SLS Lecture Browser and Server Implementation

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

September 5, 2007

Contents

1 System Organization

The SLS Lecture Browser has two major components, a web client that handles user interaction and media presentation, and a web server that responds to lecture queries from the client. There is also a small program for adding lectures to the database and a second web server that provide access to the lecture media.

1.1 Client/Server Interaction

The lecture browser is an example of a web application, which means that a client program, typically a web browser, provides the user interface to an application that runs on a web server. The user starts the application on the web server by visiting a web page. In response to user actions, the client will make requests to the web server and take some action based on the response from the web server.

There are many approaches to web applications. The lecture browser uses *AJAX* and *dynamic HTML*. In its initial request to the server, the browser retrieves the initial web page layout and all the software that it will use. Thereafter, all clicks, button presses, etc. are handled by a program running in the browser. That program asynchronously sends queries to the server, and the server returns the results as XML. The client program then locally modifies the HTML to format the results. This is how the browser is able to show help tips as the mouse is moved and highlight words as they are played.

1.2 Lecture Attributes

We currently have two sources of media, lectures from MIT Open Courseware, and lectures from MIT World, some of which are part of a seminar series, and some of which are just lectures. We have manually created short descriptions of the courses, seminars, and seminar series, and associated titles, dates, speaker information, and URLs for the media with the lectures. We also the lecture to be associated with zero or more categories, such as "physics," and a group, such as "mitworld."[1] Ideally, all this information would be stored in a repository of some sort, but at this time it is simply recorded in files that serve as input to the indexing process.

As for the lectures themselves, we manually extract the waveform from the media file. For unknown reasons, this sometimes results in a slight linear discrepancy in timing between the extracted waveform and the RealPlayer file. Since the time-aligned transcriptions are based on the extracted waveforms, each lecture has an associated timescale which can be used to compensate for the difference. After lectures are transcribed, they are automatically divided into *segments* based on content.

1.3 Indexed Data

In addition to indexing information about the lectures, we maintain a tables of courses, seminar series, and categories. When a lecture is added to the lecture index, it is assigned a unique identifier which is associated with any course, seminar series, and category information, as well as speaker, date, etc. Each segment of the transcription is assigned a unique segment identifier and associated with the lecture. Then each word of the segment is assigned a

[1]The group is not currently used.

unique word identifier, which is associared with the segment identifier, begin and end time of the word, and the actual text.

All the words of a segment are treated as a single document, and, together with the group and the course, seminar series, category, and lecture identifiers as attributes and indexed by a text indexer. Thus a text query identifies segments, and text queries can be filtered based on lectures, courses, seminar series, categories, and groups. The segments located during a text query are scanned for matching words, and then a chunk of text (called a *fragment*) surrounding the matching words is returned.

2 Lecture Server API

This section describes how a client, such as the Lecture Browser, can use the lecture server. The next section will describe the implementation of the lecture server.

The lecture server is a web application that can answer queries about lecture content, access media, authenticate a user, modify a transcription, and index a new lecture. All queries are in the form of an HTTP GET or POST request, and results are in the form of values embedded in XML. Queries can be for the list of lecture categories, or for sets of lectures or parts of lectures. Since GET requests append their parameters to the URL, they can easily be peformed from a browser.

2.1 Category Queries

Category queries have no parameters and return the complete list of category names and their identifiers. The lecture browser uses this query to fill in the menu item with the list of categories. Example output for `http://web.sls.csail.mit.edu/lectures/categories.jsp` is shown in Figure 1. If you type the URL into your web browser address area, your browser will show you the current list of categories. Each item in the list has a name, which the client shows in a menu, and an identifier, which the client can include in lecture queries to restrict the query to a particular category.

```
<categories>
    <category name="Applied Mathematics" categoryid="201"/>
    <category name="Architecture" categoryid="511"/>
    <category name="Arts and Humanities" categoryid="503"/>
    <category name="Astronomy" categoryid="303"/>
    <category name="Biology" categoryid="505"/>
    <category name="Business and Economics" categoryid="501"/>
    <category name="Classical Mechanics" categoryid="302"/>
    <category name="Cognitive Science" categoryid="509"/>
    <category name="Education" categoryid="507"/>
    <category name="Electricity and Magnetism" categoryid="401"/>
    <category name="Engineering" categoryid="506"/>
    <category name="History and Political Science" categoryid="504"/>
    <category name="Linear Algebra" categoryid="102"/>
    <category name="MIT Culture and History" categoryid="508"/>
    <category name="Mathematics" categoryid="101"/>
    <category name="Media" categoryid="510"/>
    <category name="Physics" categoryid="301"/>
    <category name="Speech Processing" categoryid="1"/>
    <category name="Technology and Innovation" categoryid="502"/>
    <category name="Vibrations and Waves" categoryid="402"/>
</categories>
```

Figure 1: `categories.jsp`

```
<results>
  <course/>
  <seminarseries/>
  <lecture>
    <segment>
      <fragment>
        <word/>
      <fragment>
    </segment>
  </lecture>
</results>
```

Figure 2: General Form of Lecture Results

2.2 Lecture Queries

Lecture queries match text and structural attributes to return a list of "hits." It is easiest to describe the query by working backwards from the general result, shown in Figure 2. The results are always enclosed in a `results` tag. Zero or more `course` tags may follow, then zero or more `seminarseries` tags. The `course` and `seminarseries` tags are only included when the results contain lectures associated with the courses or seminar series. After any course and seminar series tags there are zero of more `lecture` tags. Each lecture may have zero of more `segment` tags, each segment zero or more `fragment` tags, and each fragment zero or more `word` tags.

The client often only wants a list of lectures or segments, so the parameter `depth` can be used to control how much detail is returned. The default value is 1, which means only the lectures are returned, so thier segments, fragments, and words are omitted. A depth of 2 will also return the segments, 3 the fragments, and 4 the words.

2.2.1 A Simple Text Query

Another important parameter is `query`, which is a text query suitable for the Apache Lucene text index. Figure 3 shows the results of a query for the text "hacks", requested as `http://web.sls.csail.mit.edu/lectures/lectures.jsp?query=hacks`. This query returned a single lecture which was not part of a seminar series or course. Not all lecture attributes are listed;

```
<results query="hacks" >
  <lecture
   rpmurl="http://web.sls.csail.mit.edu/lectures/lecturerpm.jsp?lectureid=192"
   count="6"
   lectureid="192"
   keywords="cow board bridge tetazoo hack p dome hate floor art"
   date="October 20, 2005"
   name="Where the Sun Shines, There Hack They"
   number=""
   lecturer="Samuel Jay Keyser"
   duration="3642530">
  </lecture>
</results>
```

Figure 3: `lecture.jsp?query=hacks`

some, such as the course and seminar series identifiers, are elided if they are not applicable. Figure 4 describes the various lecture attributes.

2.2.2 Another Text Query

Figure 5 shows the partially elided results of a query to `http://web.sls.csail.mit.edu/lectures/lectures.jsp?query=jupiter` that returns more lectures. With this query, there were two lectures from courses and one from a seminar series. Figure 6 describes the course attributes, and Figure 7 describes the seminar series attributes.

2.2.3 Drilling Down

When browsing, a user typically starts with an initial text query and then clicks on some lecture to see it in more detail. This can be handled by performing a second text query, restricting the lecture to the one they clicked on. In this case, we want to set the `depth` parameter to 2 so that we get the segments. We also want to set the parameter `fillLecture` to `True`. This causes all the segments to be fetched instead of just the ones with hits. This allows the client to show all the segments, highlighting the ones with hits. Figure 8 shows the result, and Figure 9 describes the attributes.

count The number of hits in segments. We only retrieved the lectures, but if we had retrieved with a depth of 3, for fragments, there would have been six fragments for this lecture.

courseid If present, the course identifier for the lecture.

date The date of the lecture.

duration The length of the lecture in milliseconds.

keywords the statistical segmenter identified these words as occurring more often than normal in this lecture.

lectureid The identifier for the lecture, and can be used in queries to restrict the query to a specific lecture.

lecturer The person giving the lecture.

name The name or title of the lecture.

number In a course, the lecture number indicates which lecture in the course this lecture corresponds to, i.e. 1, 2, etc.

rpmurl The URL for the media description required by RealPlayer. RealPlayer requires that the actual media be described with a short amount of text, which will be returned by a GET request to this URL.

seriesid If present, the series identifier for the lecture.

Figure 4: Lecture Attributes

```
<results query="jupiter" >
  <course
   courseid="201"
   institution="MIT"
   department="Physics"
   number="8.01"
   name="Physics I: Classical Mechanics"
   year="1999"
   term=""/>
  <course
   courseid="401"
   institution="MIT"
   department="Physics"
   number="8.03"
   name="Physics III: Vibrations and Waves"
   year="2004"
   term=""/>
  <seminarseries
   seriesid="4"
   institution="MIT"
   name="Poetry@MIT"
   host="MIT Program in Writing and Humanistic Studies"/>
  <lecture
    rpmurl="http://web.sls.csail.mit.edu/...ctureid=168"
    count="1" lectureid="168"
    keywords="shop train poet kevin sofa parish w skylight local frogs"
    seriesid="4" date="October 17, 2002" name="A Reading by Seamus Heaney"
    number="" lecturer="Seamus Heaney" duration="3402053">
  </lecture>
  <lecture
   rpmurl="http://web.sls.csail.mit.edu/...ctureid=189"
   count="1" lectureid="189"
   keywords="rocks apollo moon object ... layers erupt"
   date="April 2, 2003"
   name="The Quest for Mars: Scientific and Human Destiny?"
   number="" lecturer="Jim Garvin" duration="5621425">
  </lecture>
  ...
</results>
```


Figure 5: `lecture.jsp?query=jupiter`

courseid The identifier of the course.

department The department that offered the course.

institution Where the course was given.

name The name of the course.

number The course's institutional number.

term The term of the course, e.g. Fall.

year The year of the course.

Figure 6: Course Attributes

seriesid The identifier for the seminar series.

host The sponsor of the seminar series.

institution Where the seminar series was held.

name The name of the seminar series.

Figure 7: Seminar Series Attributes

```
<results query="jupiter" >
 ...
 <lecture ...>
  <segment
   summary="frequency object omega pi ..."
   count="0" score="1.0"
   beginTime="1571" endTime="384448">
  </segment>
  <segment
   summary="push table pool non ..."
   count="0" score="1.0"
   beginTime="385321" endTime="578013">
  </segment>
  <segment
   summary="planets sun model string orbits..."
   count="1" score="1.0"
   beginTime="578259" endTime="1337630">
  </segment>
  <segment summary="particles gravity direction..."
   count="0" score="1.0"
   beginTime="1340219" endTime="2234896">
  </segment>
  <segment summary="salt nitrate table..."
   count="0" score="1.0"
   beginTime="2235345" endTime="2420363">
  </segment>
  <segment
   summary="bucket string gravity sense..."
   count="0" score="1.0"
   beginTime="2420907" endTime="3043149">
  </segment>
 </lecture>
</results>
```

Figure 8: `lectures.jsp?query=jupiter&lectureid=73&depth=2&fillLecture=True`

beginTime The time in milliseconds in the lecture when the segment begins.

count How many hits are in the segment.

endTime The time in milliseconds in the lecture when the segment ends.

score Sometimes related to the hit score.

summary The keywords that the statistical segmenter determined happened more frequently than normal.

Figure 9: Segment Attributes

beginTime The starting time of the fragment, in milliseconds.

count The number of hits in the segment.

endTime The ending time of the fragment, in milliseconds.

Figure 10: Fragment Attributes

The descriptions of the fragments and words, return in results of depth 3 and 4 respectively, are in Figure 9 and Figure 11.

Figure 12 shows the `lecture.jsp` query parameters.

2.3 Times Queries

Times queries retrieve a subset of a time-aligned transcription based on start and end times within a lecture. These are used by the lecture browser when

beginTime The starting time of the word, in milliseconds.

endTime The ending time of the word, in milliseconds.

term Whether or not the word is a term to be highlighted.

text The text of the word.

Figure 11: Word Attributes

beginTime Only return hits whose fragments start at or beyond this time.

categoryid Only return hits whose lecture includes the specified categoryid.

courseid Only return hits for lectures with the specified courseid.

depth The depth of the lecture tree to return. 1 is lectures, 2 is segments, 3 is fragments, and 4 is words. The default is 1.

endTime Only return hits whose fragments end at or before this time.

fillLecture If True, retrieve lecture segments even if the query did not match. This is useful in drilling down into a lecture since it returns timing information about all of the lecture segments, rather than just those with hits.

highlighter A query of words that should be marked as terms in the result. Ideally, the highlighter would actually mark matching words, but right now it just marks words that appear in the query. For example, if the highlighter query were "bright light" all instances of the words "bright" and all instances of the word "light" within the search results would be marked as terms, not just those places where the sequence "bright light" occurred. Furthermore, "NOT gravity" would highlight the word "gravity" instead of everything else. With some work, it is possible to make the Lucene text retrieval do the right thing.

lectureid Only return hits for lectures with the specified lectureid.

maxHits The maximum number of hits (fragments) to be returned.

query A text query.

seriesid Only return hits for lectures with the specified seminar seriesid.

startHit If a subset of the hits are to be returned, this indicates the offset of the first fragment within the set. This might not be implemented.

Figure 12: `lectures.jsp` Query Parameters

beginTime The starting time in milliseconds for words in the lecture.

endTime The ending time in milliseconds for words in the lecture.

highlighter The highlighter expression used for marking terms as words.

lectureid The lecture identifier for the lecture.

Figure 13: `times.jsp` query parameters

it presents the synchronized transcription. Figure 13. The results are in the form of a `results` containing `fragment` tags containing `word` tags, as described in Figure 10 and Figure11.

2.4 Accessing Media

The RealPlayer browser plugin requires that the location of media be provided indirectly. The page `lecturerpm.jsp` is responsible for this indirection.[2] The page takes one parameter, `lectureid`, and returns a result in format `audio/x-pn-realaudio-plugin` containing the text of the actual URL with the media. We have found that we get the best results by letting an Apache server send the data, rather than using a streaming server or Akamai cached locations.

2.5 User Login

The lecture server includes support for user login. The server allows authenticated users to perform additional tasks, such as submitting changes to transcriptions. At this time, the users and passwords are hard-coded, but the login and capabilities facilities in the server are relatively complete.

Currently, each user has two roles, `login` and `edit`. The `login` role allows the user to login, while the `edit` role allows the user to edit a transcription.

In general, when a user account is created, the user enters a password. Many users are not very good at password generation, so the system also generates a random string called the `salt`. The `salt` and the user password are concatenated, and an MD5 sum is computed and associated with the

[2]Unlike other pages, you cannot access this page from a web browser because the content-type of the result will invoke the media player.

user identifier and the salt. Thus, the password is not itself stored, and any dictionary of common passwords used for attacks will need to be expanded by the number of salt values, which makes the target less desirable.

The first step to login is to get a randomly generated `nonce` and the `salt` for the user from `nonce.jsp` which has `userid` as a parameter. The salt is concatenated with the password the user types to the client and an MD5 sum is computed. The sum is converted to a base-64 string and concatenated with the `nonce`. An MD5 sum of that is converted to a base-64 string and sent to the `login.jsp` page as the `clientHash` parameter.

The `login.jsp` returns an XML `login` tag with attributes of `loginOK` and `editOK` to indicate whether or not login was successful, and whether or not the user is permitted to edit. For the client, this information is only informative; it is the server's session state that determines what the user can do.

NOTE: Unless SSL is used, which we do not use, someone monitoring network traffic would be able to pretend to be in the same session. SSL could be added relatively easy were it deemed necessary. The login procedure does prevent the need for storing passwords on the server and does make the passwords themselves relatively secure.

3 Lecture Server Implementation

The lecture server's job is to convert client queries into appropriate lecture index queries, and convert the results of index queries into XML to be sent back to the client. The majority of the work is handled by the a Java class library called the index, which is described in its own section.

The lecture server is a web application, implemented as an Apache Tomcat servlet. A brief summary of web applications and Apache Tomcat is provided for context before going into details about the server itself.

3.1 Web Application Overview

The client uses the HTTP protocol for requests to the server. In HTTP, all requests are initiated from the client, so the server can only provide information to the client in response to a request from the client. The most common HTTP requests are GET and POST requests. In each, the recipient

of the request is specifed by a URL. The request can include a number of parameters and values.

When you type a URL to a browser, it performs a GET request. GET requests append the parameters and values to a URL, which means that you can perform your own parameterized GET requests by typing the URLs yourself. Browsers keep the parameters and values of POST requests hidden from site.

HTTP is stateless, which means that there is nothing in HTTP to connect a series of requests together. Web applications that need to maintain state from request to request must implement a mechanism on top of HTTP. Typically, the server will include a special unique session identifier with its first response, and thereafter the client ensures that it includes that unique value with each request. The simplest way to do this is for the server to include a set cookie request with its first response, which will cause the client to send the cookie to the server with each subsequent request. When clients do not enable cookies, the client and server can arrange for the session identifier to be included as a parameter in every request.

3.2 Apache Tomcat

The lecture server makes use of a number of facilities provided by Apache Tomcat. Apache Tomcat is a web server implented in Java. It was originally developed by Sun Microsystems, but was given to the Apache group. Sun continues to provide the specification for the server, and includes a version of the server as part of their Java enterprise software. Sun provides extensive documentation.

Like any web server, Tomcat waits for clients to make HTTP requests, and then responds with some form of content. The response might be the contents of some file, it might be completely computed, or some combination of the two, such as filling in a few values in a template.

All HTTP requests include a URL, which has several components that are used by the web server to determine which content to return. With Tomcat, the host and optional port components of the URL determine which virtual server Tomcat should act like (one server can appear to be many web hosts, and one web host can run on many web servers). Each virtual server has its own configuration, divided into applications, which are associated with the first part of the directory component of the URL. Tomcat applications are deployed (installed) on a virtual server a file called a web archive, or war file.

In addition to its deployed applications, Tomcat web servers also include a number of configuration files, some of which are automatically updated when applications are deployed. The lecture server makes use of the file `context.xml` to provide parameter values for the lecture server, such as the locations for databases, media, etc. This makes it possible for one web archive to be used on multiple Tomcat servers, and for multiple lecture servers (with different lectures) to be used on one server. The `contex.xml` file is also used to define the JDBC resource for the transcriber account database.

A web archive is a packaged directory that describes a web application. When the archive `name.war` is deployed on a server, Tomcat will map `name` in the beginning of a URL directory to the contents of the archive.

The `WEB-INF` directory in a web archive contains configuration information for the web application. The lecture server uses the file `WEB-INF/web.xml` to specify how URLs should be handled. In Tomcat, URLs are handled by *servlets*, which are Java classes that interpret the request and generate the response. The defualt servlet simply maps the request to a file and sends the contents of the file in the response. Other servlets are specified in the `web.xml` file, by associated a servlet name with a Java class, and then associating URL patterns with servlet names.

Most servlets in the Lecture Server are generated from Java Server Pages, which are a combination of static content, script, and Java code. These servlets need to appear in the `web.xml` on the server, but do not appear in the sources because the development environment automatically inserts them. For example, the file `names.jsp` would be converted into a servlet and the URL pattern `*/names.jsp` would automatically be mapped to the servlet.

Tomcat also provides `session management` for the web application. By default, every request to the application looks like a new request. Session management allows the web application and the web client to relate a series of requests, either by setting a cookie (if the client permits it) or by adding a unique session-identifying information to requests embedded in the response. There is no notion of "ending a session" in HTTP, so Tomcat tracks how long it has been since the client last made a request. The `web.xml` file specifies how long session information should be maintained when there are no client requests.

Tomcat provides session variables that applications can use to hold session state. The application can specify a *listener class* in `web.xml` whose methods are invoked when the application starts or ends, and when sessions start and

end. The listener class can be used to initialize session variables.

3.3 Lecture Server Configuration

All Lecture Server application configuration is done in the server's `context.xml` file. The lecture server web archive can be deployed under multiple names (by renaming the file), so the parameters are keyed by the application name. The names take have the format `appName/paramName`. The parameter names are

indexDirectory The `indexDirectory` is the directory that contains the text index and structural index, described in the section on the index. If the actual directory is empty, a new empty index will be created when the server is started.

rpmRoot The `rpmRoot` is a prefix added to every media URL. This makes it easy to change which web server should be used for the media.

wavRoot At one time, we allowed for waveform fragments to be returned. The `wavRoot` specified the directory the should be prefixed to unrooted `.wav` file names to determine the complete pathname for the `.wav` file.

mail.host For demonstration purposes, we have provided simple transcription editing. After an edit, mail is sent about the edit. The `mail.host` is the SMTP server used for sending the mail.

mail.from When mail is sent about an edit, `mail.from` is the name of the sender of the message. This would typically be an administration mailing list.

mail.recipient When mail is sent about an edit, `mail.recpipient` is the name of the receiver of the edit. This would typically be a mailing list of interested parties.

An example use in `context.xml` is show in Figure 14.

3.4 Listener Class

The lecture server uses the class `Listener` as a listener. The `contextInitialized` method is called by Tomcat when the application is started. This method is used to retrieve the configuration information from `context.xml`.

```
<Context>
   ...
  <Parameter
    name="lectures/indexDirectory"
    value="/scratch/segindex"
    description="The location of the lecture index" />
  <Parameter
    name="lectures/rpmRoot"
    value="http://web.sls.csail.mit.edu/lecdata"
    description="Server to use for RealPlayer rpm files" />
  <Parameter
    name="lectures/wavRoot"
    value="/s/lectures"
    description="Root path for .wav files" />
  <Parameter name="lectures/mail.host"
    value="outgoing.csail.mit.edu"
    description="Host to send mail to" />
  <Parameter name="lectures/mail.from"
    value="lecadmin@csail.mit.edu"
    description="Sender to user for sent mail" />
  <Parameter name="lectures/mail.recipient"
    value="lectrans@csail.mit.edu"
    description="Recipient of sent mail" />
  ...
</Context>
```

Figure 14: Fragment of `context.xml`

```
<%@ page contentType="text/javascript; charset=UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
...
<c:url var="lecturesURL" value="lectures.jsp"/>
...
var lecturesURL="${lecturesURL}";
...
```

Figure 15: `urls.jsp` Fragment

The version of Tomcat we are using, Tomcat 5.5, implements version 2.4 of the Servlet specification, which does not provide a way to determine under what name the application was deployed. In other words, it is not possible to determine the prefix to be used for getting parameter values from `context.xml`. However, version 2.5 of the Servlet specification does provide this ability with the `getContextPath` method, and, fortunately, Tomcat 5.5 implements this method, so we are able to invoke the method "by hand."

The values from `context.xml` are stored as attributes in the servlet context, which holds the application state, so that they can be accessed by servlets. The mail-related parameters are stored in a `Properties` object in the form needed by a mail class library.

The listener also implements the `contextDestroyed` method to cleanly close the lecture index when the application is shut down.

3.5 Initial Request

The initial request is sent to `index.html`, which is returned verbatim to the client. The `index.html` includes references to a style sheet and a number of static script files described in the section with the browser.

There is one dynamically generated script file, `urls.jsp`. As mentioned previously, Tomcat relates request to sessions either with cookies or with URLs. The file `urls.jsp` defines JavaScript variables for all the URLs that the client will use to make requests. Figure 15 shows a fragment of the file. The `<%@...%>` lines are directives to the compiler. The `c:url .../>` line defines a JSP variable `lecturesURL` that will contain the URL for `lectures.jsp` with session identification appended if the browser will

```
<%@ page language="java" contentType="text/xml; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
<%-- index gets initialized by the browser.Listener --%>
<% try {%>
<c:set var="qe" value="${index.query}"/>
<categories>
  <c:forEach var="category" items="${qe.categories}">
    <category name="${fn:escapeXml(category.name)}"
              categoryid="${fn:escapeXml(category.categoryid)}"/>
  </c:forEach>
</categories>
<%} finally {%>
    <jsp:setProperty name="qe" property="closed" value="true"/>
<%} %>
```

Figure 16: `category.jsp`

not accept cookies. Later, an ordinary JavaScript variable `lecturesURL` is defined and initialized to the value of the JSP variable (which happens to have the same name). As long as the client makes all requests through URLS from variables defined in `urls.jsp`, the session information will be provided.

3.6 Queries

Once the client has retrieved the initial page, it will make query requests in response to user queries. All lecture query results are returned as XML. Although the client only uses HTTP POST requests, HTTP GET requests are also handled, and can be useful for debugging and experimentation.

3.6.1 Category Queries

The category query has no parameters, and is used to fill in the list of lecture categories. Figure 16 shows the actual contents of `category.jsp`, which generates the list of categories. You can get a general idea of how the list of categories is generated from comparing Figures 16 and 1. The rest of this

section describes gives a more complete description.

The first line tells the JSP compiler about the page. The `contentType` is `text/xml; charset=UTF-8`, which is set by the server in the headers of the response, so that the client knows that the response is XML and the characters are encoded as UTF-8.

The next two lines tell the JSP that two tag libraries are used. Tags whose name begin with `c:` refer to the "core" library, while those whose names begin with `fn:` refer to the "functions" library.

The next line is a comment. As noted in the comment, the Listener class initializes context parameters from the `context.xml` file when the application starts. One of the values it initializes in `index`, which provides access to the class that manages the index.

The `index` a Java Object accessed through "Java Beans," which provides the most convenient way to expose objects to the JSP scripting language through "Java Beans." Java beans lets a Java object expose "properties" by defining public methods for accessing the property. For example, to expose a property called `height`, the object would need `getHeight` and `setHeight` methods. The scripting language does not seem to support calling methods on objects, so things that would be more natural to do with a method had to be done as though they were properties.

All queries must take place in a *transaction*. Databases use transactions to ensure that clients see consistent views of the data. For read operations, such as this, the transaction acts as though it takes a snapshot of the entire database, so that if any changes are made outside of the transaction (such as adding lectures or modifying transcriptions) they will be isolated from this transaction, so that this transaction does not see the data in an inconsistent state. In a write transaction, if anything goes wrong during the transaction, the database arranges for all changes made during the write to be undone, again preserving the integrity of the data.

The `query` property of the `index` bean is an example of a property that is more like a method. Accessing the `query` creates a new query bean with its own new transaction. By convention, the Lecture Server stores the query object in the `qe` JSP variable. When a transaction is started, it is important that it always be finished; if not, bad things will happen, like the database will stop working or get very slow after a while. Thus, the transaction must be wrapped in a Java `try/finally`. The `<% ... %>` notation is used to surround arbitrary Java code. Here we use it to add the `try/finally` form that ensures that the transaction which will be started by `index.query` al-

```
<c:if test="${!empty param.query}">
  <jsp:setProperty name="qe" property="query" value="${param.query}"/>
</c:if>
```

Figure 17: Setting a Query Parameter

ways gets finished at the bottom of the file by setting the `closed` property of `qe` to `true`.

The `<c:set .../>` notation is used to set a JSP variable to a value, in this case the result of runnign the script `index.query`, which is invoked because it is inside of curly braces preceded by a $. As noted above, the $< c : set... >$ tag is defined by the "core" Java server tag library, whose functionality is included at the top of the file, where the functionality is also associated with `c:`.

The `<category>` tag has not been defined to do anything special, so it just gets sent in the response, as is it's close tage, `</category>` a few lines later. Since this is essentially raw text, nothing in JSP prevents you from forgetting a close tag or mispelling a tag name.

The query bean, `qe`, has a property which is a list of beans for the categories. We need to generate one XML tag for each category, so we use the JSP tag `fn:forEach` which iterates through a list, setting its variable, `category`, to each item in the list.

The category bean has two properties that we are interested in, the `name` and the `categoryid`. Each of these could have characters that need to be specially quoted in XML, so we use `fn:escapeXml` to ensure that this happens.

3.6.2 Lecture Queries

The most common queries are lecture queries, which use the `lectures.jsp` URL. Unlike category queries, lecture queries have a number of parameters. The basic outline of `lectures.jsp` is the same as `categories.jsp`, with the `try/finally` to ensure the transaction is closed. Figure 17 shows how a query bean property is set to the value of an optional request parameter. JSP will expose all the request parameters as properties of the `param` bean, and missing properties will be "empty." When the query bean is created, the properties are all initialized to reasonable defaults, so we only set the query

```
<Resource
 name="jdbc/TrantestDB"
 auth="Container"
 type="javax.sql.DataSource"
 username="DBUSER"
 password="DBPWD"
 driverClassName="org.postgresql.Driver"
 url="jdbc:postgresql://DBHOST/DBNAME"
 maxActive="8"
 maxIdle="4"
 removeAbandoned="true"
 logAbandoned="true" />
```

Figure 18: User Database Information

bean properties for those parameters that were set in the query.

3.7 User Login

The `login.jsp` page concatenates the `nonce` it sent to the client with the base-64 string of the hashed `salt` and password, computes an MD5 sum, and converts it to a base-64 string. If this matches the `clientHash` then the server sets session parameters to indicate the pemitted roles for the user. All access control is controlled by the session parameters in the server, never by what the client claims it is allowed to do.

The user account information is stored in an SQL database, which must be described in the server's `context.xml` file in a `Resource` tag, as shown in Figure 18. The values `DBUSER` and `DBPWD` should be replaced with the database user name and password, which is on the host `DBHOST` and named `DBNAME` on that host.

4 Bulk Population

After briefly describing the process of automatically transcribing lectures, this section describes how information about the lectures is prepared for indexing, and how the indexing is accomplished.

```
<?xml version="1.0" encoding="UTF-8"?>
<document fileName="...">
<lecture title="NA" keywords="combination equation...">
<segment id="1" title="equation y zero minus ...">
6323 6534 hi
6768 6973 this
6973 7138 is
7138 7258 the
...
</segment>
<segment id="2" title="minus ...">
...
</segment>
</lecture>
</document>
```

Figure 19: Excerpt of Segmented Transcription

4.1 Getting Transcriptions

We receive lecture videos in RealPlayer format. The audio portion of the videos are extracted manually to produce an accompanying RIFF format 16KHz audio `.wav` file. These audio files are automatically transcribed and then automatically segmented to produce a "`.seg`" file, as shown in Figure 19. The file consists of one section for each segment of the lecture, and, within each segment, triples of start time, end time, and text for each word.

4.2 Preparation for Indexing

A separate XML file must be manually prepared for a group of lectures to be indexed. This is a tedious process and could be automated by better organization and integration of the tools, plus a layer of infrastructure for managing the lecture meta-information.

There are two types of descriptions, those for seminars, and those for courses. One file can consists entirely of seminars, or of all the lectures in a course.

Figure 20 shows an excert from a file describing seminars. The values of

```
<?xml version='1.0'?>
<seminars
 group="mitworld" institution = "MIT"
 media_root="12800/rm/mitworld.download.akamai.com/12800/"
 wav_root="MIT-World">
 <seminarseries
  series = "Back to the Classroom" host = "MIT Sloan School">
  <lecture
   title="The Future of Work" lecturer="Thomas Malone"
   date = "June 5, 2004"
   media="mitw-sloan-bttc-04-malone-work-05jun2004-220k.rm"
   segfile="MIT-World-2004-Sloan-BTTC-Malone-Work-05Jun2004.seg"
   timescale = "1.0">
   <category>Business and Economics</category>
   <category>Technology and Innovation</category>
  </lecture>
  <lecture
   title="Innovation: Are You A Predator or Are You Prey?"
   lecturer="James Utterback" date = "June 7, 2003"
   media="mitw-sloan-backtoclass-utterback-07jun03-220k.rm"
   segfile="MIT-World-2003-Sloan-BackToClass-Utterback-01Jun2003.seg"
   timescale = "1.0">
   <category>Business and Economics</category>
   <category>Technology and Innovation</category>
  </lecture>
 </seminarseries>
 <lecture
  title="Software Breakthroughs: Solving the Toughest..."
  lecturer="Bill Gates" date = "February 26, 2004"
  media="mitw-eecs-bill-gates-microsoft-26feb2004-220k.rm"
  host = "MIT Electrical Engineering and Computer Science Department"
  segfile="MIT-World-2004-EECS-Bill-Gates-Microsoft-26Feb2004.seg"
  timescale = "1.0">
  <category>Technology and Innovation</category>
  <category>Engineering</category>
 </lecture>
</seminars>
```

Figure 20: Excerpt of Description of Seminar Lectures

the attributes in the `seminars` tag are shared by all the child nodes. There is one seminar series with two lectures, and one lecture that is not part of a series. Each of the lectures is included in two categories. The complete list of categories is derived from the set of all categories, so a spelling mistake will result in a new category.

The `media_root` and `media` attributes are concatenated to form an unrooted path for the media. As mentioned in the server configuration section, a configuration parameter for the web application supplies the root for the media pathnames. The segmentation is provided as an unrooted file name. A parameter to the indexing program supplies the remainder of this pathname. This makes it easier to set up a lecture server on another file system, such as a stand-alone Windows version.

The `timescale` attribute is a work-around for a problem that sometimes occurs during the waveform extraction. In some cases, for unknown reasons, there will be a slight linear skew between the extracted waveform sample rate and the video sample rate, which causes the transcription to be off by several seconds at the end of a lecture. The `timescale` parameter is a compensation mechanism.

Figure 21 shows the description for some lectures in a course. The description is similar to the one for seminar series, but tailored towards the lectures in a course.

4.3 Indexing

The Java program `segindexer.jar` is used to index lectures. It takes arguments of an index directory, a root directory for segmentation files, and zero or more XML files that describe lectures. If the index directory does not contain an index, a new one is created. Otherwise, the lectures are added to the existing index. Figure 22 contains a script for indexing three groups of lectures.

5 Lecture Browser

This section describes the Lecture Browser. The Lecture Browser makes heavy use of JavaScript, AJAX, dynamic HTML, and cascading style sheets.

The initial page is `index.html`. All successive operations locally modify the HTML of the initial page, so the browser never leaves the initial URL.

```
<?xml version='1.0'?>
<course
 group="ocw" institution = "MIT" department = "Mathematics"
 number = "18.06" name = "Linear Algebra" year = "1999"
 media_root="7870/rm/mitworld.download.akamai.com/7870/"
 wav_root="18.06-1999">
 <lecture
  number="1" lecturer="Gilbert Strang"
  title="The Geometry of Linear Equations"
  media="18/18.06/videolectures/strang-1806-lec01-26aug1999-220k.rm"
  segfile="18.06-1999-L01.seg"
  timescale = "1.0">
  <category>Mathematics</category>
  <category>Linear Algebra</category>
 </lecture>
 <lecture
  number="2" lecturer="Gilbert Strang"
  title="Elimination with Matrices"
  media="18/18.06/videolectures/strang-1806-lec02-10sep1999-220k.rm"
  segfile="18.06-1999-L02.seg"
  timescale = "1.0">
  <category>Mathematics</category>
  <category>Linear Algebra</category>
 </lecture>
 ...
</course>
```

Figure 21: Excerpt of Description of Course Lectures

```
#!/bin/bash
indexer=~/workspace/segindexer.jar
index=/scratch/segindex
segroot=/s/lectures/mit-museum/transcripts/
xmlroot=/s/lectures/mit-museum/xml-info

cd $xmlroot
for f in Intro.xml 18.06-1999.info.xml 18.085-2001.info.xml ...
do
  echo $f
  java -jar $indexer -add $index -segroot $segroot $f
done
```

Figure 22: Bash Script for Indexing

Instead, event handlers send requests to the server, which returns XML. The XML is interpreted and the results are displayed by making changes to the displayed HTML.

The JavaScript is divided into five files, roughly corresponding to functionality:

ajax.js Manages communication with the server.

db.js Manages everything not handled elsewhere.

karaoke.js Manages the synchronization of played media and aligned transcriptions.

md5.js Computes an MD5 sum.

urls.jsp Contains variables for the URLs needed by requests. When the browser does not support cookies, these URLs will include the session identifier.

5.1 JavaScript Basics

JavaScript is a browser scripting language that is supported by most web browsers. In spite of the similarity in names, JavaScript has nothing to do with the programming language "Java," although the syntax is similar to

```
function makeCounter(){
  var count = 0
  return function(){
    return count++;
  };
}

var c1 = makeCounter()
var c2 = makeCounter()
```

Figure 23: Counter as a Function

that of Java and C. Much of JavaScript browser programming consists of creating objects and functions that get called in response to various events, such as mouse movement, button clicks, and timers.

In JavaScript, most things are objects, which means maps between property names and objects. The dot operator is used to get or set property values, as in `window.height` or `windows.height = 100`.

The operator `var` adds a variable to the current naming context, which is either global or the enclosing function. Even though a `var` can appear in what looks like scope-limiting curly braces, the variable scope will still be that of the enclosing function.

Unlike C, function definitions can appear inside of functions, and variable bindings of the outer function remain in effect. In Figure 23, the function `makeCounter` initializes a new copy of the variable `count` to 0 each time it is called. It then returns a function that has access to the variable `count`. Each time the returned function is called, it returns the current value of its `count` variable, and increments it. The global variables `c1` and `c2` are initialized to two counter functions. Calling `c1()`, `c1()`, `c2()`, and then `c1()` will return the values 0, 1, 0, 2.

A counter could also be created with an *object*, which is a container for associating properties and values. New objects are created by the operator `new`, as in `new Counter()`. This will call the function `Container`, called the *constructor*, which will have an implicit `this` variable. The constructor can initialize properties of `this`.

Every constructor has an associated `prototype`, accessible on the `prototype` property. All objects created by the constructor will share the properties of

```
function Counter(){
  this.count = 0;
}

new Counter();

Counter.prototype.next = function(){
  return this.count++;
}

var d1 = new Counter();
var d2 = new Counter();
```

Figure 24: Counter as an Object

the prototype, so the prototype is where methods should be placed. Some older versions of JavaScript to not initialize the prototype property until the constructor has been called once, so it is safest to call the constructor before defining methods.

In Figure 24, a `Counter` object is defined. The sequence of method calls `d1.next()`, `d1.next()`, `d2.next()`, `d1.next()` will return 0,1, 0, 2.

5.2 Ajax

In the late 1990s, Microsoft added a script-accessible object to their browser that allowed scripts to access the web server and receive the results. Soon afterwards, the other major browsers added similar functionality. At about the same time, Microsoft introduced dynamic HTML, which let the scripts running in a browser modify the HTML. Although this functionality was later standardized by W3C, Microsoft never adopted the standard, although, for compatibility, other browsers support subsets of Microsoft's implementation.

The support for AJAX requests is in the file `ajax.js`.

To send a request to a web server, the browser must first obtain the object that handles this functionality; there is one way to do this for Internet Explorer that assumes functionality available only on Windows, and another way for every other browser. Once the object is obtained, a request to a URL is opened, request parameters are set, and event handler for the response is

set, and the request is sent. When the response is received from the server, the event handler function is called and can obtain the results as parsed XML.

While the request is being processed by the server, the browser is free to handle other user requests, which could result in other requests. A server can process multiple requests in parallel, so a response for a later request could arrive before a response to an earlier requrest, which can cause problems if the browser is not expecting the responses out of order.

The script in `ajax.js` avoids this problem by implementing an operation queue. Requests are placed in the queue and acted upon serially in the order they were submittes. Optionally, a function can be called while when the operation starts and stops. For example, input can be disabled while waiting for a query.

5.3 General Display

The file `db.js` handles most user interaction. Although the Lecture Browser works on several browsers, the display is somewhat fragile, and small changes that look correct in one browser may have problems in a different version of the browser, or another browser. When making changes, frequent testing on multiple browsers is important since problems can be difficult to debug.

There are two helper functions, `println` and `printlnc`, that can be used during debugging. `println` will print a string on a "debug" pane, while `printlnc` will print the string after first clearing the debug pane.

5.4 Media and Transcripts

Most of the support for media and transcripts is in the file `karoke.js`.

A timer is used to periodically monitor the RealPlayer player object. When someone clicks on one of its buttons, it undergoes a state change which is detected by the timer function. During play, the probe rate is increased since the player position within the utterance is polled.

During play, the browser makes times requests to the server to seek time-aligned transcript information. The browser tries to stay enough ahead of the player so that the transcription of what is being played is always available. Since the user can move around through the lecture, data might not be received in order. As the data is received, it is merged into a two-level tree that is used during play to highlight words as they are being played.

Received transctiption information is associated with `span` elements. Highlighting is accomplished by changing the style of the `span` while it is played. The `span`s also have event handlers to start play at their location if they are clicked on.

6 Lecture Index

The lecture index is a Java class library that encapsulates the management of all lecture information. The index uses two databases, Berkeley DB, Java Edition, for lecture meta-information, and the Apache Lucene text indexer for text. The index is also responsible for parsing segmentation files. The index also implements the Java beans used by the server, and provides the interface for indexing new lectures.

6.1 Meta-Information Index

The package `edu.mit.csail.sls.lectures.index` holds all the classes related to meta-information, Java beans, querying, and indexing. Users of the index access the text index indirectly through the meta-information index.

6.1.1 Accessing

The main index class is `Index`, which can be used directly by programs like the bulk indexer. A secondary class, `LectureIndex`, is better-suited towards web applications, since it supports a bean-like interface for starting and ending transactions and performing a few other web tasks. The listener class for the Lecture Server creates a `LectureIndex`.

The `Index` must be initialized with the directory that holds the index. The index will have two subdirectories, `lucene` and `file`, the first being for the Lucene text index, the second for the lecture meta-information.

6.1.2 Berkeley DB

Berkeley DB Java Edition provides data storage at a lower level than that provided by relational database. Databases are a persistent mapping between keys and values, each of which is a variable-length byte-vector. The keys are kept in a tree, so the data can be traversed in key order.

In older versions of Berkeley DB, such as the version available when the Lecture Server work started using Berkeley DB, applications were responsible for packing and unpacking the keys and values themselves. In the current version, Java class definitions can be annotated so as to automate the packing and unpacking. Unfortunately, this requires some retrofitting, but fortunately the Eclipse Java development environment greatly eases this process, so some of the old-style packing/unpacking has been removed. In particular, the class `DataAccess` manages the categories and courses, which use the new style, while the remaining data uses the old style.

One of the differences between Berkeley DB and an relational database is that you are somewhat on your own for joins. For example, in a relational database, you would use a join to combine data from a combination of categories and lectures. In Berkeley DB, you need to maintain a separate table. Some of this could be simplified by using the newer facilities.

The database supports transactions. When a program makes changes to the data, such as adding a lecture, the data can be in an inconsistent state. For example, a lecture may not have all its segments. If another program were to start looking at data, the inconsistent state would cause problems. Transactions let the updater and reader remain isolated. The updater and reader each start their own transactions. The reader will not see any changes made to the database since the beginning of its transaction, and the updater will only see its own changes. If something goes wrong with the update, all the updaters changes are thrown away. If all goes well, the updater commits, and any future transactions will see all the changes.

6.1.3 Data Classes

Raw data, corresponding to a database record, has an unadorned class name, such as `Seg`. In the old style database use, there will also be a class with `DB` appended, as in `SegDB`. This class handles packing and unpacking, as well as key generation. There is often another class, with `Hit` appended, which is a handle to retrieved data. The class `SegHit` is an example. Where the raw data might have an identifier, the hit would have a pointer to the identified object, or an iterator for a group of objects. The raw class and the `-Hit` classes are Java bean classes.

There are also class names corresponding to two data classes concatenated, as in `CategoryLecture`. These provide the data for the join operation alluded to previously. They may also incude `-DB` versions, as well as

`-Iterator` versions which are iterator interfaces.

6.1.4 Queries

The class `QueryBean` provides "higher-level" access to queries than those provided by the `Index` class through a bean interface. In general, queries should be performed with the `QueryBean`, since it encapsulates the `Index` implementation. A `QueryBean` can be obtained with the `Index` method `getQueryBean()`.

6.1.5 Updates

The class `IndexWriter` provides the interface for programs that need to add lectures. An `IndexWriter` is obtained with the `getIndexWriter()` method of `Index`.

6.2 Text index

The `edu.mit.csail.sls.lectures.trans` uses Apache Lucene to manage the text index. The `TransIndexer` class adds data to the index, while the `TransQuery` class retrieves data.

7 User Index

The user index manages the user account information, such as roles, salt, and hashed passwords. The data is stored in a PostgreSQL database which is accessed using JDBC.

The `userdb` project manages the jar, while the `inituserdb` project populates it with a fixed set of users and passwords. The beginnnings of some work to migrate from Berkeley DB to JDBC are also in some of files.

8 Future Work

This section proposes changes to the lecture browser system. These changes fall into several broad categories:

- Adding capabilites for the end user. This includes history mechanisms, such as bookmarks and back-tracking, as well as annotations, the ability

to find related content, support for additional media players, and voice query.

- Simplifying system administration. This includes replacing the tedious hand-editing of XML files describing lectures to be indexed with an easier to use interface, integration of the automatic transcription tools into the system, real support for correcting transcriptions, real support for submitting new lectures, and support for additional media formats.

- Making the system usable in environments outside of the Spoken Language Systems Group at MIT. The lecture browser system is organized as a group of Eclipse projects that require the installation of additional Eclipse plug-ins and various 3rd-party tools and libraries, and the setup procedure needs to be documented. In addition, the Tomcat and Apache configuration needs to be documented.

  The automatic transcription process must be available, either by packaging it (which would involve licensing and compute power), or offering it as a service in a way that could be integrated with the administration process.

The remainder of this section will describe changes in the implementation that are required to support the new capabilities. In many cases, multiple capabilities require the same implementation changes.

8.1 Managing Lecture Information

Changes need to be made in the way lecture information is managed to support several of the proposed changes. This section describes the problems with the current implementation and proposes a new way to manage the information.

8.1.1 Problems

Lecture content and meta-data are currently managed by hand in the filesystem, with the information cached during the indexing process. The server caches this information in the index, and supports adding additional information to the index, and changing cached information, but it cannot propagate

updates beyond the cache, so changes made via the server do not survive index reinitialization.[3]

Attempting to propagate changes from the cache back to the file system would be problematic, since there are no controls on the file system. If the file system is modified by hand, the cached version will not agree. Furthermore, manually editing XML files is error-prone, which could lead to problems when the server tried to merge a change into an invalid file.

A problem that would be created by adding bookmarks is that in any situation where semi-permanent names are created, such as the URLs for bookmarks, the implementation of the content associated with the names should be isolated from the names. In the current system, lectures are identified by their lecure identifier, which is an artifact of the indexing process. If a new lecture is added to the XML files where it fits logically, rather than at the end, then the lecture identifiers will change.

Each lecture is associated with a unique URL for its media, but this is not good long-term identifier since these URLs are derived from the externally maintained layouts of MIT's Open Courseware and MIT World web sites, and could change at any time. Furthermore, the media itself sometimes changes. For example, for copyright reasons, portions of lectures are sometimes blacked-out. We have also seen media editing, which changes the time-alignment.

8.1.2 Modifications

The problems with lecture information can be solved by moving the lecture information that is currently stored in the filesystem and modified by text editors into a managed database that is modified by lecture browser administration programs. The combination of a database and gated access to the information solves the multiple copies, concurrency, and formatting problems associated with the direct editing of files.

The lecture database must be carefully organized to meet the unique needs of the lecture data. The basic bookmarked entity is the lecture, not the media. If someone views a video of a lecture and bookmarks a location, they want the bookmark to correspond to that location in the lecture, not to a particular offset in a particular edit of the lecture in a particular format. For example, if they made the bookmark on a cell-phone browser that was

[3]The index must be reinitialized when underlying libraries change, or when changes to the browser demand additional indexing.

using a low-bandwidth video stream and later wanted to view it on a desktop computer with a high-bandwidth stream, they would like the bookmark to "just work" for their new configuration.

Although the basic bookmarked entity is the lecture, the bookmark needs to include information about the media, or, at least the edit version of the media. When they mark the point where the lecturer says "Here we see..." that is 31:23.242 minutes into the version of the lecture they are watching, they will want playing to start at "Here we see...." three weeks later when an edit to the lecture has changed the time to be 32:46.126 minutes into the media. Since the edits occur independently of the lecture browser system, we need to bookmark with time positions in a version of the media, and then realize the bookmark by converting it to a semantic position and finding that semantic position in the new version of media.

There should be an entity that corresponds to a lecture, and most metadata should be associated with that entity. Each lecture must have a unique *permanent* identifier, generated when the administrator creates an entry for the lecture.

Media is referenced in the database by parameterizable URL expressions. Since lecture browser administrators do not know when lectures change, a group of changes to the lecture database is assigned a generation number. When a media URL is first associated with a lecture, a unique identifier for the media is generated, and associated with the URL expression, the current generation, and a hash of the media (such as an MD5 sum). When a new set of media, possibly the same, is installed, the hash of the new media is recomputed, and, if it is different, the media has changed. In this case, a new media identifier is generated and associated with the URL expression, the current generation, and the new hash. The lecture is flagged as needing to be re-transcribed. It is assumed that if several media files for a lecture change, they all correspond to the same media edit, i.e. they are time-aligned with each other.

Lecture transcriptions are associated with the lecture, media generation, and a transcription specification, since all of these effect the results of the transcription. A transcription specification is parameterized by things like the speaker, auxillary data, language model, and the version of the transcription process. This makes it possible to experiment with changes to the transcription tools and analyze the results, without needing to worry about a change hurting browser users (since the indexing can be configured to not show the experimental transcriptions).

Every automatically transcribable lecture needs a media file for the waveform; when a video generation changes, a new wave file must be extracted or provided that corresponds to the new generation.

8.1.3 Media Refresh

We receive media from MIT Open Courseware and MIT World either on a disk as a copy of the web site, or from the actual web site. As lectures are added or media edited, this information changes, but we do not know when the changes occur. The term "media refresh" refers to a scan of the web site to find new lectures and changed media.

After a media refresh, the administrator would be presented with a list of all the detected media and anything already known about it. They could choose to ignore, transcribe, or update each potential lecture. For example, if the Open Courseware web site dump had been reorganized, this would make it relatively easy for the administrator to find the new locations of the media, as well as to identify new media.

After new media is transcribed, it could be indexed. The index could also be searched for similar media, so as to identify lectures whose media may have been edited slightly. Dynamic programming could be used to provide to provide a map from time offsets in the old media to those in the new media, so that existing user bookmarks and other annotations could be preserved.

Since metadata is separated from the index, the index can be recreated from the metadata at any time. This provides a number of benefits: The same metadata can have more than one index, so that index development could safely share the same lecture data as a deployed system, or so that multiple copies of the index could be maintained to be used by multiple servers to achieve better scaling.

8.2 Transcription

Normally, new lectures are automatically transcribed. We expect that some users will want to manually correct parts of the automatic transcriptions by replacing a portion of the transcription with their own transcription. The new transcription would be time-aligned (forced-alignment) with just the portion of the lecture whose transcription was replaced.

Since we also continue to work on improving the automatic transcriptions, we want to be able to update the automatic transcriptions associated with lectures. If users have edited or in other ways annotated a transcription or lecture, it is important that their work is not lost during a retranscription. Much as with source control systems, there needs to be a process in which they can refresh changes committed by other users or by an improved automatic transcription. Furthermore, correct hand-transcriptions provide valuable statistical information about the types of mistakes made by the automatic transcription process, and could be used both for evaluation and training puroposes.

8.3 Browser Changes

The user visible enhancements require changes to the JavaScript that runs in the browser. Some changes, such as making the "Back" and "Forward" butttons and history work are difficult or impossible because browsers do not currently support this functionality for AJAX applications.[4] The browser buttons and history can be made to work by making more requests go to the server for HTML, and by encoding the browser state in the URLs. This would also simplify bookmarks. There would be a slight performance penalty, but it would probably not be noticeable since we already need enough bandwidth for the videos.

8.4 Documentation

- Setting up the Eclipse environment.
- Setting up Tomcat.
- Setting up Apache.
- Setting up PostgreSQL database.

[4]It is easy to find sample code that works in one or two browsers for simple cases by taking advantage of anomalies in various browsers, but none of them have worked in more complex scenarios.

8.5 Query Results

Query results do not work properly since any words mentioned in the query counts as a hits within the lecture. For example, if "gamma ray" were searched for, only lectures containing "gamma ray" would be returned, but, in the lecture, in addition to returning "The gamma rays were deadly" fragments like "The manta ray disturbed the detector." and "Gamma was 3.4 in this test" would also be shown.

Lucene allows a text stream to be tokenized and broken into fragments that are then filtered by a query. To make use of this capability, we need to be able to preserve the timing information through the filtering.

One way to do this would be to index words by character text position in the document since tokens can include this information. Alternatively, the document could be pre-fragmented, and fragments could be indexed by text character position. The highlighter would identify the fragments, and the character position of the start token in the fragment would link back to the timing information. Pre-fragmenting would also allow for statistical fragmenting based on content, rather than simply looking for timing gaps.