1 Introduction

In this assignment, you will explore several more Web technologies, and continue to build components useful towards your course project, by building a topic-specific Web crawler. A topic-specific crawler looks for documents or data matching a particular category – here, the category will be specified as an XPath expression.

This assignment will entail:

- writing a servlet-based Web application that extends the one from Milestone 1, and enables topic-specific “channels” defined by a set of XPath expressions, displays documents that match a channel;
- adapting your crawler that traverses the Web, looking for HTML and XML documents that match one of the XPath expressions;
- routing documents from the crawler through a stream engine for processing one at a time;
- writing an XPath evaluation engine that determines if an HTML or XML document matches an XPath expression;
- building a persistent data store, using Oracle Berkeley DB, to hold retrieved HTML/XML documents and channel definitions.

The resulting application will be very versatile; one use of it could be to write an RSS aggregator with keyword-defined channels. In this case, the XPath expressions would find RSS feeds (which are just XML documents) that contain articles with the specified keywords, and the stylesheet would select the matching articles from the feeds and format them for the user. However, your application will have many other uses as well.

2 Developing and running your code

These instructions are the same as in Milestone 1. We strongly recommend that you do the following before you start writing code:

1. Carefully read the entire assignment (both milestones) from front to back and make a list of the features you need to implement. There are many important details that are easily overlooked!
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.
We recommend that you continue using the VM image we have provided for HW0 and HW1. This image should already contain all the tools you will need for HW2. You can check out the framework code for HW2 using the same process as for HW0 and HW1 (check out from https://bitbucket.org/upenn-cis555/555-hw2.git). You should, of course, commit the code to your own repository on bitbucket, in case your virtual machine crashes.

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.

We strongly recommend that you regularly check the discussions on Piazza for clarifications and solutions to common problems.

3 Milestone 2: Streaming Crawler and XPath engine

The next milestone will consist of an XPath evaluator for streams of documents. You will extend your Milestone 1 project to run on a stream processing engine that enables multithreaded execution. You’ll also extend your servlet and storage system to enable users to register “subscription channels” with the system. Finally, you’ll build a stream processing component that checks documents against the various “channels” and outputs results per user.

You can implement XPath matching in any (correct!) way of your preference: one option is to follow the SAX-based model established in XFilter (which will be extra credit), but one can also use other schemes involving recursive traversal and matching of the XPath on the DOM (which will be “normal” credit).

3.1 Rework the Crawler as a “Spout,” “Bolt,” and Shared Modules in StormLite

Your Milestone 1 project had a simple execution model, in which you controlled the execution of the crawler and presumably did this in a crawler loop. Now we want to break in into a smaller unit of work that can be parallelized.

To do this, we’ll be using a CIS 455/555-custom emulation of the Apache Storm stream engine, which we call StormLite (it should show up in your HW2 repo already). Please see the documents on StormLite and see TestWordCount as an example of a multithreaded stream job. Storm has spouts that produce data one result at a time, and bolts that process data one result at a time.

You should refactor your Milestone 1 to refactor the crawl task to run within StormLite, as follows. Note that StormLite supports multiple worker threads but you can control the number of “executors” (and start with 1).

1. You’ll maintain (or update) your frontier/crawl queue of URLs from Milestone 1. However, you want to place it in a “spout” (implement IRichSpout) so it
sends URLs one at a time to the crawler via the nextTuple interface.

2. You’ll maintain your BerkeleyDB storage system from Milestone 1. This will also be a shared object, at least across some aspects of your Milestone 2 implementation. Again, you may want to use a “singleton factory” pattern.

3. The crawler should be placed in a bolt – its execute method gets called once for each URL from the crawler queue. The crawler should output documents one at a time to its output stream. See the IRichBolt interface and the example code.

4. The next step should be a bolt whose execute method takes a document, writes it to the BerkeleyDB storage system, and outputs a stream of extracted URLs.

5. Finally, there should be a bolt that filters URLs (using appropriate techniques and data structures) and updates the shared frontier queue.

Based on the sample code in test.upenn.cis.stormlite, the StormLite PDF document, and (in fact) the documentation on Storm available from Stack Overflow and the Web, you should be able to assemble a stream dataflow like the one illustrated above.

Your new implementation on StormLite should make it fairly easy to extend to a multithreaded, extra-credit-earning version of the crawler (see Section 5.1). You can assume a single crawler queue bolt, but should look at how the fieldGrouping and shuffleGrouping specifiers allow you to specify how data gets distributed when there are multiple “executors” such as multiple copies of the crawler, parser, etc.

3.2 Extended Servlet-based Web Interface
For Milestone 2, you will also enhance the servlet to support the following functions for logged in users.

Channels
Now that you have users and XML, we want to “connect” users with “interesting” XML. To do this, any logged in user will be able to create channels. A channel is a named pattern describing a class of documents. An example of a channel definition would be

sports : /rss/channel[contains(text(),"sports")]”

and you can see an example of content that would match the channel at:
http://rss.nytimes.com/services/xml/rss/nyt/Sports.xml

Assume that channels and their names are global.

You should implement a servlet interface to create a channel, as a GET call:

localhost:8080/create/{name}?xpath={xpath-pattern}

Updated Login Screen
As before, you should have a login/registration form at localhost:8080/register.jsp. Once a user is logged in, you should have a “home page” at localhost:8080/.

- List all channels available on the system
• Display the documents matching a subscribed channel, which triggers the servlet at

localhost:8080/show?channel={name}

Only if the user has subscribed to the channel.

Obviously you’ll need to add some logic to the BerkeleyDB storage system to store user subscriptions and to store which documents correspond to a channel (see Section 3.2 for how this will be populated). How you implement most of the functionality of the Web interface is entirely up to you; we are just constraining the URL interfaces. In order to make things consistent across assignments, we are specifying how the channel must be displayed by the “show” request.

• For each channel, a <div class="channelheader"> element around its header, with the string “Channel name: ” followed by the name of the channel; a and the string “created by: ” followed by the name of the user who created the channel.
• For each XML (e.g., RSS) document that matched an XPath in the channel:
  o The string “Crawled on: ” followed by the in the same format as 2002-10-31T17:45:48, i.e. YYYY-MM-DDThh:mm:ss, where the T is a separator between the day and the time.
  o The string “Location:” followed by the URL of the document
  o A <div class="document"> element with the contents of the document.

We expect this application to run on your application server from the first assignment. If you did not complete the first assignment, or for some other reason do not want to continue to use the application server that you wrote, you may use Jetty (http://www.eclipse.org/jetty/) as described previously to test your servlet. There will not be any penalties for using Jetty or Tomcat.

3.2 XPath Engine as a StormLite Bolt

You need to write a class edu.upenn.cis455.xpathengine.XPathEngineImpl that implements edu.upenn.cis455.xpathengine.XPathEngine (included with the code in Bitbucket), and evaluates a set of XPath expressions on the specified HTML or XML document. Once you’ve tested that individually, you’ll incorporate it into (call it from) a StormLite bolt.

The implementation object (instance of XPathEngineImpl) should be created via the XPathEngineFactory. The setXPaths method gives the class a number of XPaths to evaluate. The isValid(i) method should return false if the i-th XPath was invalid, and true otherwise. You can implement one of two strategies:

• DOM parsing. Here, you should make isSAX() return false. Then implement the evaluate(d) method, which takes a DOM root node as an argument. This contains the representation of a HTML or XML document, and it returns an array with the i-th element set to true if the document matches the i-th XPath expression, and false otherwise.
• SAX / event-driven parsing. You should implement evaluateSAX() which is for event-driven parsing, which takes in the unparsed document as a stream as well as (optionally) an event handler (which you supply via the XPathEngineFactory). If you choose this implementation, your isSAX() method should return true.

To make things simpler, we are supporting a very limited subset of XPath, as specified by the following grammar (modulo white space, which your engine should ignore):
XPath → axis step
  axis → /
  step → nodename ([test])? (axis step)?
  test → step
  → text() = "..."
  → contains(text(), "...")
  → @attname = "...

where nodename and attname are valid XML identifiers, and "..." indicates a quoted string. This means that a query like /db/record/name[text() = "Alice"] is valid but the similar (and valid XPath) query /db/record/child::name[text() = "Bob"] is not. Recall that if two separate bracketed conditions are imposed at the same step in a query, both must apply for a node to be in the answer set.

Below are some examples of valid XPaths that you need to support (not an exhaustive list):

```
/foo/bar/xyz
/foo/bar[@att="123"]
/xyz/abc[contains(text(),"someSubstring")]
/a/b/c[text()="theEntireText"]
/blah[anotherElement]
/this/that[something/else]
/d/e/f[foo[text()="something"]][bar]
/a/b/c[text() = "whiteSpacesShouldNotMatter"]
```

If you have never written a parser for a context-free grammar before, we encourage you to write a “recursive descent parser.” If you Google around a bit, you should find plenty of explanations and example code on the Web (even the Wikipedia page has some!). (Of course we expect you to write your own code, but these examples should give you an idea of how it might look.)

You will want to think of the XPath match as a recursive process, where an XPath is matched if its step node-test matches and all of its tests (recursively) match and its next step matches. The easiest HTML/XML parser to use in Java is probably a DOM (Document Object Model) parser. Such a parser builds an in-memory data structure holding an entire HTML or XML document. From there, it is relatively easy to evaluate an XPath expression recursively. Sample code using a DOM (as well as SAX) parser is available in the examples directory. You do not need to use a SAX parser as with XFilter, but you can do so for extra credit (see Section 5.2).

Once your XPath engine works over individual documents, you’ll want to write a StormLite bolt whose `execute()` method instantiates the XPath engine for a given input document (passed in as a tuple), looks up all of the channels defined in the BerkeleyDB database, and for each document that matches an XPath for a channel, records the document as a match to the channel. Subsequently, the servlet interface will be able to show the documents as matches.
3.3 Unit Tests
To test the XPath engine, you are to supply XPath rules to find RSS 2.0 documents which contain items whose title or description contains the character strings war or peace. This should be in a file called warandpeace.xp.

In addition, you must implement at least 5 unit tests, using the JUnit 4 package (see the section on Testing below for helpful Web page references). JUnit provides automated test scaffolding for your code: you can set up a set of basic objects for all of the test cases, then have the test cases run one-by-one.

Your JUnit test suite should instantiate any necessary objects with some test data (e.g., parse a given HTML or XML document or build a DOM tree), then run a series of unit tests that validate your servlet and your XPath matcher. In total you must have at least 5 unit tests (perhaps each designed to exercise some particular functionality) and at least one must be for the servlet and one for the XPath evaluator.

Note that you should use JUnit 3 for this assignment, not any other version.

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

1. Your servlet class must be called edu.upenn.cis455.servlet.XPathServlet. Please check the capitalization: XPath is not the same as Xpath! The XPath engine class must implement the edu.upenn.cis455.xpathengine.XPathEngine interface. Note that there are two alternative methods for XPath evaluation – one based on DOM and one on SAX. The SAX version can simply return null (and the method isSAX should return false) if you are not doing the extra-credit parser.
   a. Your XPath engine class must be must be created by the XPathFactory when the appropriate static method is called.
   b. If you do implement the event-driven interface, your SAX parser event handler must also be returned by the XPathFactory if you need one to be passed into the engine.
2. Your submission must contain a) the entire source code, as well as any supplementary files needed to build your solution, b) a working ant build script called build.xml (such as the one included in the svn repository), 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) any special instructions for building or running. You must also complete all the yes/no questions in the file.
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. You must implement your own HTTP client (open socket, send headers, parse response, etc.), but not HTTPS client.
5. Your code must contain a reasonable amount of useful documentation.

Reminder: All the code you submit (other than the Mozilla/JTidy/TagSoup parser, the standard Java libraries, 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.
4 Testing

4.1 'Sandbox'
We have implemented a small sandbox for you to test your code on. It runs on machines in Penn Engineering, so it will be fast to access, and it will not contain any links out of itself. The start URL of the sandbox is https://dbappserv.cis.upenn.edu/crawltest.html. There should be adequate XML and HTML documents there to test your XPath matching.

4.2 JUnit
In order to encourage modularization and test driven development, you will be required to code test cases using the JUnit package (http://www.junit.org/) – a framework that allows you to write and run tests over your modules. A single test case consists of a class of methods, each of which (usually) tests one of your source classes and its methods. A test suite is a class that allows you to run all your test cases as a single program. You can get more information here: http://www.onjava.com/pub/a/onjava/2004/02/04/juie.html.

For Milestone 1, you must include 5 test cases and for Milestone 2, a test suite consisting of these 5 and at least 2 more for each new component. If your test suite uses any files (e.g., test inputs), please put them into your project folder and use a relative path, so your tests will run correctly on the graders' machines.

5 Extra credit
There are several enhancements you can add to your assignment for extra credit. In all cases, if you implement an improved component, you do not need to implement the simpler version described above; however, your improved component must still pass our test suite for the basic version, and you will lose points if it does not. A safer approach is to implement the basic version first and to extend it later; if you choose to do this, and if you submit both versions, please document in your README file how to enable the extra functionality.

5.1 Advanced crawler design (+5%/+15%)
You can implement a multi-threaded crawler for 5% extra credit. You can implement a crawler based on the Mercator design, namely supporting their innovations in the 'URL Frontier' and 'Content-Seen Test' sections, for 15% extra credit. This should fit into the StormLite framework.

5.2 DFA-based XPath engine (+20%)
You can implement a DFA-based XPath engine for 20% extra credit; see the XFilter paper for a starting point. This will entail using a SAX parser. Your XPath engine should itself be the document handler for SAX. It will get the XPaths via setXPaths() at the beginning, so it can translate them into path nodes. Calls to evaluate() will be parametrized with the document content and (optionally) a SAX handler (which you should provide through the XPathEngineFactory if you need one; otherwise that should return null). evaluate() will parse, handle events, and then return the array of booleans.

5.3 Channel subscriptions (+5%)
For 5% extra credit, allow users to choose which channels to subscribe to from a list of available channels. Add a list of the channels to which they’re subscribed, in addition to all channels. Add a “subscription” servlet at: localhost:8080/subscribe?channel={name}
5.4 Crawler web interface (+15%)

For 15% extra credit, provide a Web interface for the crawler. An admin user (not all users) should be able to start the crawler at a specific page, set crawler parameters, stop the crawler if it is running, and display statistics about the crawler's execution, such as

- the number of HTML pages scanned for links,
- the number of XML documents retrieved,
- the amount of data downloaded,
- the number of servers visited,
- the number of XML documents that match each channel, and
- the servers with the most XML documents that match one of the channels.

This will entail using some sort of interprocess communication, so it's not for the faint of heart.