INF-3981
Master's Thesis in
Computer Science

Improving Freshness of Web-Extracted Metadata

by

Tord-Arne Heimdal

December, 18th, 2009

Faculty of Science & Technology
Department of Computer Science
University of Tromsø

Abstract

Live video search is emerging as a platform for multimedia production and entertainment service. Such systems rely on a stream of live video and metadata describing the video content. A high quality source for such metadata can be found on the web. Identifying and extracting metadata from web pages can be done by crawling and scraping. However, general crawler politeness rules limit per-site polling frequency, and therefore the freshness of the retrieved data is also limited.

In this thesis we present a metadata extraction system capable of combining high metadata freshness, while at the same time adhering to polling politeness rules. To achieve this, the proposed solution uses a pool of web sources containing overlapping information scheduled in a round-robin fashion. Our experiments and analysis show that our system is capable of keeping the average metadata freshness higher than any single-source solution, while at the same time adhere to polling politeness rules.

Acknowledgements

First, I would like to thank Håvard D. Johansen, my supervisor, for letting med work with him on this thesis. Also, his valuable feedback, support, and knowledge about scientific work has been much appreciated.

Thanks to Professor Dag Johansen, my co-adviser, for valuable ideas and epidemic enthusiasm, and to all the members of the iAD project.

Thanks to my fellow students and friends during the last five years, Børge Jakobsen, Robin Pedersen, and Joakim Simonsson.

Also, thanks to the technical staff at the Computer Science Department, for support, and to Jan Fuglesteg for all help during my student years.

Special thanks to my family & friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Live video search is emerging as a platform for multimedia production and entertainment services [15]. Such systems rely on a stream of live video and metadata describing the video content. This allows composition of personalized videos that can be played out as one continuous stream on-the-fly. However, the quality of these services depends on how fast the video can be made searchable and presented to the user. Therefore, the rate at which the system gains access to *fresh* metadata is very important.

A key input to such systems are text-based metadata that describes the video content minute for minute. Such metadata can be generated in many ways, ranging from automatic extraction by analyzing the live video with different techniques like audio-to-text conversion [29, 19] and feature detectors, to manual human generated annotations. The precision and recall numbers possible to attain by using automatically generated metadata will however vary according to the tool used, and even the best tools available are not able to detect all important events in a video stream with audio included. As an example, if we look at a soccer video containing an audio track with commenting, audio-to-text generation of metadata is possible. However, it is ineffective on audio tracks that contains little or unclear commentary speech, resulting in only low-quality metadata. We also have closed captioning, which is a direct transcript from speech to text, but it will often contain information that is irrelevant to the game, and it also lacks a well defined structure. Another important aspect, is that most of the automatic extraction techniques are very cpu demanding, and might take longer time to execute than the video itself. Therefore, the performance of such tools are not able to provide a live search system with data that is accurate enough, and at a frequency rate that is acceptable.

Humans are generally good at analyzing complex video data in real time. However, the process of manually annotating videos is often time consuming and tedious. Fortunately companies are willing to invest human resources for this task. As such, there exists a large pool of human generated, semi-

structured and live updated information available on the web. For instance, several news sites provides live comments for soccer matches with important event information published on a minute-to-minute basis. Examples of such sites are *Sky Sports* [1], *Live Goals* [2] and *Yahoo! Sports*[3]. Because of the high update frequency, and good accuracy, this metadata is well suited for indexing a live video stream.

1.1 Problem Definition

Although the high quality metadata is readily available, it is generally only published as Hypertext Markup Language (HTML) data on Internet web servers. Such data can be extracted automatically using existing technologies like *crawling* and *scraping*. This can be done by launching a site specific crawler identifying a specific web page containing the data we want to extract. And then we create a site specific *scraper* capable of identifying and retrieving the wanted data. However, general crawler politeness rules limit per-site polling frequency, and therefore the freshness of the retrieved data is also limited. This is particularly limiting on the freshness in scenarios where multiple pages per-site must be monitored for updates.

> *This thesis shall study the problem of identifying and extracting video metadata from web sources for the purpose of feeding a live system with fresh data.*

The goal is to construct a prototype metadata extraction system, that can combine hight metadata freshness while at the same time adhering to polling politeness rules.

1.2 Scope and Limitations

This thesis will use the soccer domain whenever there is need for data or concrete examples. Our thesis will not be focusing on a complete implementation of the proposed system design for live metadata gathering. Rather we will implement the parts that are necessary for performing our *freshness* and *completeness* analysis.

1.3 Method and Approach

The report from the ACM task Force on Core of Computer Science has divided computer science into three major paradigms [27].

[1] www.skysports.com
[2] www.livegoals.com
[3] www.eurosport.yahoo.com

- `Abstraction` where scientists uses an already deployed model, system or algorithm to simulate a process. In this approach, progress comes from testing, studying, and analyzing the simulation.
- `Theory` is the approach where the scientists tries to understand the underlying mathematical ideas. He poses theorems and seeks to prove them in order to find relationships.
- `Design` is the third approach where scientists tries to use their knowledge to build a solution after formulating a problem. By working systematically, testing, and comparing results, the engineer seeks to find the best solution to a problem.

Our main approach have been on the design, because we will design and prototype a system capable of extracting fresh metadata from web sources. However, we have also worked within the abstraction domain, in order to analyze our system design and implementation.

1.4 Outline

This section have described the background, and defined the problem and scope of this thesis. The rest of the thesis is organized as follows. Chapter 2 contains a thorough analysis of three football metadata sources, introduces the area of web extraction and related work in that field. Chapter 3 describes our system design and implementation, followed by Chapter 4, where we describe our experiments and evaluate them. Finally, in Chapter 5, we conclude our work, findings, and outline future work.

Chapter 2

Background and Related Work

In this chapter we first give a thorough analysis of three web sites providing live soccer commenting features. Then we introduce *web data* and how we can exploit the semi-structured features of web data when *mining* for structured data. After the introduction, we give a brief overview of the two fields *Crawling the web* and *Scraping the web*.

2.1 Football Metadata Sources

In this section we compare the information available at three different sites that provides information about Premier League football matches. The sites we analyzed were *Sky Sports* [1], *Live Goals* [2] and *Yahoo! Sports*[3].

The survey will give a general description of each site and describe the general layout. Then it will dig deeper into the information and content available at each one. In this setting the content described will be information that is residing in some kind of HTML structure that is common for each football match described, and therefore is possible to detect and retrieve as structured information.

In addition to content description, we will give a brief description of the HTML structures that each match description page is built upon.

2.1.1 Yahoo! Sports

Yahoo! Sports web site is a collaboration between the american internet information company *Yahoo! Inc.* and the company behind europe's leading sports multimedia platform, the France based company *Eurosport Group*.

[1] `www.skysports.com`
[2] `www.livegoals.com`
[3] `www.eurosport.yahoo.com`

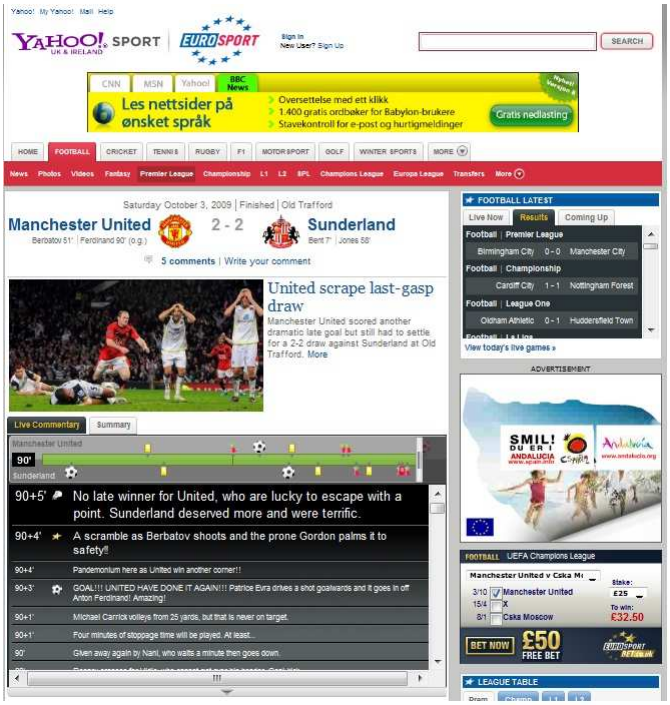

Figure 2.1: Overview Yahoo! Eurosport

They united in 2007 to create a major new online experience for sports fans in UK & Ireland, Spain, Italy and Germany. Their vision [4] is to unite Eurosport's high quality editorial content with Yahoo!'s social media platform, to create a winning combination for sport fans. Their site contains information about several sports, including football, cricket, tennis, rugby, formula one, golf and several winter sports.

Figure 2.1 is a screenshot of the site displaying a football match between Manchester United and Sunderland. As can be seen in the figure, the layout consists of a site header with advertisement, general information, searching, and navigation. The horizontal navigation bar allowing the user to navigate from any page to any major sport. The right center side of the site is used for advertisement, and displaying dynamic information relevant to the sport category you are browsing. In the bottom left corner of the screenshot in Figure 2.1, the information displayed is live scoring update from ongoing matches and the league tables for the four top leagues in England. The left center side of the site is used for displaying the main content, in this case the football match page. The bottom section of the site contains copyright information, and links to more information about the site. In addition there is a navigation box with links to all the sports that the site is writing about,

[4] `http://help.yahoo.com/l/uk/yahoo/sport/general/what_is.html`

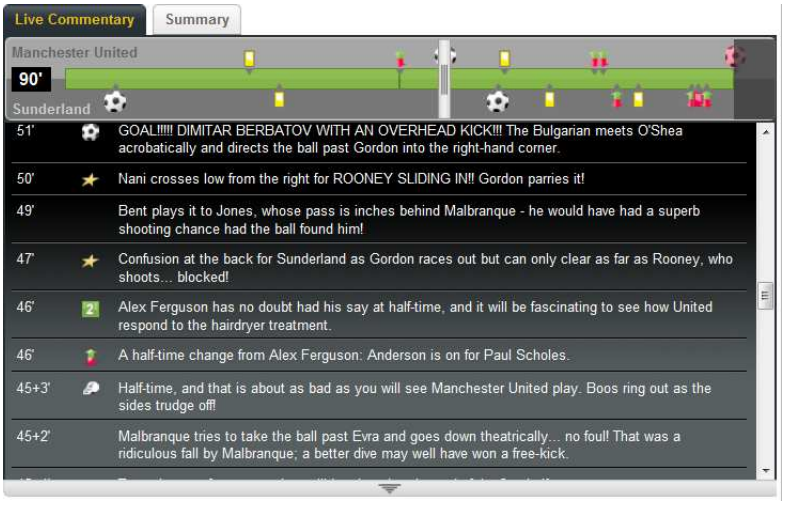

Figure 2.2: Yahoo! Slider Feature

and information about other online services Yahoo! offers. Their End User License Agreement [5] (EULA) states among other things that the site content can only be used for non-commercial purposes.

2.1.1.1 Content

The available content can be classified as either match facts or match events. Examples of match facts are the name of the teams playing and name of the stadium where the match is played. Events are happenings occurring during the match that can be timestamped. For instance, a goal is scored or the referee blows the whistle for half time. Some content lands in the middle. For instance, the match result is changed dynamically when a goal event occurs. However, this information will also be detected when monitoring new events.

At this site the match facts available are: home- and away teams, result, captains of both teams, the match date, lineups, managers, match stadium, injuries, suspensions, and match status. The lineup includes information about available substitutes. Injuries are players not available to play on each team, because of injuries in front of the game. Suspensions are players not available because they have received to many bookings earlier in the season, and are therefore illegible for the match at hand. Match status identifies if the match is not started, started or finished.

The site also has a few features handling events during the game. For instance, a live event description feature, see figure 2.1 and 2.2. In this live feature the user is live updated with detailed game events like goals,

[5] http://uk.docs.yahoo.com/info/terms.html

bookings, substitutions, time added, and other interesting events occurring during the game. Each comment is timestamped with minute granularity, and events happening in overtime are timestamped by adding how many minutes over 90 minutes the event happened (e.g. 90+5'). Because of the rather rough granularity, some events will have the same timestamp, and the only ordering available is then which event were posted first. Each major event is also accompanied by an illustrating icon in the comment. For instance, a comment describing a goal will be accompanied by a football icon, and a tackle resulting in a yellow card will be accompanied by a yellow-card icon. There is also a nice feature, see Figure 2.2, allowing the user to select events on a timeline, or use a slider for looking up a specific comment describing an important match event. When an event is selected, the live commentary list is scrolled to the selected point in time and the comment can easily be read. In addition to the live commentary, there is also available a *summary tab*, and when it is selected, only the major events like goals, red cards and big chances are displayed in the commentary.

Another feature is live updating of the two teams, lineups by using a football field image with the names of the starting players distributed across the field at their playing positions. When, for instance a substitution is made, the player on the field is marked with a red arrow illustrating that he has left the field. In addition there is a table underneath the field, where the available substitutions are. When a player is sent on the field, this is marked by a green arrow and a timestamp stating when the substitution was made. Other events like goals and bookings are also marked on the field by adding a football- or card-icon above the name of the player involved.

After the match has been played there is also added a link and introduction to a comprehensive match article describing the game. When it comes to user generated information, registered users are allowed to leave comments on the match page. Registration is free. It is also possible to comment on the match article, and rate the article by clicking a *buzz up* icon.

2.1.1.2 Data

All the pages on the site describing premier league football matches have the same basic Hypertext Markup Language (HTML) structure. General match information, like home- and away team, match date, and status, is placed on top in a table like fashion by the use of HTML `<div>` tags and Cascading Style Sheets (CSS) formatting, see Figure 2.3. Underneath is the introduction and link to the match article, which is also placed in a `<div>` structure of its own. Then comes the live commentary, summary and timeline slider structures. By the use of JavaScript and another set of `<div>` tags, it is possible to choose either live commentary or summary by clicking on links presented as tabs on the page. Both the live comments and

```
<div class="bd matchresults matchresultsafter">
    <div class="hd">
        <div class="wrapper">
            <h2>
                <span>Saturday October 3, 2009</span>
                <span class="status">Finished</span>
                <span class="last">Old Trafford</span>
            </h2>
        </div>
    </div
</div>
```

Figure 2.3: Yahoo! Sample HTML code

summary comments are presented by using an HTML unordered list tag. To create the live team lineup section, they have placed an image of an empty field on the page, and then created and placed an unordered list with the names of the players in the starting lineup on top the image. This is done by using styling tricks that makes it possible to specify placement on the page down to the pixel. The substitutes for each of the teams is also placed in an unordered list inside a `<div>`. On the bottom is another `<div>` structure with an unordered list containing the user generated comments.

The rigid structure of the HTML match pages makes it easy to identify structured information on the Yahoo! site. It will be relatively easy to traverse the list of live commentary, and for instance identify the events by looking for a specified image tag.

2.1.2 Live Goals

The *Live Goals* site domain is registered on *LiveGoals.com*, located in Hellerup, Denmark. There is little information available about the company behind, their vision and purpose. Although, the index page header states that the site provides *football livescore*, *result & fixtures* with *live commentary* from soccer matches world wide. The financial motive for the site seems to be live betting. in addition, they sell advertisement towards betting companies, and provide links to several betting sites. They also provide live betting odds from many of the major betting sites [6].

The screenshot in Figure 2.4 shows the site layout when displaying a live match. As can be seen, the layout consists of a header containing available languages, some contact-, advertising- and site map links, and a text describing the content of the current main page. Underneath the header is a horizontal navigation bar, containing links to all the major features of the site. The left side of the site contains different kinds of navigational links, as popular leagues, international events, scandinavian leagues, and also different advertisement boxes. On the right side there are more advertisement

[6] Bet365, Expekt, Ladbrokes, Betsafe, Unibet

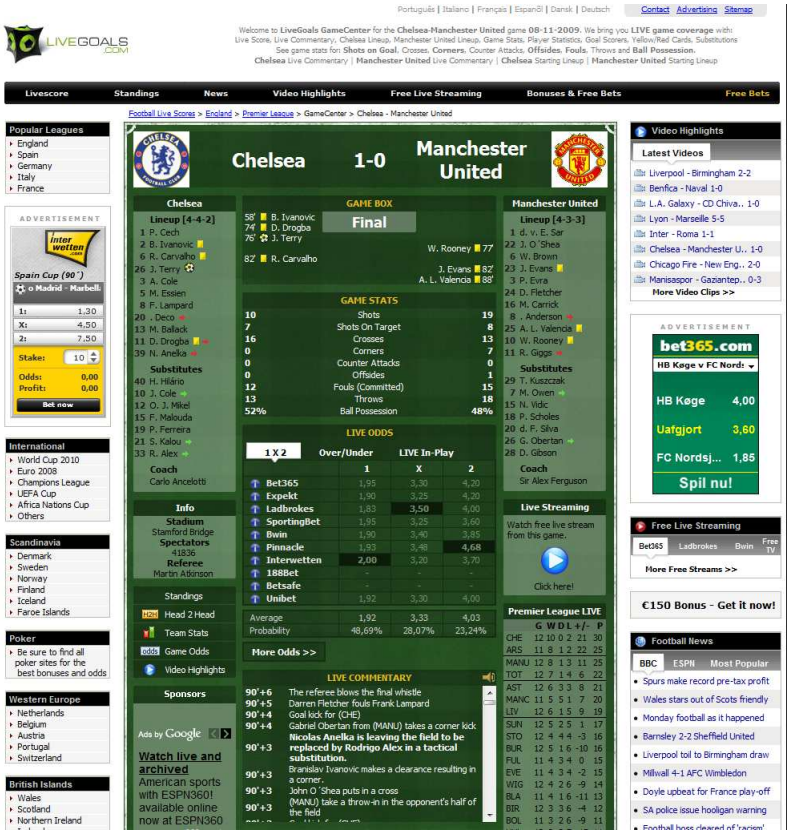

Figure 2.4: Live Goals Site Layout

boxes and also video highlights, free live streaming, and football news from some major news sites, like the British Broadcasting Corporation. There is also links to partner sites on the bottom of the right side. In the middle is the main content of the site, containing the information about the football match.

2.1.2.1 Content

In comparison with *Yahoo! Sports*, *Live Goals* does not have match facts about team captains and suspensions. But is has attendance information, telling the official number of peoples attending the match, and the name of the referee.

When it comes to live features, also *Live Goals* has live commentary, a summary section, and live lineup information. The live commentary section has minute granularity. Event detection is possible by identifying images for each specific event. The summary section displays goal-, and booking-events. Each such event contains event time, event image and the name of the player

Figure 2.5: Live Goals Player Stats

involved. This summary can easily be parsed to retrieve structured data. In the live lineup feature the major events; bookings, goals and substitutes, are depicted by images behind the player involved. As mentioned in the introduction, the site has a betting focus, and each game has a live odds box displaying game odds at some of the major online betting sites. *Live Goals* also has a *player stats* feature, see Figure 2.5, consisting of a football field with player jerseys, including numbers, on it. And as can be seen in the figure, it is possible to view player stats like goals, shots on target, shots off target, and offsides by moving the mouse pointer on top of the wanted player jersey.

And, as the *Yahoo!* site, registered users are allowed to comment on the games.

2.1.2.2 Data

Similarly to the Yahoo! Sport site, *Live Goals* pages describing premier league matches also have a common HTML structure. The common structure consists of a main `<div>` and several nested `<div>` tags and tables for the content described above. The *event-information* div contains a table with the name of the teams playing and the result of the game. There are also two `<div>` tags for each of the teams lineups, and they are called *hometeam-lineup* and *awayteam-lineup*, respectively. The div containing information about stadium, spectators, and referee, is nameless. Then there is a *eventCenterPane* `div`, containing five other `<div>` tags for game summary, game stats, live odds, live commentary and player stats. The game

box div contains a table with two other tags, one for each team's incidents. Game stats are listed in a table inside the outer div. Also the live odds contains a table where the odds are displayed. There is also a table structure inside the live commentary , holding each comment published on the site. Finally, the players stats contains a player field , and several jersey image tags with specific coordinates for each jersey. The players statistics are displayed by the use of a javascript snippet activated by a *mouse over effect*, displaying a popup box over each jersey.

On the bottom is a *commentary-section* , where each user comment is displayed inside a nameless .

2.1.3 Sky Sports

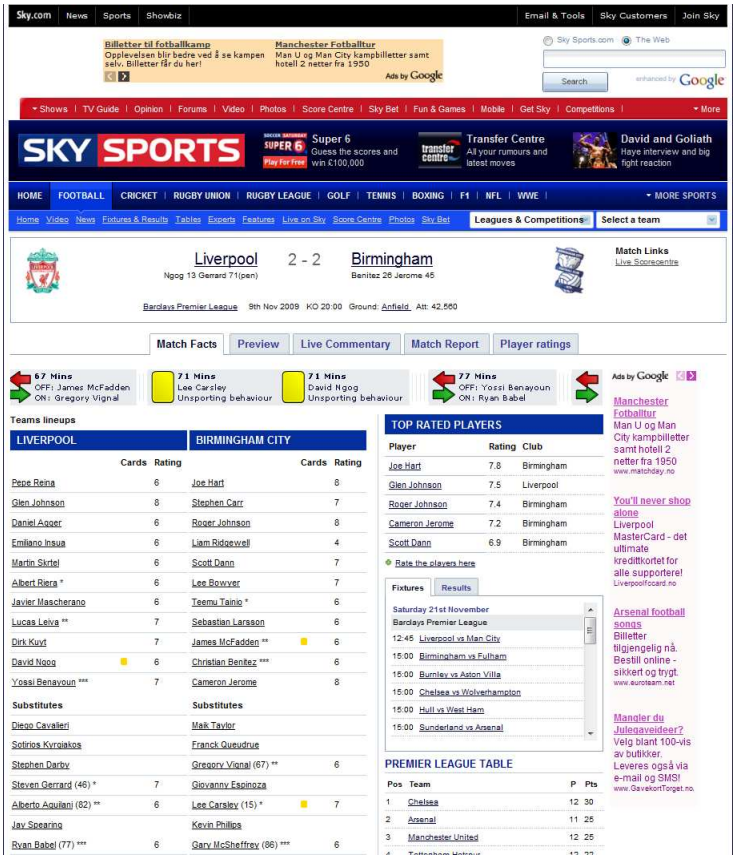

Figure 2.6: Sky Sports Layout

Sky Sports is owned by *British Sky Broadcasting*, which is a company that operates a subscription television service in the UK and Ireland. They have

for many years had exclusive broadcasting rights for *Premiership football*, and this have been the foundation of their success. Although their main focus is on producing TV content, they have also committed resources into their online services. And the *Sky Sports* site covers all the popular sports in UK and Ireland. This includes football, cricket, rugby, golf, tennis, boxing, and formula one, among others.

Figure 2.6 is an overview of the site layout covering a Premier League match. As the figure shows, the top of the page consists of different horizontal navigation bars, advertisement, and a search box. Then comes the *Sky Sports* logo with links to three featured articles. Underneath that comes the main navigation bar for navigating between the different sports covered by *Sky*. Based on the the chosen sport another navigation bar pops up underneath the main navigation bar, enabling browsing to the major events for that sport. At the bottom comes the main content space, and in Figure 2.6, this is a page covering a Premier league match.

2.1.3.1 Content

In comparison with the other two sites, *Sky Sports* have most of the same information when it comes to the match facts. However, it does not have information about team captains or team managers available. But in addition to the two other sites, it has information information about yellow and red cards without doing any live comment event detection.

When it comes to other features, *Sky Sport* have most of the same features as the other two, and even some more. Although, it does not have a countdown feature, as *Yahoo!*, and no match status field as both the other two have. Also, there is no live odds feature present.

Their live commenting section has minute granularity, similarly to the other sites. Event type detection is also possible by looking for event specific image url's. The summary section displays major events like goals, substitutions and booking, no different from the other sites when it comes to content. The live lineup feature also includes player ratings, and as the others information about booking and substitutions made.

Similarly to the Yahoo! Sports site, the Sky Sport site has an after match article. What makes it different is a comprehensive match statistics section containing information as possession, territorial advantage, shots on and off target, tackles and tackles success, and more. The article page also has special section about *goal of the match*, *man of the match*, *save of the match*, and *talking point*. The three first are described and justified, and the *talking point* is an description of the match event that most likely will be the hearth of the post-match discussion.

Sky also has a comprehensive pre-match information as statistics about their last meetings and their resent results against other teams. There is also a preview article discussing the teams resent performances, current injuries

and it also suggests possible starting lineups. And finally they have a result prediction.

When it comes to user interaction, they have a feature where the users can rate the players with point from one to ten. The average is computed and presented for each of the team squads.

2.1.3.2 Data

The main content for each game description consists of a *match header*, which is several nested `<div>` tags containing the teams logos, names, the result, the event, date, game starting time, stadium, and attendance. For navigation between the major features offered in the match coverage, *Sky* has chosen to implement a navigation bar rendered as *tabs*. This tab bar is static and built up by using special styling for an unordered list with links to the different features, see Figure 2.7 for code sample. The match facts page, see Figure 2.6, contains the live lineup and summary feature. An Adobe flash object is used to render the summary data, which makes the information unavailable in ordinary HTML format. This because the data is only possible to access through an Adobe flash player. For the live lineup, they have used a couple of tables inside a div. The preview page contains no fancy structures, a div with paragraphs for the article content. The same is done for the live commentary content. Additionally, the math report contains additionally a couple of tables for the statistics summary. The player ratings page is built upon a form for picking up the rating chosen by the user, to align the data they have used a couple of tables in combination with styling.

```
<div class="ss-tabs ss-tab-style1 ">
<ul class="anchors">
<li id="match-tab-facts" class="tabs-selected">
<a href="/football/match_facts/link.html">Match Facts</a>
</li>
<li id="match-tab-preview">
<a href="/football/match_preview/link.html">Preview</a>
</li>
<li id="match-tab-live">
<a href="/football/match_commentary/link.html">Live Commentary</a>
</li>
<li id="match-tab-report">
<a href="/football/match_report/link.html">Match Report</a>
</li>
<li id="match-tab-ratings">
<a href="/football/user_ratings/link.html">Player ratings</a>
</li>
</ul>
</div>
```

Figure 2.7: Sky Sample HTML code

2.1.4 Comparison

In summary, the three sites we have surveyed have overlapping content when it comes to *match facts*, *live event descriptions* and *features*. *Sky* has the most complete picture by covering a wide variety of pre-match information, minute granularity live commenting, and a comprehensive post-match article and statistics section. *Live Goals* and *Yahoo!* have very similar information, although Live Goals have a statistics section that is not present at Yahoo!.

In Table 2.1 and Table 2.2 we have categorized and compared the information available at each site. The categories are *Match facts* and *Live Events*. When analyzing the information, we have considered if the information on the HTML page has some structure in it that allows for easy identification and retrieval. An example is the home-team- and away-team-names, on each site this information is in a specific structure and therefore easy to retrieve by extracting that structure from the page. Another example is live commenting events like yellow and red cards. These events have an image url related to it, and therefore it will be easy to identify comments describing such events. On the other side we have match articles. Although, containing a lot of useful information and event descriptions, there is no structure that enables identification of these, making it difficult to identify any structured information.

	Sky Sports	Live Goals	Yahoo!
Attendance	yes	yes	no
Away team	yes	yes	yes
Captains	no	no	yes
Date	yes	yes	yes
Home team	yes	yes	yes
Lineup	yes	yes	yes
Managers	no	yes	yes
Result	yes	yes	yes

Table 2.1: Match Facts

As can be seen in Table 2.1, all three sites have the most common match facts available. *Sky* is missing captains- and managers names, but this is not that crucial. *Live Goals* is only missing captains, and *Yahoo!* is missing attendance information.

When it comes to live event information, see Table 2.2, they have a lot of common information. The most important events like, goals, bookings, live lineup, half- and end-time whistle is detectable and extractable.

	Sky Sports	Live Goals	Yahoo!
Goal	yes	yes	yes
Penalty	yes	no	no
Yellow card	yes	yes	yes
Red card	yes	yes	yes
Offside	no	no	yes
Own goal	no	yes	no
Lineup	yes	yes	yes
Half-time	yes	yes	yes
Full-time	yes	yes	yes
Substitution	yes	no	yes
Special	no	no	yes

Table 2.2: Events

2.2 Web Extraction

Extracting structured data from unstructured- and semi-structured web data is an old field of study [1] [10] [13], and the fact that most web pages have some structure can be exploited to generate structured data.

A semi-structured document, like for instance web pages, are organized and grouped together in semantic entities, which may or may not have the same attributes. The order of the attributes might not be important, and not all attributes may be required. Also the size and type of the same attributes in a group may differ. And it is obvious that it is much harder to query and retrieve information from such sources, as opposed to structured information sources like databases.

Anyhow, semi-structured means that there is some structure in the document that can be identified and extracted. Web pages are, for the most part, built up by HTML code and clean text. The structure in these documents comes from the HTML tags used to build up the page. If a set of similar structured HTML documents are describing similar content, the identifiable information can be semantically identical. The information can then be extracted and put in a database, and then we have created structured data from the semi-structured web content.

An example of such semantically equal pages are the Premier league match pages described in Section 2.1. Each site has a unique way of building up their web pages describing a match, and all pages describing matches on a site have identical HTML structures, although the content is different. Because of recurring page structures within a site, we can create one *wrapper*, and extract structured match information from all the pages on that site which describes premier league matches. And finally put the extracted information in a database for later retrieval.

Wrappers are specialized pieces of software that parses through web data

looking for structure that identify data of interest and maps that data to a suitable format as for example a relational table.

2.3 Web mining

Web mining [6], or the art of searching for valuable information in the ever growing ocean of information available on the world wide web, consists of two major operations. The first is to find and extract web pages that might contain valuable information, also called *Web crawling* or *spidering*. The second is to identify and extract wanted information by creating cite specific wrappers, often called *web scraping*. In the following sections we will give a brief introduction to the two above mentioned areas.

2.3.1 Crawling The Web

A *web crawler* is usually given a set of starting url's as a starting point for the crawl. Then the crawler parses through each *seed page* harvesting hyperlinks leading to other pages on the same site or possibly pages on another site. The harvested hyperlinks are recursively visited according to a set of polices controlling how thorough the crawl should be. For most crawling projects the crawl need to be substantially limited and executed with smartness, the reason for this is the share volume of web pages on the web, available bandwidth and time.

To limit a crawl we can apply a *selection policy* stating which links to follow and thereby which pages to download. This *selection policy* should be based on the purpose of the crawl. For instance, is the purpose to only parse HTML content and avoid other content types, we limit the crawler to only download HTML content and drop all other types. If we want to harvest pages from a particular site, we do a *path-ascending crawl* [8] by starting with the index page of the site, identify all links out from that page and follow each one looking for new links to follow. If the purpose is to gather information about a particular subject or topic, we can do a *focused crawl* [7] [21] [22]. When executing a focused crawl we want to identify interesting pages without actually downloading them, and this is a difficult problem. One way of predicting the content is to use the anchor text of the hyperlink as a hint to what the content will be. Some projects [20] [25] aim to crawl the *deep web* [4], which refers to the content hidden behind HTML forms. To apprehend information behind such forms, a user must submit a form with valid input values. Implementing crawlers able to deal with this complexity, is a complex task.

The web is constantly changing, pages are added, modified and deleted. Outdated information is less valuable for many systems, therefore, pages have to be re-visited at some frequency. Two possible approaches are *uniform-* and *proportional* re-visiting, where the uniform approach all pages in a

collection is re-visited with the same frequency. Or *proportional* re-visiting, which involves re-visiting pages proportional to the update frequency of the page.

Because web crawlers can retrieve data in much faster pace and depth than humans browsing manually, they can put to much load on a web server and therefore cripple its performance. Users might receive poorer service quality because of this, which is not acceptable. Therefore web crawlers must act in a polite manner, and adhere to crawling policies and politeness norms that limits the polling frequencies to an acceptable level. Several polling frequency intervals have been proposed, but one of the first was from Koster [7], who suggested a polling frequency of 60 seconds. This frequency has been shown to be to large, and most crawlers today use a more aggressive polling frequency. Also dynamic polling frequencies are used. Dynamic polling can for instance be based on the download rate of the first page retrieved from a site, as described in [14].

The web is very large, and therefore there is need for parallelizing the crawling process to achieve maximized download rate. This can be done by running multiple instances of a crawler and orchestrate the different crawler by using a scheduling algorithm. The scheduler must especially deal with duplicate url's to avoid download the same page several times. When crawling the whole web, a *distributed crawler* must be used, which uses a cluster of computers to perform the crawl efficiently. An example of such a distributed crawler is *Nutch*[23].

2.3.2 Scraping The Web

Scraping the web is about extracting structured information from semi-structured web data. Web data is described as semi-structured because HTML code imposes some structure that can help identify and retrieve structured data from a web page. *Wrappers* are pieces of software that are implemented specifically for parsing through text looking for structure that can be used to identify information pieces. Wrappers are often created for a specific task, and an example in this thesis domain can be a wrapper implemented for extracting live commentary text for specific events. For instance a wrapper unleashed on a *Yahoo!* match description page, looking for match comments describing scored goals or yellow cards.

There are several approaches for creating wrappers, and [18] gives a brief survey over web data extraction tools where they have identified and described the following groups.

Languages for wrapper development which are languages especially designed to assist users in constructing wrappers. Examples of such languages are *Minerva* [9], and *Web-OQL* [3].

[7] `http://www.robotstxt.org/guidelines.html`

HTML-aware tools are tools that rely on inherent structural features of HTML documents for accomplishing data extraction. Example tools are *W4F* [28] and *RoadRunner* [10].

NLP-based tools uses Natural Language Processing (NLP) techniques to learn extraction rules for extracting relevant data existing in natural language documents. This technique works best on HTML pages consisting of free-text. Representative tools are *RAPIER* [5] and *SRV* [12].

Wrapper-induction tools generate delimiter-based extraction rules derived from training examples. In comparison with NLP-based tools, they do not rely on linguistic constraints, but rather in formatting features that implicitly delineate the structure of the pieces of data found. Tools using this approach are *WIEN* [16] and *SoftMealy* [24].

Modeling-based tools are tools that, given a target structure of objects of interest, try to locate in Web pages portions of data that implicitly conform to that structure. The structure provided is built up by modeling primitives like tuples, lists. *NoDose* [1] and *DEByE* [17, 26] are examples of tools using this approach.

Ontology-based tools differ from he other tools described above, in the sense that they do not rely on any structural presentation features for the data within a document. Instead extraction is accomplished by relying on the data. This can be done by identifying a specific domain application, and then use an *ontology* to locate constants present in the page and to construct objects with them. An example is a ontology based tool developed by the *Brigham Young University Data Extraction Group* [11].

One thing to keep in mind when scraping web data, is that web sites might update their design at some point. When that happens, wrappers might have to be updated. Rewriting wrappers is tedious work, but still most wrappers are updated manually due to the fact that fully automatic wrapper generation is very hard.

Chapter 3

Design and Implementation

This chapter starts by describing the system architecture, before explaining the design and implementation details for each of the major system components.

3.1 System Architecture

Our system is one part in a larger live video search service. This service is in need of fresh web metadata for annotating live videos. Our system is responsible for extracting the metadata from a pool of web sources containing overlapping information. As shown in Chapter 2.1.4, these exists. Our solution is based on the idea that we can use the *pool of web sources* with overlapping data, and ensure that no single source is overloaded, while still keeping the data at a reasonable freshness rate.

Our system architecture consists of four main components: an *orchestrator*, a *crawler*, a *scraper*, and a *database* for storing the extracted web data, as shown in Figure 3.1. The *orchestrator* is in charge for scheduling the crawler to fetch the HTML pages from its specified web source. And in order to schedule the targeted crawls correctly, the orchestrator must maintain metadata about crawling statistics for each individual web source. This way the orchestrator can judge which site to pull data from in each update interval.

The *pool of web sources* connected to the system must have a minimum set of overlapping information that is possible to detect. In the soccer domain this is for instance goals, bookings, and substitutions. Sources can have additional information that is not overlapping. This will allow our system to gather more information whenever these sources are pulled. Also, the web sources must contain recurring HTML structures that allows for data identification and retrieval. The crawler must be able to identify specific pages containing the information wanted from each web source, download and store those page for later to be processed by the scraper. The scraper

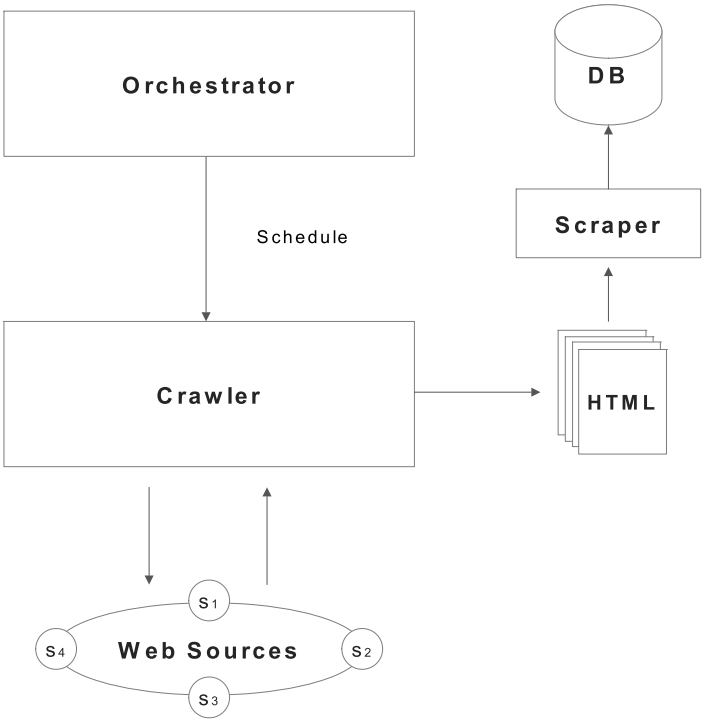

Figure 3.1: System Architecture

will parse the downloaded pages, and try to identify and extract the wanted data, and finally store it as structured data for later retrieval. A relational database will be used for persistent storage of the extracted data.

3.2 Orchestrator

The orchestrator triggers the crawler to fetch the wanted HTML page from a chosen web source, and then initiate the scraper to identify and extract the wanted data. The orchestrator keeps track of crawl statistics for each source, including *last pulling time*, *download time*, *failure rate*, and *timestamp of last discovered event*. Based on the crawl statistics and the wanted data freshness, the *orchestrator* dynamically schedules the pulling interval and which site to pull from. In our current implementation, the orchestrator follows a *Round-Robin* based schedule - this ensures a balanced load on each source. However, if a source seems to be responding slowly, the next non-struggling source will be selected instead. This keeps the freshness rate at the required level, while ensuring that struggling sources are not overloaded

more than they already are.

The *orchestrator* is initialized with a default pulling interval and have the ability to initiate a crawler and scraper capable of downloading, identifying and extracting information from the available sources. First the orchestrator selects one of the sources to pull from. This triggers the crawler, which then identifies and downloads the wanted HTML page. The scraper then takes over and parses through the HTML content, identifies and extracts the wanted information, before executing an update query to the database where the scraped data is stored. When the crawling and scraping is done, and the data is persistently stored, the orchestrator updates its metadata with crawling statistics for the selected source.

At the next update interval, the next source in line is selected, and the above procedure is repeated. When all sources have been pulled, the orchestrator takes into account the gathered crawling statistics when selecting the next source to pull from.

3.2.1 Implementation

The *orchestrator* is implemented as a python process, which schedules and executes the focused crawl for the wanted duration with a specified polling interval. The orchestrator must be initialized with information about *duration*, available *source configurations* for scheduling, and default *polling interval*. Before executing the main loop, an initial *configuration index* must be selected, which will be the first source scheduled. Also, a *count* variable, keeping track of how many times the scraper has been executed, must be initialized with zero.

To execute the crawling process for the wanted duration and freshness rate, we have implemented a *while loop* running as long as the execution *count* is below the value that is set by *duration/polling-interval*. To enforce the polling frequency we execute a *sleep* function at the end of the loop. The code below controls the loop execution.

```
# Main loop
# Executed for wanted duration with set polling interval
while count < duration/polling_interval:

    # Record start time
    startTime = time.time()

    # == CODE HANDLING SCHEDULING AND CRAWL EXECUTION ==

    # Record stop time
    stopTime = time.time()

    # Update number of crawls
    count += 1

    # Set sleeping interval and sleep
    timeTaken = stopTime - startTime
    if polling_interval - timeTaken < 3:
```

```
    time.sleep(polling_interval)
else:
    time.sleep(polling_interval - timeTaken)
```

The Round-Robin scheduling is implemented by having a *list* of *source configurations*, which are iterated over. As the code below shows; this is simply done by having a list index variable that is updated for each iteration.

```
# Select the scheduled configuration
conf_id = source_configurations[conf_index]

# == Execute the crawler and scraper as a subprocess ==

# Round-Robin schedule the available source configurations
if conf_index < 2:
    conf_index += 1
else:
    conf_index = 0
```

To trigger the cite specific scraper, the orchestrator process starts another Python process carrying out the job. This is done through the Python *subprocess module* [1]. The scraper process is executed with a *configuration id* parameter, which is the primary key for a source specific crawling configuration, which will be retrieved from the database by the scraper during setup. The other parameters are standard setup for executing the scraper. The code below shows the complete scraper execution implementation:

```
# Execute the crawler and scraper as a subprocess
# Set the scheduled scrape configuration id as parameter
try:
    subprocess.Popen(
        ["python " + orch_config.SCRAPY_SCRIPT_PATH + "scrapy-ctl.py " + \
         "crawl match_spider" +\
         " --nolog " +\
         "--set CRAWL_ID='"+str(conf_id)+"'"
         ], shell=True)
except:
    print "Error running scraper"
```

The orchestrator is also responsible for updating the individual sources crawl-configurations with information about the last timestamp gathered for each scraping. This is implemented by invoking a function querying the database for information about *last timestamp*, then this value is written to the configurations residing in the database by executing an update query for each configuration individually. The implementation code can be found in Appendix A.

When the *duration* is over, and the loop is done, the orchestrator process stops executing.

[1] http://docs.python.org/library/subprocess.html

3.3 Crawler and Scraper

The *crawler* and *scraper* is designed to operate intimately; one finding and downloading the HTML page containing the wanted information, and the other parsing through the content, identifying and extracting the wanted information.

The *crawler* starts from a specified *seed url*, downloads the HTML page and analyzes the links. The links are analyzed based on a set of rules, pages linked to are classified either as *follow through links*, *information links* or links to be *discarded*. *Follow through links* are links to pages that contains other links that might lead to pages that contains the information we want to scrape. The crawler downloads pages these links point to, and analyzes these pages links in the same matter as the *seed url*. *Information links* are links to pages that contains the information we want to scrape. When an information link is discovered, the crawler downloads the HTML content and makes it available for the scraper. *Discarded links* are links identified as not containing the wanted information, and not containing links to pages that might contain links to pages with the wanted information. And as the name suggests, these links are thrown away. Figure 3.2 illustrates the crawling process.

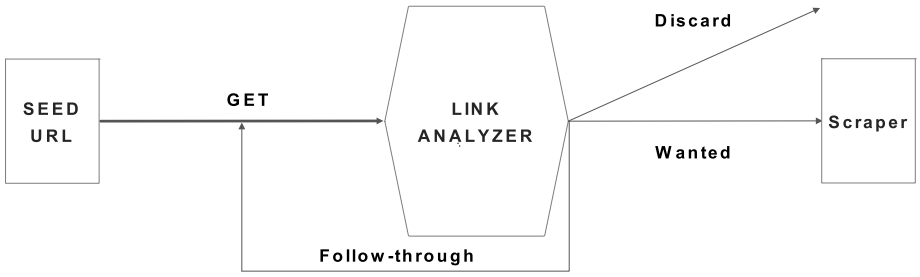

Figure 3.2: Crawling Process Flow Diagram

After the crawler has found and provided the wanted HTML page, the *scraper* takes over. As mentioned several times, the scraper shall identify and extract the information we want from the page. In Chapter 2.2 we mentioned several approaches to create wrappers capable of identifying and extracting the information we want from a specific web page. Our approach fall under the *HTML-aware tools* category, because it takes advantage of inherent structural features of HTML documents. We have done this by identifying recurring HTML structures where the information we want resides. The identification is made possible by using the *XPath query language* [2], which is a language originally designed for querying information in Extensible Markup Language documents (XML). However, this language

can also be used to query for information in HTML structured documents, which is suitable for our needs. After the data location have been identified, the data is retrieved. However, the retrieved data might not be suitable for direct insertion into the database, because there can be unwanted characters or other unwanted information. Therefore, the data is *cleaned* before insertion, this way we also ensure that the data is in the correct format before sending it to the database. Figure 3.3 illustrates the scraping process.

Figure 3.3: Scraping Process

3.3.1 Implementation

Our crawler and scraper implementation have been written by using a Python framework called *Scrapy* [2], which is an open source framework intended for crawling web sites and extracting structured data from their pages.

When writing a site-specific crawler and scraper in *Scrapy*, four major components must be implemented. First the *crawler* itself, which will start from a specified seed url and work its way through the site by jumping from link to link retrieving wanted pages and providing them to the scraping functionality of the framework. Second is the scraper functionality, which is implemented as a HTML content parser. The framework provides a functionality called *HtmlXPathSelector*, which enables the programmer to use a *XPath* to identify and query a specific HTML structure for its text content in any HTML page. By using this functionality the parser is able to pin-point and query for each wanted piece of structured information in a page.

All information scraped from a specific page must be stored in an *item* object. And that is the third component required by the framework, namely a model that defines all the structured data we want to extract from a specific page. The model is implemented by *inheriting* an *item* class which enables certain capabilities that allows for convenient storing of the structured data at gathering time and easy access to the structured data after retrieval.

[2] http://scrapy.org/

The *item* is also used as an inter-process communication object, which bring us to the fourth component of the framework, namely the *item pipeline*. Each scraped item is sent to this pipeline, which typically will clean the raw extracted data, validate it, format it, and lastly store it persistently. The following subsections will give more implementation specific details for each component mentioned above.

3.3.1.1 Initialization

As mentioned in Section 3.2.1, the orchestrator executes the Scrapy script with a parameter called *crawling id*. During initialization the Scrapy process uses the *crawling id* when executing a query retrieving the source specific *crawl configuration*. The crawl configuration contains information about *match id*, *source*, *seed url*, *rule*, and *last timestamp*. The *seed url* is then used to initiate the list of *start urls*, and the *rule* is used to create a rule identifying the page that should be scraped. The other configuration variables are used later on in the system.

3.3.1.2 Crawler

As mentioned in the introduction, the *crawler* starts from a specified seed url, downloads the page and start examining the links in that page.

When implementing the crawler functionality in *Scrapy* we first specify the seed url, and this is done by writing the following code:

```
start_urls = [
        "http://domain1.com"
    ]
```

The link analysis functionality is implemented by specifying *rules* that the crawler must follow when examining each individual link. A rule is created by specifying a *Link Extractor* object, which defines how each link will be treated. The *Link Extractor* object is created by specifying a *regular expression* describing a format the url must conform to, to be downloaded and scraped, downloaded and link analyzed, or discarded. To specify what shall be done with the links that conforms to a rule, different arguments must be given to the *link Extractor object*. For instance, if a link is identified as a link to a page that shall be scraped, a *callback* argument must be added. The *callback* argument is a string with the name of the *scraper* that will be called for each link extracted with the specified *rule*. Rules created for identifying links to pages that must be downloaded and link analyzed must contain a `boolean` argument *follow*. Following is a code sample specifying a rule that identifies the url to be scraped:

```
rules = (
    # Identify links going to soccer match details
    rule = "http://domain1.com/pl0809[^\s/]+-[^\s/]+-\d+\.html"
    Rule(SgmlLinkExtractor(allow=(rule,)), callback='parse_match'),
)
```

As you can see, this rule contains a rather complex *regular expression* specifying how url paths to soccer match details pages looks like. All links that match the regular expression is downloaded and provided to the callback function *parse match* which is the scraper that is implemented to extract the wanted information from that specifically structured page.

3.3.1.3 Scraper

After being invoked by the framework, the scraper has access to the HTML content of the page that shall be scraped. Then the scraper must be able to pin-point the structured data we want to extract. As mentioned in Section 3.3.1, this is done by using the *HtmlXPathSelector*, which has a method that is called `select`. The `select` method takes an *XPath* and uses it to identify and extract the exact HTML structure where the data we want is residing. A clever way of retrieving the XPath for a specific piece of data in a page is by using the *Firefox* [3] plugin *Firebug* [4]. As can be see in Figure 3.4, the XPath for the highlighted text can easily be retrieved. The resulting XPath will look like the following:

```
/html/body/div/div[10]/div[3]/div/div/span/div[4]/table/tbody/tr[2]/td
```

If we provide this XPath to the `select` method we can extract the number of goals for the home team in that specific match. During implementation we discovered that this method is not perfect, because the XPath given by Firebug is wrong sometimes. The error lies in that Firebug is not able return the exact XPath, often the two or three last elements of the path is incorrect. However, it gives a very good starting point for identifying the exact XPath, and by back tracking the given interactively in Firebug, it goes rather quick to identify the correct one.

The select method can return both a HTML structure object that can be further examined with another select call, or it can return the text residing inside the extracted HTML structure. The following code show how the number of goals is retrieved from the selected HTML structure:

```
item['homeScore'] = hxs.select('/html/body/div/div[2]/span[1]/text()').extract()[0]
```

If you look closely at the XPath given as parameter to the select call, you can see "text()" at the end. This specifies that we want the text data residing in the structure. The `extract` method returns a list containing this text, and we retrieve the text by specifying index zero, because the list only contains the one text we want.

In our case we also wanted to extract live comments residing in a HTML list structure. We solved this by selecting the list structure and then loop

[3] http://www.mozilla-europe.org/no/firefox/
[4] http://getfirebug.com/

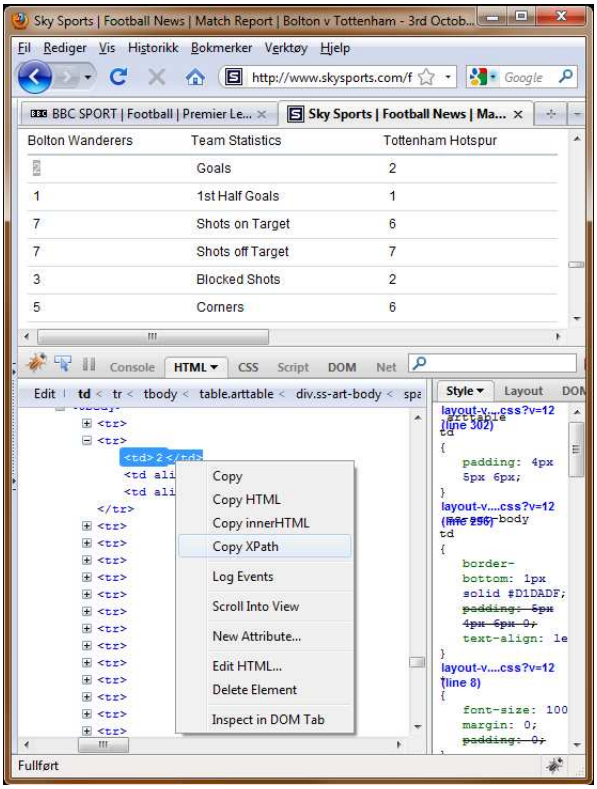

Figure 3.4: Firebug XPath Extraction

through all of the elements in that list extracting all the comments by doing a second `select` call. Then by using another set of third level `select` calls we extracted the wanted information we wanted from each comment. To avoid duplicate comments when returning to the same HTML page several times, we kept track of the timestamp of the last comment retrieved. We then compare the timestamp of each possible new comment to that timestamp, and if the new comment timestamp is older than, or as old as the last recorded timestamp, we throw it away.

When the scraping is done, all of the retrieved data is put in an *item* object, along with the crawl configuration settings, before being passed to the *pipeline* by the framework.

3.3.1.4 Items

Items objects are simple containers used to collect the scraped data. A dictionary-like API with convenient syntax is provided for declaring available fields. We implemented an item for storing scraped soccer match information in the following way:

```
# Model that defines a soccer match item

class MatchScraperItem(Item):

    # crawl id
    crawl_id = Field()

    # match id
    match_id = Field()

    # item source
    source = Field()

    # source timestamp
    source_timestamp = Field()

    # number of goals scored by home team
    home_score = Field()

    # number of goals scored by away team
    away_score = Field()

    # all live comments
    comments = Field()

    pass
```

As can be seen, the *item* implementation is very simple and straight forward. And we also added some configuration variables to the item. This because items is a natural communication point between the scraper and pipeline, and therefore we used them as inter-process communication objects as well.

The API provided for accessing created items is also easy to use, it makes it easy to access a single field or even iterate through all fields by retrieving a dictionary with all field keys and values. It is also possible to extend the item object capabilities, but this was not necessary for our purposes.

3.3.1.5 Pipelines

After a page has been scraped and the item is filled with scraped information, it is handed over to the *pipeline*. As mentioned, the pipeline is responsible for cleansing the scraped data and storing it persistent. Our pipeline implementations is very compact, as can be seen in the code below:

```
# Pipeline storing items in database

import mysql_interface

class MatchScraperPipeline(object):

    def process_item(self, domain, item):

        crawl_id = item['crawl_id']
        match_id = item['match_id']
        source = item['source']
        home_score = item['home_score']
        away_score = item['away_score']
        comments = item['comments']

        # connect to database
        db = mysql_interface.MysqlInterface()

        # update match score
        db.update_match_score(match_id, home_score, away_score)

        # update match comments
        i = 0
        for comment in item['comments']:
            db.add_match_comment(match_id, comment['timestamp'], "blank", \
                                 comment['comment'], source)

        # update last comment for current source
        db.update_last_timestamp(crawl_id, last_timestamp)

        return item
```

During the scraper implementation we discovered that the data we sent to the pipeline did not need any cleansing or re-formatting, and that is one of the main reasons for the code being compact. Our pipeline have two main purposes; the first is to update the database table *match* with information about how many goals each of the two teams have scored, and the second is to put all comments in the database table *comments*.

The pipeline is also made responsible for updating the crawl configuration with the *last timestamp* found in a comment. To communicate with the database, we have implemented a *mysql interface* which is used by the pipeline object, this interface will be described in the following section.

3.4 Database

The main purpose of the database is to store scraped data, in addition, it is used for interprocess communication between the *orchestrator* and the *scraper* framework. Figure 3.5 shows the entity relationship model the database implementation is based on. As can be seen, we have the *crawl configuration* entity for storing the information needed by the scraper. The entity has a many-to-one relationship to the *match* entity, which means that a specific crawl configurations is concerned about a specific match, and that a certain match can be updated by several crawl configurations. The *match*

entity has a one-to-many relationship with the *comment* entity, which means that one match can have several comments, but each comment can only be related to one match.

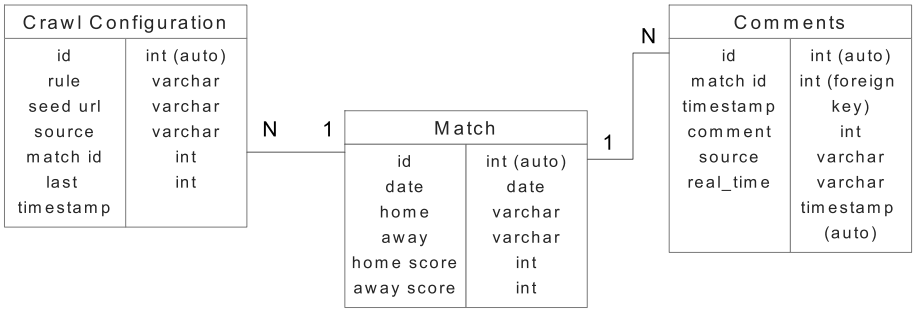

Figure 3.5: Database Entity Relationship Model

In our implementation we have chosen to use a *MySQL* [5] database. And to ease the interaction with the database we have implemented an python interface based on the external python module *MySQLdb* [6] handling our most common queries. See Appendix A for implementation details.

[5] `http://www.mysql.com/`

[6] urlhttp://sourceforge.net/projects/mysql-python/

Chapter 4

Evaluation

This chapter starts by explaining how we collected our raw-data set, finally we will describe the experiment where we launched our *scraper* on the gathered data, then we will analyze the scraped data and discuss *completeness* and *freshness*.

4.1 Raw-Data Gathering

To enable repeatable experiments, and the possibility to compare different configurations when analyzing freshness and completeness, we needed a set of raw-data. The focus of our raw-data gathering were a Premier League soccer match between Liverpool FC and Arsenal FC the 13th of December 2009. This match was chosen because it was a high stakes game were we anticipated several major events like goals and bookings would occur. The data sources we used were the three soccer sites analyzed in Chapter 2.1, the urls can be seen in Table 4.1

`http://uk.eurosport.yahoo.com/football/premier-league/2009-2010/liverpool-arsenal-282896.html`
`http://www.skysports.com/football/match_commentary/0,19764,11065_3205392,00.html`
`http://www.livegoals.com/gamecenter/liverpool-vs-arsenal-13-12-2009`

Table 4.1: Source Urls

To gather the raw HTML data we implemented a simple Python script, see Appendix A. The script was configured to download the above HTML pages at a polling interval of 30 seconds. Each version is saved to disk with a filename describing the source, version number, and file creation time, in the following manner.

```
-rw-r--r-- 1 tord tord 167542 2009-12-13 17:28 yahoo-56-1260721695.252973.html
-rw-r--r-- 1 tord tord 168480 2009-12-13 17:28 yahoo-57-1260721725.322187.html
-rw-r--r-- 1 tord tord 165957 2009-12-13 17:29 yahoo-58-1260721755.266330.html
```

In addition the script creates a statistics file with an entry for each downloaded page. Each entry contains information about; *version*, *start time*, *end time*, and *filename*. Both start and stop time is a floating point number expressed in seconds since the epoch, in UTC [1]. Start time is logged right before starting downloading the wanted page, and stop time is logged right after the page is downloaded. This enables us to calculate the total downloading time of the page. See below for example entries.

```
0 1260720017.052713 1260720017.567754 sky-0-1260720017.252011.html
1 1260720047.069432 1260720047.543971 sky-1-1260720047.275525.html
2 1260720077.090556 1260720077.631078 sky-2-1260720077.290289.html
```

During testing we discovered that network errors could occur, caused by the sources web servers denied our requests. To handle these errors, we implemented exception handling ensuring that the script would continue on and fetch the next version. During the final runs we observed no errors.

To gather the data, three versions of the script was executed concurrently, one for each source url. The scripts ran over a duration of two hours. The gathered HTML files were put in a separate directory for each source. All three scripts were executed on the same computer, which was a Dell Precision 290 running a 64-bit Microsoft Windows 7 operating system, on an Intel Q6600 Quad core CPU (each core 2.4GHz), with 4 gigabytes of memory. The resulting raw data set consisted of 231 different HTML page versions for each source.

4.2 Scraping Experiment

The purpose of the scraping experiment was to test our scrapers ability to extract data from the raw HTML data gathered. Collect a data set of live comments for analyzing freshness and completeness. And also enable repeatable experiments, and the possibility to compare different configurations.

The experiment was executed by modifying the *Orchestrator* implementation, described in Subsection 3.2.1, to work on the 693 HTML pages collected from the three sources. And as our scraper implementation required an url to work, we installed a local *Apache Web Server* [2], serving the gathered pages. One of the modifications done on the *Orchestrator* were that we needed to read the stats file from each of the slurps. This to create a python dictionary [3] for each of the sources containing information about *start time*, *stop time*, and *filename*. As dictionary *key* we used the HTML page version number. We could then further modify the code to iterate

[1] UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time, or GMT)

[2] `http://httpd.apache.org`

[3] `http://docs.python.org/tutorial/datastructures.html#dictionaries`

through this dictionary and execute an update query to the source specific scraper configuration and change the *seed url* for each scraping iteration. We also changed the control of the main loop to instead of executing at a certain frequency for a duration of time, to execute 231 times. As this was the number of HTML page versions we had for each source. In addition we had to change the scraper launching code to deal with the fact that we had to block until each scraping was done. This because the scraper relies on timestamp updates retrieved from the database, and this information must be updated after a scraping is finished before another scraping can be executed.

We also had to expand the database model to handle some extra information. The *crawl configuration* table had to be expanded with a *current source timestamp* to store the stop time of the downloaded page relative to the start time of the first downloaded page version for the source. And to enable freshness analysis on the gathered data, we had to expand the *comments* table with a column called *source timestamp*, which is collected from the new column added to the *crawl configuration* and inserted in each comment by the scraper.

Some minor modification had to be made to the scraper also, for instance to handle the new *source timestamp* information, we had to expand the *item* with a new field, and update each source specific extraction code to put this information into the scraped item. The pipeline and MySQL interface also had to be modified to deal with this. We also decided to skip extracting the home and away scores, because we were only interested in the comments, and in addition experienced some problems with the Yahoo HTML pages for the six first minute of the game. The problem seemed to stem from the fact that the Yahoo pages did not display the score for these first minutes, and therefore the HTML structures the scraper relied on was corrupted. We later on discovered that this also corrupted the comments extraction for the same six minutes.

Another experience we had was that all of the *Sky* HTML pages created a *to many redirects exception* in the scraper framework code. After investigation we discovered that these pages contained some piece of javascript code that triggered a page refresh after a certain time, and when we removed this piece of code the *Sky scraper* code functioned as supposed to. One problem was that we had 230 pages left where this piece of code had to be removed, however we solved this by using the below Python script.

```
import os
import sys

# dictionary for slurp information
sky = {}

# CREATE DICTIONARY WITH INFORMATION ABOUT ALL SKY FILES
f = open("sky_stat.dat")
i = 1
```

```
for line in f.readlines():
    if i == 1:
        i = 2
        continue
    run = line.split(" ")
    sky[run[0]] = {'start': run[1], 'stop': run[2], 'html': run[3]}

f.close()

# FOR EACH HTML FILE
for key, value in sky.items():
    print value['html']

    # OPEN FILE
    f = open(value['html'].replace("\n", ''), 'r')

    # CREATE LIST OF ALL LINES IN HTML FILE
    data_list = f.readlines()

    f.close()

    # REMOVE LINES CONTAINING JAVASCRIPT CODE TRIGGERING REFRESH
    del data_list[87:99]

    # REWRITE FILE WITH NEW LIST OF LINES
    f = open(value['html'].replace("\n", ''), 'w')
    f.writelines(data_list)
    f.close()
```

After the code modifications, HTML editing and Web server setup, we executed three separate runs, one for each of the sources. Each run scraped, timestamped and stored the comments in the MySQL database.

The resulting experiment statistics can be found in Table 4.2. As can be seen, the total number of comments extracted from those 693 pages is 287. However the slurping period started a couple of minutes before the actual match started and ran for a couple of minutes after the match had finished. In addition the match had a fifteen minute break in the middle, where no comments where added. Hence, not all page versions contains new comments. 287 comments mean that an accumulated comments rate for all sources per minute in this 90 minute game is about 3.2. However, as anticipated many of the comments will be overlapping information describing the same events.

Number of sources	3
Total number of pages	693
Total number of comments extracted	287
Average accumulated rate per game minute	3.2
Average rate per game minute	1.06

Table 4.2: Scraping Experiment Statistics

Table 4.3 contains statistics over comments distribution and rate per source. As the table shows.*Livegoals* have a much higher average comments

rate per match minute than the other two. Which means that they contribute more to the comments pool and that they could have shorter texts spread over several comments per minute or that they describe each event more thoroughly. Or perhaps they describe more events than the other two sources, although the event might have less importance to the match.

Source	Number comments	Average rate per game minute
Yahoo	66	0.73
Sky	76	0.84
Livegoals	145	1.6

Table 4.3: Comments Source Distribution Statistics

4.3 Event Completeness Analysis

In this analysis we identified important events among the comments extracted in the scraping experiment. The events we chose to identify were: *game start*, *goals*, *yellow cards*, *red cards*, *substitutions*, and *start second half*. According to the official game statistics [4], these important events should amount to seventeen in total during the game. Tree goals, six substitutions, six yellow cards, and of course start of first half and start of second half. What we wanted to investigate was event overlap between the sources, and to look at the event distribution over the 105 (two times 45 minutes, plus a 15 minute break) game minutes.

What we discovered was that the *Sky* and *Yahoo!* scrapings, had sixteen sixteen of the comments containing the wanted event descriptions. The one missing was a substitution happening in overtime, that the scraper missed due to the fact that this feature was not implemented. However, the *Livegoals scraper* also missed a couple of the substitutions at the end, but before the overtime. The reason for that is how the scraper avoids duplicate detections, and therefore risks to throw away some comments if there are several comments in the same minute that are spread over more than one file version. Naturally this did not inflict the other two sources, because their comments rate per minute is below one.

Figure 4.1 shows the event distribution during the game time relative to the timestamp of the first source detecting the start signal, and the events tagged in the timeline by the timestamp of the first source detecting the event. As the graph shows, only four events occur in the first half of the game, and therefore most of the events happen in the second half. The event rate also seems to increase the further into the game we are. This can be explained naturally by looking at the events we are detecting, because in

[4] http://www.premierleague.com/page/MatchReports/0,,12306 48593,00.html

most games the substitutions are carried out in the second half and most of the time at the end.

Before carrying out the freshness analysis, we manually inserted the two comments that the *livegoals* scraper did not retrieve, this to have a complete overlap of events between the sources. We also removed the start signal event, as this were to be used as reference point only.

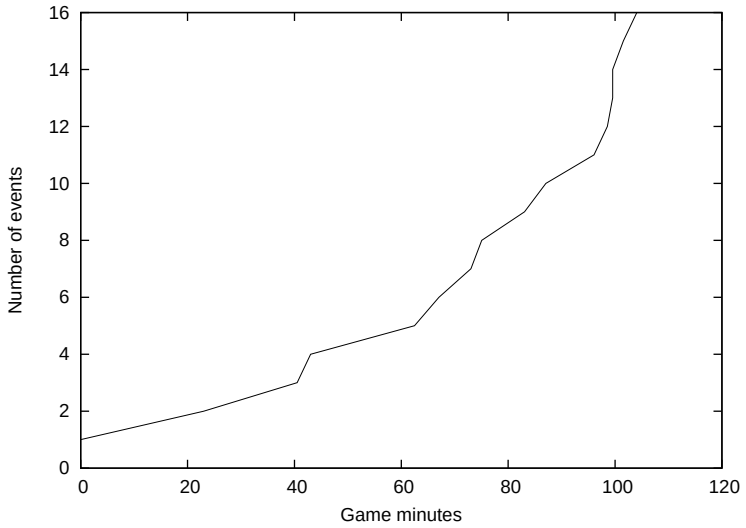

Figure 4.1: Event Distribution

4.4 Freshness Analysis

The purpose of the analysis was to compare the metadata freshness attained using a single source versus our system using three sources.

Freshness can be quantified as the time elapsed between an event is possible to detect, and when it is actually detected by our system. For instance, if the earliest possible point of detecting an event is 167 seconds into the game, and it is detected 180 seconds into the game, the freshness is 13. Therefore, as experiment baseline we used the earliest possible point of detection by simulating pulling as often as required. We did this by using our gathered raw data statistics, and created the baseline by selecting the HTML source timestamp that contained the event at the earliest point.

Our experiment configuration used a 60 second polling interval per site, as this polling interval is a reasonable visiting policy cite. The polling interval allowed our system to carry out a 20 second Round-Robin scheduled polling algorithm alternating between the three sites, resulting in a 60 second polling interval per site. The polling intervals were scheduled to start

from the timestamp of the first source detecting the match start signal.

Figure 4.2 shows the freshness in seconds per event for one source versus our system using three sources. As can be seen, using three sources we are able to attain better freshness for each event, versus the single source approach. However, this is a best case scenario, as we might not be able to retrieve data from the source detecting the event first. This because the second or third best source might not contain the event in that polling interval, perhaps causing our system several more polling intervals before apprehending the event. Although this might happen, our system will in general be able to detect the event faster than a single source approach.

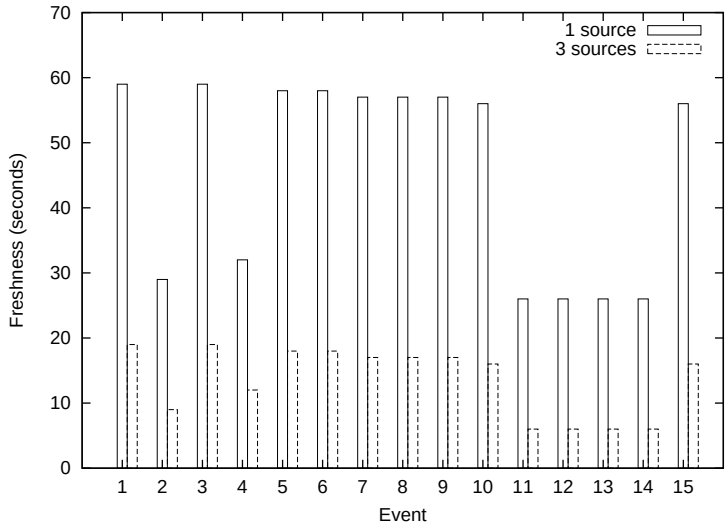

Figure 4.2: Freshness Per Event

Figure 4.3 shows the average freshness for each of the three sources alone and the average freshness when using all three combined. All three sources alone have a relatively high freshness difference compared to the baseline. But combined they achieve higher freshness, showing that our system can achieve higher freshness than other systems relying only on a single source. However, also here we must consider the fact that our scheduler will not be able to retrieve data from the source detecting the event first. And by that the average freshness will not be as good as in the best case scenario, although the potential is there.

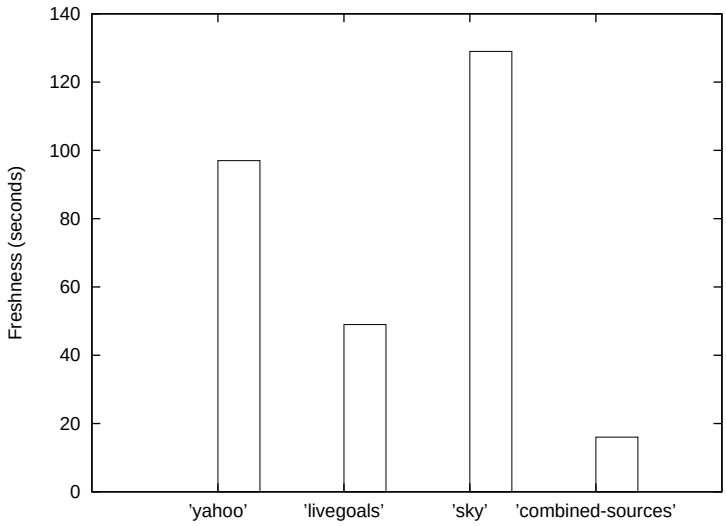

Figure 4.3: Average Freshness

Chapter 5

Conclusions

In this thesis we have addressed the problem of identifying and extracting video metadata from web sources. We have focused on combining high metadata freshness while at the same time adhering to general polling politeness rules. We proposed a solution using a pool of web sources containing overlapping information to improve the metadata freshness versus a single-source approach.

5.1 Concluding Remark

Our experiments have shown that we are able to identify and extract wanted metadata from several web sources, which can be further analyzed to retrieve structured information for annotating videos. The experiments also show that our approach is capable of retrieving metadata with higher freshness than any single-source approach. However, further improvement might be gained by implementing a more advanced scheduling algorithm than our round-robin.

5.2 Future Work

Future work would be to improve on our current scraping functionality. For instance, the duplicate detection code is not working perfectly well for sources with a higher comments rate than one per minute. A potential solution to this problem is to develop another more advanced algorithm for duplicate detection. To tackle HTML pages using javascript to automatically refresh pages, we would like to make some adjustments to the Scrapy framework. We would like to increase the source pool by identifying other possible web sources, and develop HTML content scrapers for these.

Further investigation into more advanced scheduling algorithms. For instance, taking advantage of source specific live update intervals, and adjust the scheduling based on the time each source is updated. It would also be

interesting to execute a large scale live system experiment, and address the problems that would occur in such a scenario. Finally it would be interesting to look into automatic event detection in the raw data material we used in our study. And by implementing that, be one step closer a system capable of delivering fresh event metadata for live video annotation.

References

[1] Brad Adelberg. Nodose—a tool for semi-automatically extracting structured and semistructured data from text documents. pages 283–294, 1998.

[2] Don Chamberlin Mary F. Fernandez Michael Kay Jonathan Robie Anders Berglund, Scott Boag and Jrme Simon. Xml path language (xpath) 2.0, December 2007.

[3] Gustavo O. Arocena and Alberto O. Mendelzon. Weboql: Restructuring documents, databases, and webs. pages 24–33, 1998.

[4] M. K. Bergman. *The Deep Web: Surfacing Hidden Value.*, 2001.

[5] Mary Elaine Califf and Raymond J. Mooney. Relational learning of pattern-match rules for information extraction. pages 328–334, 1999.

[6] Soumen Chakrabarti. Data mining for hypertext: a tutorial survey. *SIGKDD Explor. Newsl.*, 1(2):1–11, 2000.

[7] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific web resource discovery. volume 31, pages 1623–1640, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[8] Viv Cothey. Web-crawling reliability. *J. Am. Soc. Inf. Sci. Technol.*, 55(14):1228–1238, 2004.

[9] Valter Crescenzi and Giansalvatore Mecca. Grammars have exceptions. *Inf. Syst.*, 23(9):539–565, 1998.

[10] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. *RoadRunner: automatic data extraction from data-intensive web sites.* ACM, New York, NY, USA, 2002.

[11] D. W. Embley, D. M. Campbell, Y. S. Jiang, S. W. Liddle, D. W. Lonsdale, Y.-K. Ng, and R. D. Smith. Conceptual-model-based data extraction from multiple-record web pages. *Data Knowl. Eng.*, 31(3):227–251, 1999.

[12] Dayne Freitag. Machine learning for information extraction in informal domains. *Mach. Learn.*, 39(2-3):169–202, 2000.

[13] Joachim Hammer, Hector G. Molina, Junghoo Cho, Arturo Crespo, and Rohan Aranha. Extracting semistructured information from the web. 1997.

[14] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, pages 219–229, December 1999.

[15] Dag Johansen, Håvard Johansen, Tjalve Aarflot, Joseph Hurley, Åge Kvalnes, Cathal Gurrin, Sorin Zav, Bjørn Olstad, Erik Aaberg, Tore Endestad, Haakon Riiser, Carsten Griwidz, and Pål Halvorsen. Davvi: a prototype for the next generation multimedia entertainment platform. In *MM '09: Proceedings of the seventeen ACM international conference on Multimedia*, pages 989–990, New York, NY, USA, 2009. ACM.

[16] Nicholas Kushmerick. Wrapper induction: efficiency and expressiveness. *Artif. Intell.*, 118(1-2):15–68, 2000.

[17] Alberto H. F. Laender, Berthier Ribeiro-Neto, and Altigran S. da Silva. Debye - date extraction by example. *Data Knowl. Eng.*, 40(2):121–154, 2002.

[18] Alberto H. F. Laender, Berthier A. Ribeiro-Neto, Altigran S. da Silva, and Juliana S. Teixeira. A brief survey of web data extraction tools. *SIGMOD Rec.*, 31(2):84–93, 2002.

[19] Chunxi Liu, Qingming Huang, Shuqiang Jiang, Liyuan Xing, Qixiang Ye, and Wen Gao. A framework for flexible summarization of racquet sports video using multiple modalities. *Comput. Vis. Image Underst.*, 113(3):415–424, 2009.

[20] Jayant Madhavan, David Ko, Lucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Halevy. Google's deep web crawl. *Proc. VLDB Endow.*, 1(2):1241–1252, 2008.

[21] F. Menczer. ARACHNID: Adaptive Retrieval Agents Choosing Heuristic Neighborhoods for Information Discovery. In *Proc. 14th International Conference on Machine Learning*, pages 227–235, San Francisco, CA, 1997. Morgan Kaufmann.

[22] Filippo Menczer and Richard K. Belew. Adaptive information agents in distributed textual environments. pages 157–164, 1998.

[23] José E. Moreira, Maged M. Michael, Dilma Da Silva, Doron Shiloach, Parijat Dube, and Li Zhang. Scalability of the nutch search engine. pages 3–12, 2007.

[24] Ion Muslea, Steven Minton, and Craig A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):93–114, 2001.

[25] Alexandros Ntoulas, Petros Zerfos, and Junghoo Cho. Downloading textual hidden web content through keyword queries. pages 100–109, 2005.

[26] Berthier Ribeiro-Neto, Alberto H. F. Laender, and Altigran S. da Silva. Extracting semi-structured data through examples. pages 94–101, 1999.

[27] Eric Roberts, Russ Shackelford, Rich LeBlanc, and Peter J. Denning. Curriculum 2001: interim report from the acm/ieee-cs task force. *SIGCSE Bull.*, 31(1):343–344, 1999.

[28] Arnaud Sahuguet and Fabien Azavant. Building intelligent web applications using lightweight wrappers. *Data Knowl. Eng.*, 36(3):283–316, 2001.

[29] Dian Tjondronegoro, Yi-Ping Phoebe Chen, and Binh Pham. Sports video summarization using highlights and play-breaks. pages 201–208, 2003.

Appendix A

Source Code

A.1 Orchestrator

Listing A.1: "Orchestrator"

```
1 #!/usr/bin/env python
2
3 """
4     Orchestrator process for web data extraction system
5 """
6
7 import sys
8 import getopt
9 import datetime
10 import time
11 import logging
12 import orch_config
13 import subprocess
14 import MySQLdb
15
16 __author__      = "Tord Heimdal"
17 __copyright__   = "Copyright 2009, VMGF project"
18 __credits__     = ["Tord Heimdal"]
19 __license__     = "?"
20 __version__     = "beta"
21 __maintainer__ = "Tord Heimdal"
22 __email__       = "theimdal@gmail.com"
23 __status__      = "Development"
24
25 # Function that updates the given configurations with the last
       timestamp
26 # collected.
27 #
28 # source_configurations = list of id's for source configurations
29 #
30 def update_configs_with_timestamp(source_configurations):
31
32     # Connect to database and get cursor
33     conn = MySQLdb.connect(host="whitebox.td.org.uit.no", user="root",
           \
34                            passwd="kpax", db="master")
35     cursor = conn.cursor()
36
```

```
37      # Query and fetch last recorded timestamp
38      try:
39          cursor.execute("select * from comments ORDER BY timestamp DESC
                ")
40          result = cursor.fetchone()
41          last_timestamp = result[2]
42      except:
43          print "Error retrieving last_timestamp: ", sys.exc_info()[0]
44          return
45
46      # Update all configurations with the last found timestamp
47      for config in source_configurations:
48          try:
49              cursor.execute("UPDATE crawl_configuration SET
                    last_timestamp=%s WHERE id=%s", (last_timestamp, config
                    ))
50          except:
51              print "Error updating configuration: ", sys.exc_info()[0]
52              return
53
54      print "Configurations updated"
55
56      return
57
58  def main(argv):
59
60      # Set system duration
61      duration =  2 * 60 * 60 # in seconds
62
63      # Set source configurations for scheduling
64      source_configurations = [7, 8, 9]
65
66      # Set the wanted polling interval
67      polling_interval = 30 # in seconds
68
69      # Initiate count variable - which tells how many times we have
            polled for
70      # information.
71      count = 0
72
73      # Set the configuration index - which tells which index the
            scheduled
74      # source configuration is at.
75      conf_index = 0
76
77      # Main loop
78      # Executed for wanted duration with set polling interval
79      while count < duration/polling_interval:
80
81          # Record start time
82          startTime = time.time()
83
84          # Select the scheduled configuration
85          conf_id = source_configurations[conf_index]
86          print "CONFIG " + str(conf_id) + " scheduled"
87
88          # Execute the crawler and scraper as a subprocess
89          # Set the scheduled scrape configuration id as parameter
90          try:
91              subprocess.Popen(
92                  ["python " + orch_config.SCRAPY_SCRIPT_PATH + "scrapy-
                        ctl.py " + \
```

```
93                  "crawl match_spider" +\
94                  " --nolog " +\
95                  "--set CRAWL_ID='"+str(conf_id)+"'"
96                  ], shell=True)
97          except:
98              print "Error running scraper"
99
100         print "Scraping " + str(count) + " done"
101
102         # Invoke function updating configurations with the
                last_timestamp found
103         try:
104             update_configs_with_timestamp(source_configurations,
                    source_configurations[conf_index])
105         except :
106             print "Unexpected error:", sys.exc_info()[0]
107
108         # Round-Robin schedule the available source configurations
109         if conf_index < 2:
110             conf_index += 1
111         else:
112             conf_index = 0
113
114         print "Next config " + str(source_configurations[conf_index])
115
116         # Record stop time
117         stopTime = time.time()
118
119         # Update number of crawls
120         count += 1
121
122         # Set sleeping interval
123         timeTaken = stopTime - startTime
124         if polling_interval - timeTaken < 3:
125             time.sleep(polling_interval)
126         else:
127             time.sleep(polling_interval - timeTaken)
128
129     print "done crawling"
130     sys.exit()
131
132 if __name__ == '__main__':
133      main(sys.argv[1:])
```

Listing A.2: "Orchestrator configuration"

```
1  #!/usr/bin/env python
2
3  """
4      Module containing all configuration for the Orchestrator service
5  """
6
7  __author__      = "Tord Heimdal"
8  __copyright__   = "Copyright 2009, VMGF project"
9  __credits__     = ["Tord Heimdal"]
10 __license__     = "?"
11 __version__     = "beta"
12 __maintainer__ = "Tord Heimdal"
13 __email__       = "theimdal@gmail.com"
14 __status__      = "Development"
15
16 # Configuration constants
```

```
17
18 SCRAPY_SCRIPT_PATH   = "../scrapy/match_scraper/"
19 LOG_FILENAME         = "/home/tord/master/orchestrator/log.out"
20 MYSQL_SERVER_ADDRESS = "whitebox.td.org.uit.no"
```

A.2 Crawler, Scraper and Mysql Interface

Listing A.3: "Match scraper"

```
1
2 import sys
3 import re
4 from scrapy.contrib.spiders import CrawlSpider, Rule
5 from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor
6 from scrapy.selector import HtmlXPathSelector
7 from match_scraper.items import MatchScraperItem
8 from scrapy.conf import settings
9 import match_scraper.mysql_interface
10
11
12 class MatchSpider(CrawlSpider):
13
14     domain_name = "match_spider"
15
16     # Retrieve crawl id argument
17     crawl_id = str(settings['CRAWL_ID'])
18
19     # query for crawl configuration
20     try:
21         db = match_scraper.mysql_interface.MysqlInterface()
22         conf = db.fetch_configuration(crawl_id)
23     except Exception as inst:
24         print "Database connection error."
25         sys.exit(1)
26
27     # init configuration
28     match_id = conf[4]
29     source = conf[3]
30     seed_url = conf[2]
31     last_timestamp = conf[5]
32     last_source_timestamp = conf[6]
33     source_timestamp = conf[7]
34
35     print
           "========================================================================"
36     print "CONFIGURATION"
37     print
           "========================================================================"
38     print "source:   " + source
39     print "seed url: " + seed_url
40     print
           "========================================================================"
41
42     seed = "http://localhost/" + seed_url
43     start_urls = [ seed ]
44
45     # evaluate current timestamp against last_timestamp and
           last_source_timestamp
```

```
46      def evaluate_timestamp(self, current, last_timestamp,
            last_source_timestamp):
47
48          # compare timestamp to last gathered global comment and last
                gathered
49          # source comment
50          if(len(current) <= 2):
51              if(int(current) < int(last_timestamp) or int(current) <=
                    int(last_source_timestamp)):
52                  return True
53          else:
54              if(int(current[0:2]) < int(last_timestamp) or int(current)
                    <= int(last_source_timestamp)):
55                  return True
56
57          return False
58
59      # dispatch page to correct parser
60      def parse_start_url(self, response):
61
62          # if yahoo
63          if(self.source == "yahoo"):
64              return self.yahoo(response)
65          # if sky
66          elif(self.source == "sky"):
67              return self.sky(response)
68          # if livegoals
69          else:
70              return self.livegoals(response)
71
72
73      # LIVEGOALS parser
74      def livegoals(self, response):
75          hxs = HtmlXPathSelector(response)
76          items = []
77          item = MatchScraperItem()
78
79          # Add identification info
80          item['crawl_id'] = self.crawl_id
81          item['match_id'] = self.match_id
82          item['source'] = self.source
83          item['source_timestamp'] = self.source_timestamp
84
85          # scrape summary information
86          try:
87              score = hxs.select('//*[@id="event-result"]/text()').
                    extract()[0]
88              item['home_score'] = score[0]
89              item['away_score'] = score[2]
90          except:
91              pass
92              print "Skipped summary info due to error in parsing."
93
94          # scrape match comments
95          comments = []
96          commentList = hxs.select('//*[@id="Commentary-Contents"]')
97
98          for comment in commentList.select('.//tr'):
99
100             # error handling
101             try:
102
```

```
103                 tmp = {}
104
105                 # extract time information from class variable u'min0
                        comment380950033'
106                 timeString = comment.select('td[1]/text()').extract()
                        [0]
107                 timeString = re.sub("\D", "", timeString)
108
109                 if len(timeString) == 3:
110                     tmp["timestamp"] = timeString[0:2]
111                 else:
112                     tmp["timestamp"] = timeString
113
114                 skip_comment =  self.evaluate_timestamp(tmp["timestamp
                        "], self.last_timestamp, self.last_source_timestamp
                        )
115
116                 if skip_comment:
117                     continue
118
119                 # extract comment
120                 try:
121                     tmp["comment"] = comment.select('td[3]/text()').
                            extract()[0]
122                 except:
123                     tmp["comment"] = comment.select('td[3]/strong/text
                            ()').extract()[0]
124
125                 # append the new event to the comments list
126                 comments.append(tmp)
127
128             except:
129                 pass
130                 print "Skipped comment due to error in parsing"
131
132             item['comments'] = comments
133
134         items.append(item)
135
136         return items
137
138     # SKY parser
139     def sky(self, response):
140         hxs = HtmlXPathSelector(response)
141         items = []
142         item = MatchScraperItem()
143
144         # Add identification info
145         item['crawl_id'] = self.crawl_id
146         item['match_id'] = self.match_id
147         item['source'] = self.source
148         item['source_timestamp'] = self.source_timestamp
149
150         # scrape summary information
151         try:
152             score = hxs.select('/html/body/div/div[9]/div[1]/div/div/
                        table/tbody/tr/td[2]/text()').extract()[0]
153             score = score.replace('\t', '').replace('\n', '').replace('
                        ', '')
154             item['home_score'] = score[0]
155             item['away_score'] = score[2]
156         except:
```

```
157             print "Skipped summary info due to error in parsing."
158             pass
159
160         # scrape match comments
161         comments = []
162         commentList = hxs.select('/html/body/div/div[9]/div[3]/div/div/
                span/div[4]')
163
164         for comment in commentList.select('.//p'):
165
166             # error handling
167             try:
168
169                 tmp = {}
170
171                 # extract time information from class variable u'min0
                        comment380950033'
172                 timeString = comment.select('b/text()').extract()[0]
173                 tmp["timestamp"] = timeString.replace(' ', '')
174
175                 skip_comment =  self.evaluate_timestamp(tmp["timestamp
                        "], self.last_timestamp, self.last_source_timestamp
                        )
176
177                 if skip_comment:
178                     continue
179
180                 # test to check if the <p> is a real comment - skip it
                        if not
181                 try:
182                     cast = int(tmp["timestamp"][0])
183                 except:
184                     print "Skipped comment without time reference"
185                     raise
186
187                 # extract comment
188                 tmp["comment"] = comment.select('text()').extract()[0]
189
190                 # append the new event to the comments list
191                 comments.append(tmp)
192
193             except:
194                 print "Skipped comment due to error in parsing"
195                 pass
196
197
198             item['comments'] = comments
199
200         items.append(item)
201
202         return items
203
204
205     # YAHOO parser
206     def yahoo(self, response):
207         hxs = HtmlXPathSelector(response)
208         items = []
209         item = MatchScraperItem()
210
211         # Add identification info
212         item['crawl_id'] = self.crawl_id
213         item['match_id'] = self.match_id
```

```
214         item['source'] = self.source
215         item['source_timestamp'] = self.source_timestamp
216 
217         # scrape summary information
218         try:
219             item['home_score'] = hxs.select('/html/body/div/div[2]/div/
                    div/div[2]/div[2]/h3/span[1]/text()').extract()[0]
220             item['away_score'] = hxs.select('/html/body/div/div[2]/div/
                    div/div[2]/div[2]/h3/span[3]/text()').extract()[0]
221         except:
222             print "Skipped summary info due to error in parsing."
223             pass
224 
225 
226         # scrape match comments
227         comments = []
228         commentList = hxs.select('/html/body/div/div[2]/div/div/ul/li
                [1]/div/div[2]/ul/li')
229 
230         for comment in commentList:
231             try:
232                 tmp = {}
233 
234                 # extract time information from class variable u'min0
                        comment380950033'
235                 timeString = comment.select("@class").extract()[0]
236                 tmp["timestamp"] = timeString[:5].replace("min", "").
                        replace(" ", "")
237 
238                 skip_comment =  self.evaluate_timestamp(tmp["timestamp
                        "], self.last_timestamp, self.last_source_timestamp
                        )
239 
240                 if skip_comment:
241                     continue
242 
243                 # skip event at time 0
244                 if tmp["timestamp"] == "0":
245                     continue
246 
247                 # extract comment from span[2] text value
248                 tmp["comment"] = comment.select("span[2]/text()").
                        extract()[0]
249 
250                 # append the new event to the comments list
251                 comments.append(tmp)
252             except:
253                 print "Skipped comment due to error in parsing"
254                 pass
255 
256 
257         item['comments'] = comments
258 
259         items.append(item)
260 
261         return items
262 
263 SPIDER = MatchSpider()
```

Listing A.4: "Match item"

```
1 # Define here the models for your scraped items
```

```
2 #
3 # See documentation in:
4 # http://doc.scrapy.org/topics/items.html
5
6 from scrapy.item import Item, Field
7
8 class MatchScraperItem(Item):
9
10     # crawl id
11     crawl_id = Field()
12
13     # match id
14     match_id = Field()
15
16     # item source
17     source = Field()
18
19     # source timestamp
20     source_timestamp = Field()
21
22     # number of goals scored by home team
23     home_score = Field()
24
25     # number of goals scored by away team
26     away_score = Field()
27
28     # all live comments
29     comments = Field()
30
31     pass
```

Listing A.5: "Item pipeline"

```
1 # Pipeline storing items in database
2
3 import mysql_interface
4
5 class MatchScraperPipeline(object):
6
7     def process_item(self, domain, item):
8
9         crawl_id = item['crawl_id']
10        match_id = item['match_id']
11        source = item['source']
12        home_score = item['home_score']
13        away_score = item['away_score']
14        comments = item['comments']
15        source_timestamp = item['source_timestamp']
16
17        # connect to database
18        db = mysql_interface.MysqlInterface()
19
20        # update match score
21        db.update_match_score(match_id, home_score, away_score)
22
23        # update match comments
24        i = 0
25        for comment in item['comments']:
26            db.add_match_comment(match_id, comment['timestamp'], "blank
                  ", \
27                                 comment['comment'], source,
                                       source_timestamp)
```

```
28
29              if i == 0:
30                  last_timestamp = comment['timestamp']
31                  i += 1
32
33          # update last comment for current source
34          db.update_last_timestamp(crawl_id, last_timestamp)
35
36          return item
```

Listing A.6: "Scraper settings"

```
1  # Scrapy settings for match_scraper project
2  #
3  # For simplicity, this file contains only the most important settings
       by
4  # default. All the other settings are documented here:
5  #
6  #     http://doc.scrapy.org/topics/settings.html
7  #
8  # Or you can copy and paste them from where they're defined in Scrapy:
9  #
10 #     scrapy/conf/default_settings.py
11 #
12
13 import match_scraper
14
15 BOT_NAME = 'match_scraper'
16 BOT_VERSION = '1.0'
17
18 SPIDER_MODULES = ['match_scraper.spiders']
19 NEWSPIDER_MODULE = 'match_scraper.spiders'
20 DEFAULT_ITEM_CLASS = 'match_scraper.items.MatchScraperItem'
21 USER_AGENT = '%s/%s' % (BOT_NAME, BOT_VERSION)
22 ITEM_PIPELINES = ['match_scraper.pipelines.MatchScraperPipeline']
```

Listing A.7: "Mysql interface"

```
1  import MySQLdb
2
3  class MysqlInterface():
4
5      # constructor
6      def __init__(self):
7          self.conn = MySQLdb.connect(host="whitebox.td.org.uit.no", user
               ="root",\
8                                      passwd="kpax", db="master")
9          self.cursor = self.conn.cursor()
10         pass
11
12     # method that returns last timestamp for all comments
13     def get_last_timestamp(self):
14         try:
15             self.cursor.execute("select * from comments ORDER BY \
16                                 timestamp DESC")
17             return self.cursor.fetchone()
18         except:
19             print "Error in get_last_timestamp: ", sys.exc_info()[0]
20
21     # Method that updates a given crawl configuration with the given
           timestamp
```

```
22      def update_last_timestamp(self, crawl_id, timestamp):
23          try:
24              self.cursor.execute("UPDATE crawl_configuration SET\
25                                   last_source_timestamp=%s WHERE id=%s",
                                         \
26                                   (timestamp, crawl_id))
27          except:
28              print "Error updating last timestamp"
29
30      # Method that returns a given crawl configuration
31      def fetch_configuration(self, crawl_id):
32          try:
33              self.cursor.execute("select * from crawl_configuration
                     where \
34                                   id =%s", (crawl_id,))
35              return self.cursor.fetchone()
36          except:
37              print "Mysql error when retrieving configuration"
38
39      # Method that updated a specific match with the given home and away
             goals
40      def update_match_score(self, match_id, home, away):
41          try:
42              self.cursor.execute("UPDATE master.match SET \
43                                   home_score=%s, away_score=%s WHERE id=%
                                         s", \
44                                   (home, away, match_id))
45          except:
46              print "Mysql error when updating match score"
47
48          pass
49
50      # Method that inserts a given comment data into the database
51      def add_match_comment(self, match_id, timestamp, event, comment, \
52                            source, source_timestamp):
53          try:
54              self.cursor.execute("INSERT INTO comments (match_id,
                     timestamp, \
55                                   event, comment, source,
                                         source_timestamp) \
56                                   VALUES(%s, %s, %s, %s, %s, %s)",\
57                                   (match_id, timestamp, event, comment,
                                         source, \
58                                    source_timestamp))
59          except:
60              print "Mysql error when adding match comment"
61
62          pass
```

A.3 Experiment

Listing A.8: "Experiment script"

```
1
2 #!/usr/bin/env python
3
4 """
5     Experiment script
6 """
7
8 import sys
```

```
 9 import os
10 import getopt
11 import datetime
12 import time
13 import logging
14 import orch_config
15 import subprocess
16 import MySQLdb
17
18 __author__     = "Tord Heimdal"
19 __copyright__  = "Copyright 2009, VMGF project"
20 __credits__    = ["Tord Heimdal"]
21 __license__    = "?"
22 __version__    = "beta"
23 __maintainer__ = "Tord Heimdal"
24 __email__      = "theimdal@gmail.com"
25 __status__     = "Development"
26
27
28 # Funcion parsing and retrieving slurp stats
29 def retrieve_slurp_stats():
30     # change to www dir
31     os.chdir("/home/tord/master/www/")
32
33     # dictionaries for slurp information
34     yahoo = {}
35     sky = {}
36     livegoals = {}
37
38     # create yahoo dict
39     f = open("yahoo_stat.dat")
40     i = 1
41     for line in f.readlines():
42         if i == 1:
43             i = 2
44             continue
45         run = line.split(" ")
46         yahoo[run[0]] = {'start': run[1], 'stop': run[2], 'html': run
               [3]}
47
48     f.close()
49
50     # create sky dict
51     f = open("sky_stat.dat")
52     i = 1
53     for line in f.readlines():
54         if i == 1:
55             i = 2
56             continue
57         run = line.split(" ")
58         sky[run[0]] = {'start': run[1], 'stop': run[2], 'html': run[3]}
59
60     f.close()
61
62     # create livegoals dict
63     f = open("livegoals_stat.dat")
64     i = 1
65     for line in f.readlines():
66         if i == 1:
67             i = 2
68             continue
69         run = line.split(" ")
```

```
70          livegoals[run[0]] = {'start': run[1], 'stop': run[2], 'html':
                run[3]}
71
72      f.close()
73
74      return [yahoo, sky, livegoals]
75
76  # Function that updates the given configurations with the last
        timestamp
77  # collected.
78  #
79  # source_configurations = list of id's for source configurations
80  #
81  def update_configs_with_timestamp(source_configurations):
82
83      # Connect to database and get cursor
84      conn = MySQLdb.connect(host="whitebox.td.org.uit.no", user="root",
            \
85                             passwd="kpax", db="master")
86      cursor = conn.cursor()
87
88      # Query and fetch last recorded timestamp
89      try:
90          cursor.execute("select * from comments where match_id=7 and
                source='livegoals' ORDER BY timestamp DESC")
91          result = cursor.fetchone()
92          last_timestamp = result[2]
93      except:
94          print "Error retrieving last_timestamp: ", sys.exc_info()[0]
95          return
96
97      # Update all configurations with the last found timestamp
98      for config in source_configurations:
99          try:
100             cursor.execute("UPDATE crawl_configuration SET \
101                             last_timestamp=%s WHERE id=%s", \
102                             (last_timestamp, config))
103         except:
104             print "Error updating configuration: ", sys.exc_info()[0]
105             return
106     return
107
108 # Function that insert the correct seed url into the given
        configuration
109 def update_configuration(count, conf_id, slurp):
110
111     # Connect to database and get cursor
112     conn = MySQLdb.connect(host="whitebox.td.org.uit.no", user="root",
            \
113                             passwd="kpax", db="master")
114     cursor = conn.cursor()
115
116     # Update specified configuration with new seed url
117     new_seed = slurp[str(count)]['html'].replace("\n", '')
118     source_timestamp = slurp[str(count)]['stop']
119     try:
120         cursor.execute("UPDATE crawl_configuration SET seed_url=%s, \
121                         current_source_timestamp=%s WHERE id=%s", \
122                         (new_seed, source_timestamp[0:10],conf_id))
123     except:
124         print "Error updating configuration with new seed: ", sys.
            exc_info()[0]
```

```
125
126     return
127
128 def main(argv):
129
130     # Set source configurations for scheduling
131     source_configurations = [12]
132
133     # controll the number of iterations done
134     count = 0
135
136     # Set the configuration index - which tells which index the
            scheduled
137     # source configuration is at.
138     conf_index = 0
139
140     # EXPERIMENT STATS
141     slurp_stats = retrieve_slurp_stats()
142
143     # Main loop
144     # Executed for wanted duration with set polling interval
145     while count < 231:
146
147         # Select the scheduled configuration
148         conf_id = source_configurations[conf_index]
149
150         # update configuration with correct html version
151         update_configuration(count, conf_id, slurp_stats[2])
152
153         # Execute the crawler and scraper as a subprocess
154         # Set the scheduled scrape configuration id as parameter
155         retval = subprocess.call(
156             ["python " + orch_config.SCRAPY_SCRIPT_PATH + "scrapy-ctl.
                    py " + \
157              "crawl match_spider" +\
158              " --nolog " +\
159              "--set CRAWL_ID='"+str(conf_id)+"'"
160              ], shell=True)
161
162         print "RETURN VALUE: " + str(retval)
163
164         # Invoke function updating configurations with the
                last_timestamp found
165         try:
166             update_configs_with_timestamp(source_configurations)
167         except :
168             print "Error updating configs with last timestamp"
169         count += 1
170
171 if __name__ == '__main__':
172      main(sys.argv[1:])
```

Listing A.9: "Yahoo polling script"

```
1 # -*- coding: utf-8 -*-
2 #!/usr/bin/env python
3 import urllib2, time, os.path, os
4
5
6 # Config
7 dest = "yahoo_liverpool_arsenal"
8 name = "yahoo"
```

```
src = "http://uk.eurosport.yahoo.com/football/premier-league/2009-2010/
    liverpool-arsenal-282896.html"
length =  2 * 60 * 60 # in seconds
wait = 30
###########

try:
    os.mkdir(dest)
except:
    pass


statfile = file(os.path.join(dest,"stat.dat"), "w")
statfile.write("#filename start stop\n")

count = 0
error = False
while count < length/wait:

    startTime = time.time()
    try:
        u = urllib2.urlopen(src)
        fn = "%s-%d-%f.html" % (name, count , time.time())
        f = file(os.path.join(dest, fn), "w")
        data = u.read()
        f.write(data)
        f.close()
    except:
        error = True
        print time.ctime(), "error downloading page nr ", count
    finally:
        stopTime = time.time()
        if not error:
            statfile.write("%d %f %f %s\n" % (count, startTime,
                stopTime, fn))
            print time.ctime(), "downloaded nr ", count
        else:
            error = False
        count += 1

    timeTaken = stopTime - startTime
    if wait - timeTaken < 3:
        time.sleep(wait)
    else:
        time.sleep(wait - timeTaken)
```

Listing A.10: "Sky polling script"

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python
import urllib2, time, os.path, os


# Config
dest = "sky_liverpool_arsenal"
name = "sky"
src = "http://www.skysports.com/football/match_commentary/0,19764,11065
    _3205392,00.html"
length =  2 * 60 * 60 # in seconds
wait = 30
###########

```

```
14 try:
15     os.mkdir(dest)
16 except:
17     pass
18
19
20 statfile = file(os.path.join(dest,"stat.dat"), "w")
21 statfile.write("#filename start stop\n")
22
23 count = 0
24 error = False
25 while count < length/wait:
26
27     startTime = time.time()
28     try:
29         u = urllib2.urlopen(src)
30         fn = "%s-%d-%f.html" % (name, count , time.time())
31         f = file(os.path.join(dest, fn), "w")
32         data = u.read()
33         f.write(data)
34         f.close()
35     except:
36         error = True
37         print time.ctime(), "error downloading page nr ", count
38     finally:
39         stopTime = time.time()
40         if not error:
41             statfile.write("%d %f %f %s\n" % (count, startTime,
                   stopTime, fn))
42             print time.ctime(), "downloaded nr ", count
43         else:
44             error = False
45         count += 1
46
47     timeTaken = stopTime - startTime
48     if wait - timeTaken < 3:
49         time.sleep(wait)
50     else:
51         time.sleep(wait - timeTaken)
```

Listing A.11: "Livegoals polling script"

```
1 # -*- coding: utf-8 -*-
2 #!/usr/bin/env python
3 import urllib2, time, os.path, os
4
5
6 # Config
7 dest = "livegoals_liverpool_arsenal"
8 name = "livegoals"
9 src = "http://www.livegoals.com/gamecenter/liverpool-vs-arsenal
      -13-12-2009"
10 length =  2 * 60 * 60 # in seconds
11 wait = 30
12 ############
13
14 try:
15     os.mkdir(dest)
16 except:
17     pass
18
19
```

```
20 statfile = file(os.path.join(dest,"stat.dat"), "w")
21 statfile.write("#filename start stop\n")
22
23 count = 0
24 error = False
25 while count < length/wait:
26
27     startTime = time.time()
28     try:
29         u = urllib2.urlopen(src)
30         fn = "%s-%d-%f.html" % (name, count , time.time())
31         f = file(os.path.join(dest, fn), "w")
32         data = u.read()
33         f.write(data)
34         f.close()
35     except:
36         error = True
37         print time.ctime(), "error downloading page nr ", count
38     finally:
39         stopTime = time.time()
40         if not error:
41             statfile.write("%d %f %f %s\n" % (count, startTime,
                   stopTime, fn))
42             print time.ctime(), "downloaded nr ", count
43         else:
44             error = False
45         count += 1
46
47     timeTaken = stopTime - startTime
48     if wait - timeTaken < 3:
49         time.sleep(wait)
50     else:
51         time.sleep(wait - timeTaken)
```