CIS 455/555: Internet and Web Systems

Spring 2013

Assignment 3: Web Services and Decentralized Systems

Due April 5, 2013, at 10:00pm EDT

1 Introduction

In this assignment, you will explore Web services and distributed hash tables by building a caching system for YouTube videos based on FreePastry. 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 (DHT), to speed the response time for future queries.

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. You can find more information about the YouTube Data API on the following web page: https://developers.google.com/youtube/2.0/developers_guide_protocol_api_query_parameters.

2 Modules

2.1 User Interface Servlet

This simple servlet, which should be in `edu.upenn.cis455.youtube.YouTubeSearch`, will present a search form, accept a `POST` from this form, and send the keyword (we will assume only a single keyword) to a machine in the DHT. The servlet will be initialized with an IP address / hostname and a port for one of the DHT cache servers, as parameters `cacheServer` and `cacheServerPort`, respectively, in `web.xml`. It will forward each request to this address using either a SOAP-style or a REST-style interface, and wait for a response. 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`.

2.2 P2P Caching System

Your caching service will consist of a virtual network of search servers/nodes, all running the same software. Each server should have a daemon listening on the `cacheServerPort`, waiting for search requests from the servlet. (Only one peer machine will typically be the one that receives these requests, but all machines will have identical software.) Given such a request, the peer should create a `QUERY` message you will define, and forward this to the appropriate DHT node. It will then wait for a `RESULTS` message you will define, returned by the peer that handles the request. The contents of this message will be sent back as a response to the requesting servlet.

2.3 YouTube client

Each search server must have a class `edu.upenn.cis455.youtube.YouTubeClient`. This class must have a constructor that takes a single `String` argument, which contains the path to the BerkeleyDB database (if you're implementing Extra Credit 6.1) or otherwise is the empty string. The class must also

have a method `String searchVideos(String keyword)` with the functionality described below.

When the server is started, it instantiates this class, starts a Pastry node, and then waits for `QUERY` messages to arrive through the DHT. Each message must contain the appropriate parameters, such as the keyword that the user is searching for. When a message arrives, the server then calls the `searchVideos` method, which decides what to do:

1. The method checks whether the keyword has an entry in the local cache. It then prints a message "`Query for <keyword> resulted in a cache <hit|miss>`" to `System.err`, where <keyword> is the keyword that is being queried, and <hit|miss> is either `HIT` or `MISS`.
2. If the keyword has an entry in the cache, the method directly constructs and returns a `RESULT` message (REST or SOAP) that contains the cached data. (You need not worry about cache expirations for the purposes of this assignment.)
3. Otherwise, the method runs a YouTube search for videos matching the keyword, put the results into the cache, and only then constructs the `RESULT` message.

The server should then return the `RESULT` message. Note that each peer will handle different keywords.

You may cache the results in a hash table, or using BerkeleyDB. Note also 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 three or four nodes on the `spec` cluster. There are 53 machines in this cluster, named `spec01` ... `spec53`, and you can connect to them by using `ssh` to log into `eniac`, and then running '`ssh spec##`'. If you cannot use one of the machines, e.g., because it is down, broken, or currently in use by your classmates, just pick another one.

Please create a main class called `edu.upenn.cis455.youtube.P2PCache` that takes the following arguments (the fifth argument is only used if you implement Extra Credit 6.1):

- The port number on the local machine to which the Pastry node should bind;
- The IP address of the Pastry bootstrap node;
- The port number of the Pastry bootstrap node;
- The port number to which the daemon should bind; and
- The path to the BerkeleyDB database (only if you implement Extra Credit 6.1).

If you want to test your code on the `spec` cluster, first start your bootstrap node on one machine, and then 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 (http://freepastry.org/FreePastry/) P2P substrate 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 `NodeIds`; therefore your application will need only use classes in the `rice.p2p.commonapi` class, which is applicable to several different peer-to-peer networks. The `NodeFactory` class assigns ports in increasing order from the starting port, as required by this assignment.

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 `NodeId` 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 (see 6.1).

Section 5 contains a brief overview of creating an application using the common API mentioned above. There is also a Pastry tutorial available online (http://freepastry.org/FreePastry/tutorial/), 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.

2.4 Ping-pong service

To facilitate testing, your server should implement a 'ping-pong' service, which should work as follows: Once every three seconds, each server picks a random ID and sends a special `PING` message to that ID via key-based routing. When a server receives a `PING` message, it should return a `PONG` message to the sender immediately. The server should print the following to `System.out`:

- When it sends a `PING` message, "Sending PING to xxx", where xxx is the ID that was chosen;
- when it receives a `PING` message, "Received PING to ID xxx from node yyy; returning PONG", where xxx is the ID to which the `PING` was sent, and yyy is the nodeID of the sender; and
- when it receives a `PONG` message, "Received `PONG` from node zzz", where zzz is the nodeID of the node that has sent the `PONG`.

We recommend that you implement the ping-pong service *first*, and use it to check that your Pastry ring is working correctly. When everything is working, each server should receive exactly one `PONG` for every `PING` it sends, and each of the servers should be sending a `PONG` once in a while.

2.4 Unit tests

As usual, you should test your code carefully. You should submit a *minimum* of two JUnit test suites, one for the Web service requesting system and one for the caching/storage system. We recommend adding further tests as well.

3 Developing and running your code

We strongly recommend that you do the following before you start writing code:

1. Carefully read the entire assignment from front to back and make a list of the features you need to implement.
2. Spend at least some time thinking about the *design* of your solution. What classes will you need? How many threads will there be? What will their interfaces look like? Which data structures need synchronization? And so on.
3. Check in your changes into svn regularly. This will give you many useful features, including a recent backup and the ability to roll back any changes that have mysteriously broken your code.

We recommend that you continue using the VM image we have provided for the previous assignments. You should be able to do most of your development in the VM image, including small FreePastry deployments; however, you should ultimately try setting up a small ring (say, three or four nodes) on the `spec` cluster.

The VM image should already contain all the tools you will need for HW3; however, you will need to add FreePastry, as well as the JAR files for the Google data API from http://gdata-java-client.googlecode.com/files/gdata-src.java-1.46.0.zip. You can check out the framework code for HW3 using the same process as for HW0-HW2 (there should now be an additonal "HW3" folder).

Of course, you are free to use any other Java IDE you like, or no IDE at all, and you do not have to use any of the scripts we provide. However, to ensure efficient grading, your submission **must** meet the requirements specified below – in particular, it must build and run correctly in the original VM image and have an `ant` build script. If you are using our scripts, this should be the case automatically.

4 Requirements

Your solution must meet the following requirements (please read carefully!):

1. Your servlet class must be called `edu.upenn.cis455.youtube.YouTubeSearch`, and the main class of your cache must be `edu.upenn.cis455.youtube.P2PCache`. Please check the capitalization – `P2PCache` is not the same as `P2pCache`! The parameters of your servlet must follow the specification in Section 2. You must also have a class in your server called `edu.upenn.cis455.youtube.YouTubeClient`, which must have the constructor and the `searchVideos` method described in Section 2.3 (with that exact signature).
2. Your submission must contain a) the entire source code for the search engine application and the servlet user interface, as well as the JUnit tests and any supplementary files needed to build your solution, b) an `ant` build script called `build.xml`, and c) a `README` file. The `README` file must contain 1) your full name and SEAS login name, 2) a description of features implemented, 3) any extra credit claimed, 4) a list of source files included, and 5) brief instructions on how to run the application on your application server. You must also complete all the yes/no questions.
3. When your submission is unpacked in the original VM image and the `ant` build script is run, your solution must compile correctly. Please test this before submitting!
4. Your servlet must display your full name and SEAS login name on the web form, and your FreePastry application must output the same information when it is started. We use this as a sanity check during grading.
5. Your solution must be submitted via `turnin` (!) before 10:00pm EDT on April 5, 2013. The only exception is if you have redeemed some of your jokers via the web interface before the deadline. (Please keep in mind that this is the last assignment for which jokers can be used!) The project name for `turnin` should be `hw3`.
6. Your code must contain a reasonable amount of useful documentation.

Reminder: All the code you submit (other than the Google Data API, the FreePastry framework, and any code we have provided) must have been written by you personally, and you may not collaborate with anyone else on this assignment. Copying code from the web is considered plagiarism.

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 is the code for a very simple application:

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

```
import rice.p2p.commonapi.Message;
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.buildEndpoint(this, "Simple App");
    this.endpoint.register();
  }

  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 `registerApplication` 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 to 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 msgToSend) {
  OurMessage m = new OurMessage(node.getLocalNodeHandle(), msgToSend);
  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`).

1.send to Node with NodeID
2.send to node with NodeHandle

6 Extra Credit

6.1 BerkeleyDB Store (+5%)

Use BerkeleyDB to store the local cache of query results on each node.

6.2 Cache Eviction and Expiration (+5%)

Add a configurable maximum storage size and content expiration time to each DHT node. If an object has been stored for longer than the expiration time, it should be deleted from the cache. If the size of the cache exceeds the maximum storage size, the least recently accessed objects should be deleted until the maximum size is reached again. As a special case, if the cache contains only one object but the size of this object exceeds the maximum, the object should be allowed to remain. In your `README` file, you should document how the parameters can be set.

6.3 Management Interface (+10%)

Add a second web interface that can be used to view the status of the cache nodes in the DHT. At any given time, the interface should show the status of three nodes: a 'center' node and its left and right neighbor in the ring. If the ring contains fewer than three nodes, all the nodes should be displayed.

For each node, the interface should show 1) the node's ID; 2) a list of cached files, as well as their sizes and (if you also implemented 6.2) their expiration times. When the node ID of the left or right neighbor is clicked, that node should become the center node, and the display should be updated (this makes it possible to 'walk' the ring, and to view the status of each node). There should also be a way to remove individual files from the cache, as well as a way to refresh the display. In your `README` file, you should provide the URL of the management interface and provide a brief description of how to use it.