TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

JavaCCL Manual

This document presents basics of library setup and usage.

1 Library setup

First you need to include the JavaCCL library in your project. Best way is to create some kind of folder inside your project faolder and store the "JavaCCL.jar" file inside it. Then tell the project to include the file (in NetBeans it is done in Project Properties - Libraries - Add JAR/Folder). After that, you are ready to use the libraries resources in your project.
There are two ways to create an instance of server (or client). First is without explicit specification of port. The library tries to create an instance of server (client) at default port (for JavaCCL it is 5252) and if it fails, it tries to create the instance at next port. If the creation fails even at next port, the process repeats for another port and so on until the object is created or the port number reaches 65535. Below you can find the example of server instance creation the first way.

columns
```
Server s = Server.initNewServer();      // init new server
   on default port (5252)
Client c = Client.initNewClient();      // init new client
   on default port (5252)
```

The second way is to specify the exact port you want to be used. This way you have full control of port, on which the instance will operate, but you have to manually take care of `IOException` when the port is not free.

columns
```
Server s = Server.initNewServer(5252);  // init new server
   on port 5252
Client c = Client.initNewClient(5253);  // init new client
   on port 5253
```

In most cases, creating instances of server and clients on devices will be enough to connect them (because of built-in client discovery), but in some cases where server or clients are separated by router or some other device preventing broadcast separation, you will need to manually connect clients to server. First you need to find out the public IP of the publicly accessible device (you can do this using many websites, eg. `http://www.whatismyip.com/`). Then you will need to call the registration method (as seen in the example below). If the publicly accessible device is server, you will need to instruct every client to register (first line). If the server is hidden and clients are public, then you will need to register every client to server (second line).

columns
```
c.registerToServer(44.33.22.11); // register a client to
   server, server is running at IP 44.33.22.11 on default
   port (5252)
s.registerClient(11.22.33.44); // register a client on
   server, client is running at IP 44.33.22.11 on default
   port (5252)
```

It is also possible to specify port on which is the client / server running.

columns
```
c.registerToServer(44.33.22.11, 5252);  // register client
   to server, server is running at given IP and port 5252
s.getClientManager().registerClient(11.22.33.44, 5253); //
   register client on server, the client is running on port
   5253
```

For proper program termination, you can call *stopService()* method to stop the instance. For server this will deregister all clients and terminate all running threads, for client it will tell the server that the client is shutting down and then stop the client thread.

columns
```
c.stopService();          // terminate server instance
s.stopService();          // terminate client instance
```

2 PCs behind router / NAT etc.

For easy deployment you can configure clients (server) using a simple XML file. You can specify IPs and ports (if needed) for server or clients. You can combine settings for client and server in one file, because the library looks for proper nodes according to which object is created (server or client). You can even include settings in larger XML file, the library will search only for node called "JavaCCL" and read client and server settings from there. Below you can find sample configuration file.

columns
```
<JavaCCL>
  <server>127.0.0.1-5252</server> <!-- Server is running on
     local host on port 5252 -->
  <client>127.0.0.1-5253</client> <!-- Client one is running
      on local host on port 5253 -->
  <client>147.230.184.143-5252</client> <!-- Client two is
     running somewhere else on port 5252-->
</JavaCCL>
```

Depending on your configuration, you will choose to configure clients using a list on server, or you give each client servers IP. Far easier is to give each client server's IP, because this way you can deploy clients without any further configuration. But in some network configurations, you will have to to choose the list of clients on server. This happens when the server does not have a public IP, which can be used to access him. In this case, you will need to determine public IPs of clients and give them to server so he can contact them. Don't forget that at least one side (server or all clients) have to be publicly available (the IP does not have to be public in scope of whole internet, but there must be a IP and port, which can be used to contact the device directly), otherwise the library won't be able to make connection.

3 Communication

Messages are sent using interface `Communicator`, which is obtained after the client (or server) registration. Then you can use method *Object sendData(Object)* to send data and receive a reply.

columns
```
Communicator comm = server.registerClient(11.22.33.44); //
   register new client
comm.sendData("testData"); // send a String "testData" to
   client
```

To receive a message, there are multiple ways to achieve it. If you want only to receive messages and not reply to them, you can register a `java.util.Observer`. All listening registrations are done using interface `ListenerRegistrator`, which is available both on server and client.

columns
```
Client c = Client.initNewClient();       // init new client
c.getListenerRegistrator().addMessageObserver(Observer); //
   register new observer
```

This way, you will receive all messages delivered to this client along with UUID of sender. The data will be packed inside instance of JavaCCL class called `DataPacket`. In case you want to directly reply to a message, you need to go a little higher (in terms of library architecture) and use ID listeners. The process of registration is the same, only you need to implement a `Listener<Identifiable>` or `Listener<DataPacket>` for message (or client) ID listening. For client message listening, you need to provide UUID of the client, which can be obtained using client's communicator. For message listening, you will provide an ID, which can be any Object. But keep in mind, that IDs are compared using *equals()*, so the ID must implement this method according to your requests. And sent messages need to implement `Identifiable` interface, which will allow the library to determine what is the ID.

columns
```
Client c = Client.initNewClient();       // init new client
c.getListenerRegistrator().setClientListener(UUID,Listener<
   DataPacket>);
c.getListenerRegistrator().setIdListener(Object,Listener<
   Identifiable>);
```

You may have noticed from the names of registration methods, that you can have multiple observers for one instance, but only one listener for each client or ID. Each registration of listeners replaces the old listener for the new one.

4 Job Management

In this part we will present the Job manager and all of its parts available to user. The example will search for highest value of fitness function. Each job will have to compute fitness function for a range of numbers and return the biggest one. The source is not complete, some parts will be pseudo-code (in orange), some actual code with comments, type-checking is left out etc. So use this code only as an idea how to do it, not as a copy-paste opportunity.

Code for the server part

columns
```
double start = 0.0, end = 100.0; // define range of values
double step = 0.01;
- divide the range in small pieces and create a Set<double
   []>, where double[] defines start and end of each sub-
   range and step size
for (double[] d : subranges) {  // submit all jobs
        s.getJobManager().submitJob(d);
}
s.getJobManager().waitForAllJobs();     // wait until all
   jobs are complete
- use s.getJobManager().getAllJobs() to get all submitted
   jobs and find the best result
```

Code for the client part

columns
```
public class Finder implements AssignmentListener {
        @Override
    public void receiveTask(Assignment a) {
        double[] range = a.getTask();
        double val;
        double max = Double.MIN_VALUE;
        double maxVal = Double.MIN_VALUE;
        for (double d = range[0], d <= range[1], d += range
           [2]) {
                val = evaluateFitness(d);
                if (val > maxVal) {
                        maxVal = val;
                        max = d;
                }
        }

        a.submitResult(new double[] {max, maxVal});
    }
    @Override
    public void cancelTask(Assignment a) {
        // we don't need this
    }
}

client.setAssignmentListener(new Finder());      // call this
    after the client is created
```

Those 2 pieces of pseudo-code is all you need, the library will take care of the communication, work distribution etc. Sometimes it might be good to distribute data required for computation not along the client, but after the client requests it. JavaCCL offers `DataStorage` which offers this kind of functionality. The data identifier can be any object you like, it is up to you how you handle the data storage implementation. The code for server and client follows.

Code for the server part

columns

```
server.assignDataStorage(new DataStorage() {

        @Override
        public Object requestData(Object o) {
                switch (o.toString()) {
                        case "data1":
                                return "object␣identified␣as
                                   ␣data1";
                        case "data2":
                                return "object␣identified␣as
                                   ␣data2";
                        default:
                                return "Illegal␣data␣request
                                   ";
                }
        }
});
```

Code for the client part

columns

```
public class Finder implements AssignmentListener {

        @Override
    public void receiveTask(Assignment a) {
        // open assignment and determine data that client
           needs
        Object data1 = a.requestData("dataIdentifier1");
        Object data1 = a.requestData("dataIdentifier2");
        // continue computation
    }
        ... rest of the class
}
```