**Spring, 2009 CIS 455 / 555**

# Internet and Web Systems

## Assignment 3

## Due April 9, 2009

# Introduction

In this assignment, you will explore Web services and distributed hash tables by building a caching system for YouTube videos. You will provide a simple Web form for accepting keyword queries, then you will retrieve matching hits from YouTube, and finally you will save the results in a cache in the distributed hash table, to speed the response time for future queries.

# 1 Requirements

As before, you will implement this assignment in Java. It will consist of several components, each described below. Your search engine will use Web service APIs to perform a keyword search of YouTube[1].

You may not use any external Java libraries except for those explicitly mentioned in this assignment, namely FreePastry and the Google sample code (and BerkeleyDB if you wish).

# 2 Modules

## 2.1 User Interface Servlet

This simple servlet will present a "search" form, accept a POST from this form, and send the keyword (we will assume only a single keyword) to the DHT, as part of a "QUERY" message you will define. It will then wait for a "RESULTS" message you will define, returned by the peer that handles the request.

Note that the servlet should return the search form if it has no POST data, and otherwise it should parse the POST request and return the RESULTs.

[1] http://code.google.com/apis/youtube/developers_guide_java.html#RetrievingVideos

## 2.2 P2P Caching System

Your caching server will consist of a virtual network of search servers/nodes, all running the same software. Each server accepts a "QUERY" message (with the appropriate parameters, such as the keyword) and decides what to do:

1. If the keyword has an entry in the local cache, then directly return a RESULT message containing the cached data. You will not worry about cache expirations for the purposes of this assignment.

2. Otherwise, run a YouTube search for videos matching the keyword, and put the results in the cache.

Note that each peer will handle different keywords. You may cache the results in a hash table, or using BerkeleyDB.

Note that Pastry allows you to simulate multiple virtual nodes on each physical machine, so you may test your code on a single machine. However, you should ultimately try it with 3 or 4 nodes on the `mod` cluster.

You will create a main class that takes the following arguments:

- Number of virtual nodes to start on this machine.

- The Pastry "bootstrap" node's IP address (may be 127.0.0.1).

- The port to use for the Pastry bootstrap node (to allow co-existence with your peers' applications). If you need to allocate more than one port on a given machine, assign additional ports in sequential ascending order, starting from the one given on the command line.

followed by any arguments specific to your server.

If you want to test your code on the `mod` cluster, first start your bootstrap node on one machine. Launch other nodes on the other machines by joining them to the ring initiated by the bootstrap node.

## 2.3 FreePastry

You will use the FreePastry[2] networking layer to implement the DHT-based store. We have provided a `NodeFactory` class that will create multiple nodes in the same ring. This class encapsulates all of the Pastry-specific functionality you will need, namely creating Nodes, shutting down Nodes, and hashing byte data (in your case, the byte value of search strings) into Node Ids; therefore your application will need only use classes in the `rice.p2p.commonapi` class, which is applicable to several difference peer-to-peer networks. The `NodeFactory` class assigns ports in increasing order from the starting port, as required by this assignment.

[2] http://freepastry.org/FreePastry/

As mentioned above, you need to figure out which node is responsible for storing a particular cached query result. You can do this by getting a byte string that represents the query, and using `NodeFactory`'s `getIdFromBytes` method to create a Pastry node ID from that byte string; you can then use the `Endpoint` route method to send a message to the node whose ID is closest to the ID you created. You do not need to deal with adding or removing nodes after searches have begun, meaning that the same node should always be assigned to a particular query. How the DHT node stores its local cache of query results is up to you. A simplistic approach would be to use an in-memory `HashMap`; a more sophisticated one, worth 5% extra credit, is to use BerkeleyDB.

Section 5 contains a brief overview of creating an application using the Rice common API mentioned above. There is also a Pastry tutorial available online[3], though much of it is not relevant since the `NodeFactory` class deals with the complexity of creating a network; you need focus only on the parts describing the `Application` interface.

# 3 Testing

As per usual, you should ensure your code is tested. We would like a *minimum* of two JUnit tests, one for the Web service requesting system and one for the caching / storage system. We recommend further tests as well.

# 4 Submission Instructions

As with the last assignment, you will use the turnin program for submission. Your submission should be given the name "a3". It should contain the source code for the search engine application and the source code for the servlet user interface, as well as the JUnit tests. You should also include a README file describing any special features or problems with your implementation, and which extra credit options you chose to implement. If you have tested the user interface on your application server, include the source code to that as well and brief instructions as to how to run the application on your application server; otherwise include a build.xml file that will install the application on a tomcat instance.

# 5 Brief Overview of a Pastry Application

This section describes how to create a simple Pastry application using the common API found in the package `rice.p2p.commonapi`. The most important class in such an application is the one that implements the `Application` interface; this is the class that receives messages from the peer-to-peer network. Here I give code for a very simple application:

```
import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;
```

[3] http://freepastry.org/FreePastry/tutorial/

```
import rice.p2p.commonapi.Node;
import rice.p2p.commonapi.NodeHandle;
import rice.p2p.commonapi.Endpoint;
import rice.p2p.commonapi.Application;
import rice.p2p.commonapi.RouteMessage;

public class SimpleApp implements Application {
    NodeFactory nodeFactory;
    Node node;
    Endpoint endpoint;

    public SimpleApp(NodeFactory nodeFactory) {
        this.nodeFactory = nodeFactory;
        this.node = nodeFactory.getNode();
        this.endpoint = node.registerApplication(this, "Simple Application");
    }

    public void deliver(Id id, Message message) {
        // Ignore incoming messages for now...
    }

    public void update(NodeHandle handle, boolean joined) {
        // This method will always be empty in your assignment
    }

    public boolean forward(RouteMessage routeMessage) {
        // This method will always return true in your assignment
        return true;
    }
}
```

The constructor takes as an argument the `Node` upon which the application is to run; it then registers itself with that node. You should use the same application name as an argument to the `registerApplication` method in each instance of the application running on any server. The registration process gives an `Endpoint` which can be used to send messages to other peers in the network. The `deliver` method is called when the peer receives a message; we'll go into more detail about that later. The last two methods, `update` and `forward`, are useful for programming more sophisticated functionality than is needed for this assignment, so you can safely leave them as empty and returing `true` respectively.

This simple application doesn't do anything useful, however; it doesn't send any messages and it ignores messages it receives. Now let's try write our own message class we can use to send and receive messages:

```
public class OurMessage implements rice.p2p.commonapi.Message {
```

```
    NodeHandle from;
    String content;
    boolean wantResponse = true;
    public OurMessage(NodeHandle from, String content) {
        this.from = from;
        this.content = content;
    }
}
```

This message carries a `String` as its content, and a NodeHandle (basically a server/port pair) of the node that sent the message so the node that receives it can send a response directly back without having to send the message around the ring.

Now, let's add a new method to our `SimpleApp`, which can be used to send a message to another peer, and update the `deliver` method so it does something with a message when it receives one:

```
    void sendMessage(Id idToSendTo, String messageToSend) {
        OurMessage m = new OurMessage(node.getLocalNodeHandle(), messageToSend);
        endpoint.route(idToSendTo, m, null);
    }

    public void deliver(Id id, Message message) {
        OurMessage om = (OurMessage) message;
        System.out.println("Received message " + om.content + " from " + om.from);

        if (om.wantResponse) {
            OurMessage reply = new OurMessage(node.getLocalNodeHandle(),
                "Message received");
            reply.wantResponse = false;
            endpoint.route(null, reply, om.from);
        }
    }
```

With this modified class, the object that owns the instance of `SimpleApp` can send a message to the node closest to a particular ID; it should then receive a reply back from that node. This demonstrates the two ways you can use the `route` method; you can either send a message to the node that "owns" a particular ID (as in `sendMessage`) or you can send a message directly to a node if you have its `NodeHandle` (as in `deliver`).