# DCL Integration Tutorial

Martin Exner

April 18, 2015

This article aims at providing a simple tutorial on how to use the Decentralized Communication Layer with third party applications written in Java.

## Introduction

The Decentralized Communication Layer (DCL) is a network of decentralized peer-to-peer networks that can be used to route communication of third party applications through secure and private channels. Each of these networks has a unique network identifier.

At the time of writing, the only network defined for DCL is the circle network with the identifier `org.dclayer.circle`. In this network, each node has an address which is computed by hashing the RSA public key of an RSA keypair the node generates at startup. `SHA-1` is used as hash algorithm, which yields addresses that are 20 bytes in length. Those addresses can then be validated by performing a crypto challenge with the node that should be checked.

Messages in the circle network are routed in a way similar to the Kademlia model. Each node forwards the message to the neighbor with the address that is numerically closest to the message's destination address. In order for this to work, each node needs to be connected to many nodes with addresses that are numerically close to its own address and each node must be connected to the two nodes that have addresses with the shortest possible distance to its own. The amount of connections to nodes with numerically distant addresses does affect the number of hops required for routing messages, but does not influence routing as much.

## Communication

There are two ways application instances can communicate using the DCL. The first is via unreliably transmitted packets, which are routed through a specific DCL network and may or may not arrive at their destination. The second method is via encrypted and connected application channels, which provide reliable transmission of data. For both the initiation of application channels and the transmission of unreliable packets over the circle network, the public key of the remote node is required as destination address.

## Overview

In order to integrate DCL communication features in third party applications, DCL provides a Java library that manages the TCP connection to the DCL service, including creating and accepting application channels and sending and receiving unreliably transmitted messages.

The `org.dclayer` packages contain all required classes.

# Usage

## Connecting to the DCL service

To use DCL in an application, a `Service` object needs to be created first. Below is an example, where `port` is an integer containing the port number the DCL service is listening on.

```
Service service = new Service(
    port);
```

Afterwards, an `ApplicationInstance` object needs to be created for the application to be connected to the service. This works best by utilizing an `ApplicationInstanceBuilder` object, which is returned by `Service.applicationInstance()`.

```
ApplicationInstanceBuilder
    builder = service.
    applicationInstance();
```

This fluent interface can be used to set the keypair to use as the application's address, to join DCL networks and to connect the application to the service. The code below will use the `KeyPair` object referenced by `addrKeyPair` as this application's address, join the default DCL networks, register the object referenced by `listener` as the `NetworkEndpointActionListener` for the default network endpoints and connect the application to the DCL service.

A network endpoint is a pair that consists of an address and a DCL network that address has joined. Unreliably transmitted packets are sent over a DCL network and between two addresses – thus between two network endpoints on the same network. The DCL application to service protocol utilizes network endpoint slots, which are essentially numbers referring to network endpoints, to specify the origin of a message sent from the application and the destination of a message sent to the application. The DCL application library uses `NetworkEndpointSlot` objects to define those network endpoint slots and to refer to the address and network of a network endpoint. The `NetworkEndpointSlot` object is required in every library method that requires the address and network that should be used to be specified and is passed in every callback method defined in the `NetworkEndpointActionListener` interface.

```
ApplicationInstance application
  = builder
    .addressKeyPair(addrKeyPair)
    .joinDefaultNetworks(listener
        )
    .connect();
```

The call to `connect()` will block until the TCP connection to the service is established and the application-to-service protocol is initiated. If an error occurs, a `ConnectionException` will be thrown. Otherwise, `connect()` will return a new `ApplicationInstance` object, which can be used to send unreliably transmitted packets and to initiate application channels.

## Callbacks

### NetworkEndpointActionListener

The `NetworkEndpointActionListener` interface defines methods for receiving callbacks upon joining of DCL networks (`onJoin()`), receipt of unreliably transmitted packets (`onReceive()`) and incoming application channel requests (`onApplicationChannelRequest()`).

### ApplicationChannelActionListener

The `ApplicationChannelActionListener` interface defines methods for receiving callbacks upon successful connection (`onApplicationChannelConnected()`) and disconnection (`onApplicationchannelDisconnected()`) of an application channel.

## Unreliably transmitted packets

### Sending

In order to send unreliably transmitted packets, the `send()` method of an `ApplicationInstance` object needs to be called. The `send()` method takes 3 arguments: a `NetworkEndpointSlot` object describing the address and network to use, a `Data` object containing the destination address and another `Data` object containing the payload to transmit, respectively.

### Receiving

Upon receipt of an unreliably transmitted packet, the `onReceive()` method of the `NetworkEndpointActionListener` assigned to the address and network the packet was received on is called and passed 3 arguments: the `NetworkEndpointSlot` object corresponding to the address and network the packet was received on, a `Data` object containing the source address and another `Data` object containing the payload received, respectively.

Note that the source address might be empty, indicating that the origin chose not to include its own address in the message.

## Application channels

### Initiating

To request an application channel to a specific address, the `requestApplicationChannel()` method of an `ApplicationInstance` object needs to be called and passed 4 arguments: a `NetworkEndpointSlot` object to use as the source of this application channel, a `String` used as an action identifier which the remote will receive in its `onApplicationChannelRequest` callback, a `Key` object containing the public key used as address by the remote that the application channel should be connected to and an `ApplicationChannelActionListener` object that will receive callbacks regarding the application channel.

### Accepting

When an application channel is requested, the `onApplicationChannelRequest()` of the `NetworkEndpointActionListener` object assigned to the network endpoint the application channel request was received on is called and passed 4 arguments: the `NetworkEndpointSlot` object corresponding to the network endpoint the request was received on, a `Key` object containing the public key used as address by the remote that requested the application channel, a `String` object containing the action identifier as specified by the remote and an `LLA` object containg the lower-level address (i.e., the IP address and port) of the remote requesting the application channel.

To accept the application channel request, return an `ApplicationChannelActionListener` object that will receive callbacks regarding the application channel.

To ignore the request, simply return `null`.

### Usage

As soon as the application channel is successfully connected, the `onApplicationChannelConnected()` method of the `ApplicationChannelActionListener` that was either passed when calling the `requestApplicationChannel()` method or returned from the `onApplicationChannelRequest()` callback method will be called. The `onApplicationChannelConnected()` callback will be passed an `ApplicationChannel` object that refers to the established application channel.

The `ApplicationChannel` object has the following methods that can be utilized:

`getOutputStream()`
: Returns a `BufferedOutputStream` object that can be used to write data which will be securely sent through the

interservice connection of the local and the remote service and then made available for the remote to read from its `InputStream` object.

Call `flush()` on the `BufferedOutputStream` object to make sure the data written is sent.

`getInputStream()`
: Returns an `InputStream` object that can be used to read the data which the remote wrote into its `BufferedOutputStream` object obtained via `getOutputStream()`.

`getRemotePublicKey()`
: Returns a `Key` object containing the public key used as address by the remote.

`getActionIdentifier()`
: Returns a `String` object containing the action identifier that was passed to the `requestApplicationChannel()` method by the end that initiated the application channel.

`wasInitiatedLocally()`
: Returns `true` if this application channel was initiated by this end, `false` otherwise.