A Tutorial for GNUnet 0.9.x (Java version)

Florian Dold

April 25, 2012

1 Getting Started

1.1 Installing GNUnet

This tutorial assumes that you have GNUnet $\geq$ 0.9.3 installed on your system. Instructions on how to do this can be found at https://gnunet.org/installation. Make sure that you run `configure` with the option `--enable-javaports`. Start GNUnet with the command `gnunet-arm -s` and convince yourself that the default GNUnet services are running by typing `gnunet-arm -I`.

1.2 Installing gnunet-java

Check out the latest version of gnunet-java with subversion:

```
$ svn checkout https://gnunet.org/svn/gnunet-java/
```

Tools for gnunet-java development are located in `tools/`, while the `bin/` directory contains shell wrappers for java programs interfacing with GNUnet. Now run the command

```
$ ./tools/build
```

to compile `gnunet-java`.

To test whether your gnunet-java installation is working, try to run the "gnunet-nse" program (in `bin`, which should display the current estimated size of the network).

Throughout the tutorial it will be useful to consult the javadocs for GNUnet-java, either built them yourself with

```
$ ./tools/make-javadoc
```

or use the online version at https://gnunet.org/javadoc/

2 Creating an extension

3 A simple gnunet-java program

Check out the template directory for gnunet-java extensions with the following command:

```
$ svn checkout https://gnunet.org/svn/gnunet-java-ext/
```

Now edit `envcfg`. This file contains the necessary information so that scripts in the `gnunet-java-ext/tools directory`, as well as the shell-wrappers in `gnunet-java-ext/bin` can find your gnunet-java installation. Note that the template directory already contains an executable example extension that will print a "hello world" message, in this section you will learn how to write your own.

3.1 The Basics

```
public class HelloGnunet {
    public static void main(String[] args) {
            new Program(args) {
                public void run() {
                    System.out.println("Hello, GNUnet");
                }
            }.start();
}
```

Calling `start` initializes gnunet-java, parses the command line, loads configuration files and starts the task scheduler, with the code in the `run` method as initial task.

Exercise: Try to get the code above running. Place your code in the `src/` directory (so that you can use the build script in `tools`, copy and modify the example shell-wrapper `bin/gnunet-ext` until you can run your own program with it.

3.2 Adding and using command line arguments

Command line options are added by annotating members of your `org.gnunet.util.Program` subclass with the Option-annotation.

Here is a simple example:

```
new Program(args) {
    @Option(
        shortname = "n",
        longname = "name",
        action = OptionAction.STORE_STRING,
        description = "the name of the person you want to greet")
    String name;
[...]
}
```

You can now specify value for the member `name` at the command line, either by the long name with two dashes (`--name=Foo` / `--name FOO`) or the short name (`-n Foo`) with one dash.

Inside of the `run` method, the field `name` will then be initialized with the passed argument, or `null` if the option has not been passed.

The Option annotation can not only be used with Strings, but also with booleans and numbers. These are a few of the available options:

- `STORE_STRING`: Store a string in a `String` variable
- `STORE_NUMBER`: Store a number in a member of primitive type
- `SET`: Set a `boolean` to `true`

By default, the following arguments are available on the command line:

- `-h` / `--help` shows the help text
- `-v` / `--version` shows version information
- `-c` / `--config` specify an additional configuration file to load
- `-L` / `--log` specify the log level
- `-l` / `--logfile` specify a file to write the logs to

You can change the about text and the version information by overriding the `getVersion` or `getAboutTest` methods in your `Program` subclass.

Exercise: Add a few different command line options to your program and print them to `System.out`!

4 The statistics API

In this section we will use the statistics API of gnunet-java. This service allows us to store numbers under a subsystem and a name, which are still available to you and other components of your peer after your program exits.

4.1 Establishing a connection with the statistics service

```
Statistics statistics = new Statistics(getConfiguration());
```

The Statistics constructor is called with the default configuration, provided by the method `getConfiguration` of the `Program` class. Calling the constructor establishes a connection to the statistics service. As with most API calls in gnunet-java, this operation is asynchronous. This is one of the main reasons why you have to wrap your program in the overridden `run` method of `Program`: Once all your asynchronous calls are made, the run method returns, and gnunet-java keeps the system running until all work has been done.

Always remember that you always explicitly have to destroy your `Statistics` instance with the `destroy(boolean sync)` method. Otherwise there might be pending operations that prevent the termination of your program. The parameter of `destroy` determines whether pending set-requests to the statistics service should be satisfied or dropped.[1]

4.2 Setting statistics

You can use the newly created `statistics` handle like this to set a value:

```
statistics.set("gnunet-java-hello", "the answer", 42);
```

4.3 Retrieving statistics

Retrieving a value is a little bit more complex. Because of the asynchronous nature of the gnunet-java APIs, the `startGet` method does not directly return values, but a handle (implementing the interface `Cancelable` to cancel the get request. The actual values are accessed by passing a callback object to the `startGet` method.

Example:

[1]This argument is about to be removed in the C version and will likely be removed soon. So don't worry about it, just passing `true` will do.

```
// the name parameter is the empty string, this gets all options of the specified subsystem
Cancelable getCancel = statistics.get(RelativeTime.SECOND, "gnunet-java-hello", "",
new Statistics.StatisticsReceiver {
    public void onDone() {
        System.out.println("everything␣done");
    }
    public void onReceive(String subsystem, String name, long val) {
        System.out.println(subsyste + "␣" + name + "␣" + val);
    }
    public void onTimeout() {
        System.out.println("timeout␣occured");
    }
});
```

Exercise: Write a program that sets statistics values, and check the result with the `gnunet-statistics` command line tool.

Exercise: Write a program to read and print statistics values.

5 The core API

The core API allows sending messages to other connected peers.

5.1 Defining new Messages

All GNUnet messages follow a common communication protocol. Every message consists of a header (with the message size and the message type) and a body.

You can define a new type of nessage in gnunet-java by annotating a class with how to represent its members in binary format.

Additionaly, you have to register your new message type with gnunet-java, giving it a unique id. Here is an example:

```
@UnionCase(4242)
public class ExampleMessage implements GnunetMessage.Body {
    @UInt8
    public int age;
    @ZeroTerminatedString;
    public String name;
}
```

The `@UnionCase` annotation specifies the message id of the message body below (4242 in the example). GnunetMessage.Body is a union of messages, and ExampleMessage is one (new) member of the union.

Every time you add a new type of GNUnet message, you have to run the `tools/update-msgtypes` command. This generates the file `src/org/gnunet/construct/MsgMap.txt`, which allows the system to load the right java class when reading a message from the network.

The above message then contains a value annotated with `@UInt8`: An **8**-bit **U**nsigned **int**eger. There are similar annotations for integers of other sizes, and `@Int`N annotations for signed integers. The second member is a String, whose binary representation appends a zero-byte to the string to mark its end.

Other useful annotations can be found in the package `org.gnunet.construct`. Among them are annotations for arrays of fixed or variable size (`@VariableSizeArray, @FixedSizeArray`), for embeding other messages in your message (`@NestedMessage` and for implementing your own message unions.

Exercise: Define a message that contains a 32-bit signed integer.

5.2 Connecting to Core

After creating a handle to core by calling the `Core` constructor, you have to specify what types of messages you are interested in. The core service will only send messages of these types to you, and only notify you of connecting peers if they share a subset of the messages you are interested in.

The `handleMessages` method allows you to specify an object of a class inheriting `Runabout`. The Runabout is a mechanism for single-argument multiple dispatch in Java. You have to define one `visit` method for every type of message you are interested in. Once `Core` receives a message, it is dispatched dynamically to the `visit` method with the appropriate signature. Note that every `visit` method, as well as the receiver's class, has to be public in order for the dynamic dispatch to work.

Example:

```
public class MyMessageReceiver extends Runabout {
    public visit(MyFooMessage m) { /* do something */ }
    public visit(MyBarMessage m  { /* do something else */ }
}
```

After specifing your message handler, the `init` method has to be called with a callback object. This starts the handshake with the core service, once done the callback object's `onInit` method will be called with your peer's identity.

5.3 Sending a message to another peer

Before you can actually send a message, you have to wait until the core service is ready to send your message. This is done by calling the `notifyTransmitReady` method. You have to provide a callback object to this method, whose `transmit` method is invoked with a `MessageSink` object once the core is ready to transmit your message. Call the `send` method in the `MessageSink` in order to finally transmit it.

Example:

```
// arguments: messagePriority, timeout, targetPeer, messageSize, transmitter
core.notifyTransmitReady(0, RelativeTime.FOREVER, myIdentity, 42, new MessageTransmitter() {
    public transmit(Connection.MessageSink sink) {
        sink.transmit(myMessage);
    }
    public onError() {
        // do something
    }
```

You can use `Construct.getSize` to calculate the size of a message, or just do it manually.

Exercise: Write an echo program for core: Send a message to the local peer and receive it!

6 Other useful APIs

Many of GNUnet's services are not yet available as a gnunet-java API.

The other two service APIs currently implemented are nse (in `org.gnunet.java.nse.NetworkSizeEstimation`), a service that gives an estimation of the current size of the network, and DHT (in `org.gnunet.java.dht.DistributedHashTable`), a service that allows key/value pairs to be stored distributed across the network.

7 Writing your own client/server

GNUnet is split up into components, every component runs in its own process. In the previous sections you have used existing APIs to interface with other services written in C. gnunet-java also provides the tools necessary to directly interface with services yourself. The `org.gnunet.util.Client` class allows to connect to a GNUnet service and exchange messages with the service.

At the time of writing of this tutorial, the server/service API is not yet fully implemented, so writing new services in Java is not yet "easy" in 0.9.3. However, you can write daemons and user interfaces using the `Program` class.

Exercise: Write a `Service` and a `Program` with a client that communicates with it.

Exercise: Write an API for a GNUnet service that has not been implemented yet in gnunet-java and contribute it back to the project.

8 The state of gnunet-java

The `gnunet-java` project is under heavy development, expect changes that break your stuff! Please report any bugs or feature requests at https://gnunet.org/bugs/