderly shutdown a number of nodes of the Chord DHT, that then notify their predecessors and successors in the Chord ring, so that they can update their finger tables and successor lists. This command requires one parameter called `names`, that defines a list of names of peers in a similar fashion as the `create` command. This command has an optional parameter `all`, which shuts down all peers.

  Example:

  ```
  oc > shutdown -names mypeer2
  Node with name mypeer2 left.
  oc >
  ```

- `crash`: This command can be used to simulate the crash of a number of peers. These peers do not notify their predecessors and successors. This command requires one parameter called `names`, that defines a list of names of peers in a similar fashion as the `create` command. This command also has an optional parameter `all`, which crashes all peers.

  Example:

  ```
  oc > crash -names mypeer3
  Crashing node mypeer3.
  Node with name mypeer3 crashed.
  oc >
  ```

- `show`: This command displays the peers currently running in the order they are located on the Chord ring. It possesses an optional parameter `count`, that requires no value and counts the number of peers currently running.

  Example:

  ```
  oc > show
  Node list in the order as nodes are located on chord ring:
  Node mypeer0 with id 19
  Node mypeer4 with id 21
  Node mypeer5 with id D9
  Node mypeer1 with id E5
  oc > show -count
  No. of nodes currently running 4
  oc >
  ```

4.2 Connecting to and using a remote network

The console can also be used to create a network with help of the `ocsocket` communication protocol. For this purpose within one console a single Open Chord peer using the `ocsocket` protocol is instantiated. The invocations (e.g. insert) are performed with help of the (synchronous) methods of `Chord` interface.

- `joinN`: This command is used to create a single peer within a JVM. If no parameters are provided the peer creates a new network and listens on the standard port (`4242`) of Open Chord.

  Within one console assumed to run on a machine with IP-address `192.168.0.1` just type `joinN` and hit enter. The following should be the result.

  Example:

  ```
  oc > joinN
  Creating new chord overlay network!
  URL of created chord node ocsocket://192.168.0.1/.
  oc >
  ```

  In another console assumed to run on a machine with IP-address `192.168.0.2` just do the following.

```
oc > joinN -port 8080 -bootstrap 192.168.0.1:4242
Trying to join chord network with boostrap URL ocsocket://192.168.0.1:4242/
URL of created chord node ocsocket://192.168.0.2:8080/.
oc >
```

The `port` parameter can be used to define a port different from the standard port. The `bootstrap` parameter requires the host and the port of the bootstrap peer in the format shown above.

- `insertN`: This command inserts a `String` into the DHT. It requires two parameters: `key` and `value`. Both parameters are `String`s and the first defines the key, with which the `String` defined with help of the second, is inserted into the DHT.

  Example:

  ```
  oc > insertN -key test -value test
  oc >
  ```

- `retrieveN`: This command retrieves all data stored with the key provided with help of `key` parameter.

  Example:

  ```
  oc > retrieveN -key test
  Values associated with key 'test':
  test
  oc >
  ```

- `removeN`: This command requires the same parameters as `insertN` and removes the provided `String` stored with the provided `key`.

  Example:

  ```
  oc > removeN -key test -value test
  Value 'test' with key 'test' removed.
  oc >
  ```

- `refsN`: This command shows the finger table, successor list, and predecessor of the Open Chord peer, which is running in the JVM of the console.

  Example:

  ```
  oc > refsN
  Node: 0C
  Finger table:
    7D (0-158)
  Successor List:
    7D
  Predecessor: 7D
  oc >
  ```

- `entriesN`: This command shows the data stored by the Open Chord peer, which is running in the JVM of the console.

  Example:

  ```
  oc > entriesN
  Entries:
    key = A9 , value = [( key = A9 , value = test)]

  oc >
  ```

- `leaveN`: This command causes the peer to leave the DHT in an orderly fashion, so that it notifies its predecessor and successors about its departure.

  Example:

  ```
  oc > leaveN
  Leaving network.
  oc >
  ```

5 Limitations of current implementation

The current Open Chord implementation does not allow remote **class loading**, which makes it necessary, that all implementations of `Key` and all class definitions of objects, which should be stored within the DHT, must be locally available on each peer. There exist two possibilities to circumvent this problem.

1. A class that is supposed to be used as a `Key` or data within the DHT can be itself saved within the DHT with its fully qualified name as key. This classes can then be loaded by a custom `ClassLoader` that loads classes from the DHT. Details on class loading can be found in [2]. Then just the `Key` implementation for keys of class definitions must be present on each peer.

2. Another possibility is to store objects in their byte representation within the DHT. To convert objects to their byte representation and recreate an object from this representation the classes `ByteArrayOutputStream` and `ByteArrayInputStream` in conjunction with `ObjectOutputStream` and `ObjectInputStream` provided in the `java.io` package can be used. This would only provide the possibility to store objects of arbitrary classes as data within the DHT. For keys there still must be a common `Key` implementation, that is available on each peer.

It is also not possible to easily **exchange the communication protocol** (`ocsocket` or `oclocal`) of Open Chord, as the available protocols are hard-coded into the `URL` class. Based on this the factory methods for the corresponding `Proxy` and `Endpoint` implementations are also hard-coded. Therefore to add a new communication protocol to Open Chord not only classes that implement the protocol must be created, but also the factory methods for `Proxy` and `Endpoint` implementations must be changed and the `Proxy` and `Endpoint` classes have to be recompiled. This should be changed into a mechanism that allows registration of new communication protocols and their implementing classes.

Open Chord currently assumes that every participant of the DHT is **trustworthy**. Therefore every participant can remove arbitrary data. This may be prevented by addition of a security manager, that checks incoming requests. Replication is only guaranteed when peers are trustworthy, as not the peer that inserts data, but the peer responsible for a data item is responsible to replicate this item. Also **data of arbitrary size** can be stored in the DHT, as it is assumed that peers are trustworthy. For the same reason peers are expected to remove data they do not need anymore. Therefor data is stored for an arbitrary time in Open Chord, as there exists no **leasing mechanisms**, that can be used to store data for a certain time.

To get access to the Open Chord DHT the current implementation requires, that an application instantiates a full peer, that provides storage space to the network. Therefor it is not possible to use the DHT without participating in it.

References

[1] 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 the 2001 conference on applications, technologies, architectures, and protocols for computer communications*. ACM Press, 2001, pp. 149–160. [Online]. Available: http://citeseer.ist.psu.edu/442155.html

[2] B. Christudas, "Internals of java class loading," http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html, January 2005. [Online]. Available: http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html