GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computing & Mathematics

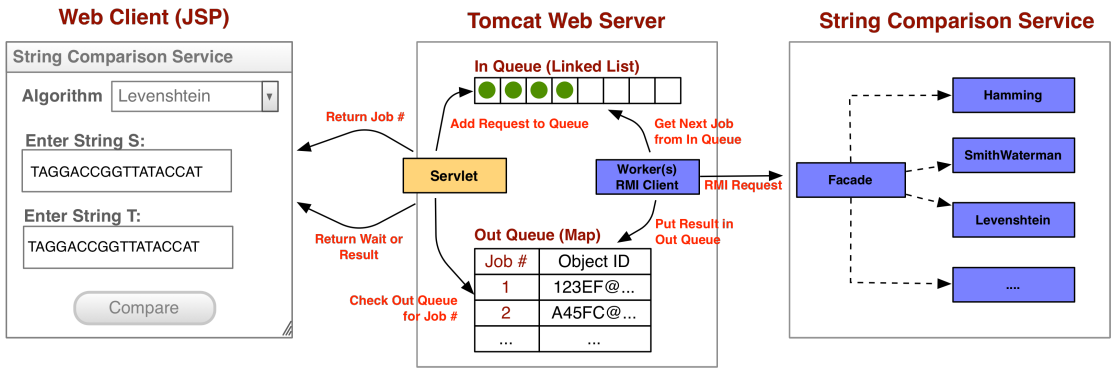
B.Sc. Software Development – Distributed Systems (2016)
ASSIGNMENT DESCRIPTION & SCHEDULE

An Asynchronous RMI String Comparison Service

Note: This assignment will constitute 50% of the total marks for this module.

1. Overview

You are required to use the Java RMI framework to develop a remote, *asynchronous* string comparison service. A client should be able to remotely connect and pass two strings to the service for comparison. The service should use one of a number of optional string comparison algorithms to compute the edit distance or optimal alignment between the two strings. An overview of string comparison methods is provided in Section 5. The following diagram depicts the overall system architecture:



Submitted form requests should be handled by a servlet and add to a message queue (the In-Queue). When the remote RMI service receives a request from the worker RMI client request, it should **immediately** return a separate remote object (an instance of ***Resultator***) that is added to the *Out-Queue*. Items in the *In-Queue* should be periodically polled and processed by dispatching a request to the RMI service. As the RMI client already has a remote reference to the ***Resultator*** object, it can call a method called *isProcessed()* to check if the string comparison process is complete.

The point of this exercise is to give you some "hands-on" experience programming an asynchronous remote software service (a message façade). Asynchronous communication is an important topic in distributed computing, as it provides a degree of scalability if the number of potential requests is unknown or may vary significantly.

2. Minimum Requirements

You are required to use the RMI framework to implement the string comparison service. Your implementation should include the following features:

1. A web client request should be placed in a message queue to await processing. Each request should be allocated a job number. The job number should be added to an out-queue (a Map) with an initial mapping of ***job number → Resultator***. The servlet handler should return the job number to the client which in turn should poll the server every 10 seconds for a response. When a response is received with a completed task, the result of the string comparison should be displayed in the browser. *A web application stub for this component of the assignment is available on Moodle.*

2. An interface called ***StringService*** should expose a remote method with the following signature:

   public Resultator compare(String s, String t, String algo) throws RemoteException;

   where *s* and *t* are the two strings to compare, *algo* is the string matching algorithm to use and *Resultator* is a ***remote object reference*** that allows the RMI service provider to push an asynchronous response to the RMI requestor (a pass by reference from the service provider to the service requestor). The string comparison service should delegate calls to an instance of the algorithm specified in the request, running in a separate thread. Before comparing the strings, the string algorithm thread should be put to sleep for a time, i.e. *Thread.sleep(1000)*, to slow the service down and simulate a real asynchronous service.

3. The interface called ***Resultator***[1] should expose the following remote methods:

   ```
   public String getResult() throws RemoteException;
   public void setResult(String result) throws RemoteException;
   public boolean isProcessed() throws RemoteException;
   public void setProcessed() throws RemoteException;
   ```

   The remote method *setResult()* should be used by the service provider to update the state of the returned "pass-by-reference" object with a relevant string response (edit distance or optimal alignment). The method *getResult()* should return this text to a caller. The service provider should flag the *Resultator* object as processed by calling the method *setProcessed()*. Finally, calling *isProcessed()* should return whether or not the process is complete. Note that the *Resultator* object is a remote object reference that is returned when the remote method *compare(...)* is called. This object should be stored in the Out-Queue (Map).

3. Deployment and Delivery

The project must be submitted by midnight on Friday 9th December 2016 using both Moodle and GitHub.

- **GitHub**: submit the HTTPS clone URL, e.g. https://github.com/my- account/my-repo.git, of the public repository for your project. All your source code should be available at the GitHub URL.

[1] -ator, suffix: a person or thing that performs a certain action: agitator, escalator, radiator.

- **Moodle**: submit a Zip archive using the Moodle upload facility (under "Main Assignment (50%) Upload". Ensure that the name of the Zip archive is be *{id}*.zip where *{id}* is your GMIT student number. The Zip archive should contain the following files and structure:

Marks	Category
README.txt	Contains a description of the application, extra functionality added and the steps required to run the application. This file should BRIEFLY provide the instructions required to execute the project.
comparator.war	A Web Application Archive containing the resources shown under Tomcat Web Application. All environmental variables should be declared in the file WEB-INF/web.xml. You can create the WAR file with the following command from inside the "comparator" folder: **jar –cf comparator.war ***
string-service.jar	A Java Archive containing the RMI String Comparison Service and a servant class with a main() method. The application should be run as follows: **java –cp ./string-service.jar ie.gmit.sw.Servant** You can create the JAR file with the following command from inside the "bin" folder of the Eclipse project: **jar –cf string-service.jar ***

4. Marking Scheme

Marks for the project will be applied using the following criteria:

Marks	Category
(30%)	Tomcat Web Application (including message queues)
(30%)	Asynchronous RMI Service (including RMI client in Tomcat app)
(20%)	Packaging & Distribution (Moodle and GitHub)
(20%)	Documented (and relevant) extras.

Each of the categories above will be scored using the following criteria:

- 0–30%: Not delivering on basic expectation
- 40-50%: Meeting basic expectation
- 60–70%: Tending to exceed expectation
- 80-90%: Exceeding expectations
- 90-100%: Exemplary

5. Approximate String Matching Algorithms

String matching algorithms have been an active area of research in computer science over the last sixty-five years. Among their wide range of applications include text storage, compression and manipulation, speech recognition, spell-checking, grammar parsers and biological sequence analysis. While exact-matching string comparison is a trivial task, the problem of in-exact matching still poses challenges, with heuristics normally used to compare long strings in a space and time efficient manner. The following list outlines the main algorithms used for approximate (in-exact) string matching. These algorithms typically are based on a dynamic programming matrix (DP) and generally have a space and time complexity of $O(n^2)$.

- **Edit Distance Algorithms:**
  The edit distance between two strings is the minimum amount of insertions, deletions or substitutions required to convert one string into another.

- ***Hamming Distance (1950)*:** the number of positions at which the corresponding symbols in two strings of similar length are different.
- ***Levenshtein Distance (1965):*** the minimum edit distance (insertions, deletions or substitutions) between two strings. The space complexity can be reduced from O(n^2) to O(n) using the Iterative Levenshtein Distance algorithm.
- ***Damerau-Levenshtein Distance (1966):*** the minimum edit distance (insertions, deletions, substitutions or transpositions) between two strings.
- ***Jaro–Winkler Distance (1990):*** A variation of the Jaro Distance that returns the edit distances as a fuzzy value [0...1].
- ***Euclidean Distance (~300BC):*** the straight-line distance between two string vectors in Euclidean space.

- **String Alignment Algorithms:**
  - ***Needleman-Wunsch (1970):*** Global pairwise alignment
  - ***Smith Waterman (1981):*** Local Alignment
  - ***Hirschberg's Algorithm (1975):*** a divide and conquer version of Needleman–Wunsch.

Note that highly optimized versions of edit distance algorithms can be implemented using bit vectors, e.g. the Bitap (1992) and Manber-Wu (1994) algorithms. The latter is used by the ***grep*** utility in Linux / Unix for searching from a command line.

See the following URLs for further details on string comparison algorithms:

- http://www.ibm.com/developerworks/library/j-seqalign
- http://ntz-develop.blogspot.ie/2011/03/fuzzy-string-search.html

Please note that ***you are not expected to re-write these well-known algorithms*** for this project. You should feel free to use existing Java implementations of these algorithms and incorporate them into your assignment. However, you should ensure that you cite the original source in the source code and/or README.