Task 1 - Java RMI

The purpose of the task is to get basic understanding of distributed objects and an idea how accessing them and working with them can affect performance of the application.

Prerequisites

The task is based on computing the distance between two nodes in a graph, simulating work with dynamic data structures. No other special knowledge is required.

Java RMI is the technology used for the remote procedure invocation. For implementing the assignment, understanding of the following is required:

- Implementation of a remotely accessible object (class java.rmi.server.UnicastRemoteObject, inheriting from this class, method exportObject)
- Connecting the client and server using the RMI registry (class java.rmi.Naming, the rmiregistry application)

A simple example of a remote function call for Java RMI is available. A local implementation of the task, which is meant to serves a base to the solution is provided. Download the example and local implementation from: [http://d3s.mff.cuni.cz/teaching/middleware/files/as1.zip](http://d3s.mff.cuni.cz/teaching/middleware/files/as1.zip)

Assignment details

The core of the assignment is a simple distance measuring between two nodes in a graph. The node objects implement the interface Node:

```java
public interface Node {
    Set<Node> getNeighbors();
    void addNeighbor(Node neighbor);
}
```

The `addNeighbor(Node)` method adds the argument node to the set of neighbors of the receiver node object. This method is used when generating a graph. The `getNeighbors()` method returns a set of all neighbors of the receiver node and is used by the distance computation algorithm. The actual distance computation is done through a `Searcher` interface:

```java
public interface Searcher {
    public static final int DISTANCE_INFINITE = -1;
    public int getDistance(Node from, Node to);
}
```

The `getDistance(Node, Node)` method computes and returns the distance between nodes `from` and `to`. If `to` is not reachable from `from`, it returns `DISTANCE_INFINITE`.

Your tasks are:

1. Explore the provided implementation of the task that works locally only.
Measure the speed of execution on several types of randomly generated graphs, both sparse and dense ones.

2. Create a server that would provide a remotely accessible object with the Searcher interface. Update the provided implementation to search using the remote searcher object through the Searcher remote interface, along the local searcher. The server object with the Searcher interface should accept objects with the Node interface, which are not remotely accessible.

Measure the speed of the implemented variants. Question: How does the server Searcher object access the Node objects?

3. Update the server to provide remotely accessible objects with the Node interface that would be created upon client request. Update the provided implementation to allow computation of distance in the graph using a local Searcher working with server Node objects along the existing functionality.

Measure the speed again. Do not forget that it is necessary to measure the same graphs (local and remote ones) to get a relevant comparison. Question: How does the local Searcher object access the server Node objects?

4. Add another variant: compute the distance using the remote Searcher object on the server, to which you pass (from the client) the remote Node objects from the server graph.

Measure the speed again. Question: How does the server Searcher access the server Node objects?

5. Compare the speed of the four variants when both client and server are running on the same machine. Plot the results into a chart with four data series corresponding to the four variants.

Do the same for a situation when client and server are on different physical computers connected by a network. Test in a higher latency environment, e.g. between networks in the campus and in the local lab.

Compare the results of the two situations. (To measure the same graphs, use a fixed random number generator seed.)

6. Results of the previous measurements help to distinguish when it is faster to pass dynamic data structures by value and when it is faster to pass them by reference. The assignment demonstrates this on extreme all-or-nothing cases when either everything is passed by value or everything is passed by reference. But often a combination of both is the right choice.

In the provided implementation of the Searcher interface, there is another method for computing the distance, getTransitiveDistance(int, Node, Node). This method in each step retrieves not only direct neighbors of a node, but a whole set of neighbors that are at most as far from the node as specified by the first argument.

This algorithm uses the getTransitiveNeighbors(int) method of the Node interface that returns all neighbors that are close enough -- i.e. that are accessible by at most \( n \) edges where \( n \) is an argument to that method.

Use this algorithm and measure the speed for different values of the argument \( n \). Find out for which values of the argument the measured times are comparable to those obtained from the settings described in 2 and 3. Use randomly generated sparse and dense graphs and test in a higher latency environment.