First assignment

Java RMI

Send the finished task by e-mail to your teaching assistant. Deadlines for the submission are on the web page of the course: http://d3s.mff.cuni.cz/teaching/middleware/.

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, rmiregistry application).
- Compilation (rmic application before Java 8).

A simple example of a remote function call for Java RMI is available.

Assignment details

http://d3s.mff.cuni.cz/teaching/middleware/files/as1.zip

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

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

The `addNeighbor(Node)` method adds the object to a set of neighbors of the called object and is used when generating a graph. The `getNeighbors()` method returns a set of all neighbors of an object and is used for distance computation. 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 oFrom, Node oTo);
}
```

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 `Searcher` interface. Update the provided implementation to allow search through the remote interface along the local one. The server `Searcher` interface would accept local objects with the `Node` interface.
Measure and compare the speed of the implemented variants. How does the server Searcher access the local 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 the graph distance using a local Searcher working with the server Node objects along the existing functionality.

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

4. Add another variant: compute the distance using the server Searcher interface to which you pass (from the client) the graph as server Node objects.

Measure and compare the speed again. How does the server Searcher access the server Node objects?

5. Compare the speed of all variants when both client and server are running on the same machine vs. a situation when client and server are on different physical computers connected by a network.

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), that 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 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. Test in a higher latency environment, e.g. between networks in the campus and in the local lab.