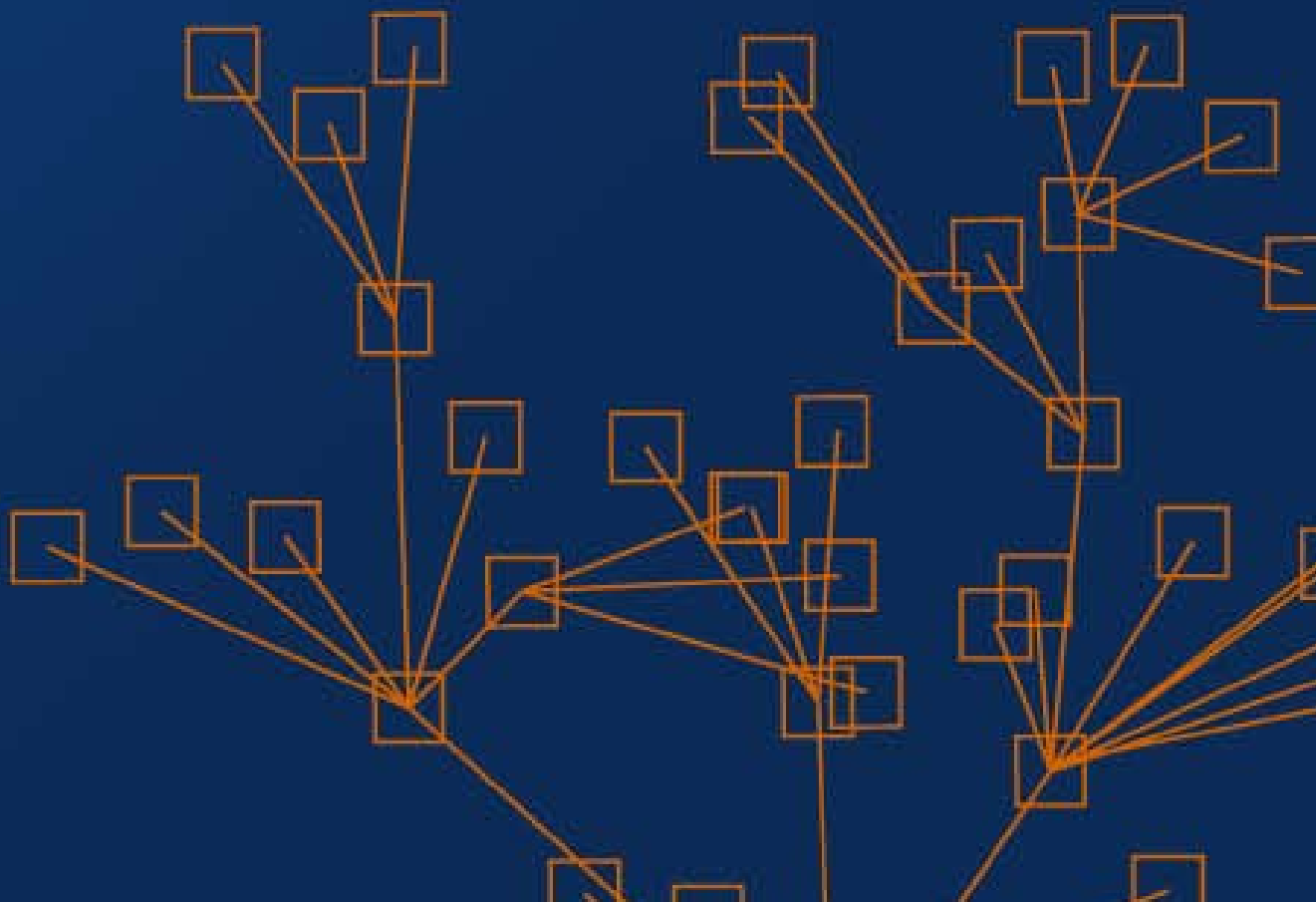
netpreserve.org
international internet preservation consortium

Prototypes related to IIPC Access Working Group Use Cases

IIPC Access Working Group

May 2006 | Version 1



1 Introduction

The IIPC use cases document illustrates that a web archive has many types of users and that several methods for access are needed. The discussions in the IIPC Access WG indicate that different archives want specific features in their access tools (although some central features are wanted by all). This led to the suggestion that the WG develop a common architecture of many small, interacting components, each of which encapsulates one clearly defined functionality. Such an architecture would allow each archive to concentrate its efforts on the features that it finds most important. The biggest challenge would be to converge towards a set of interfaces that individual components should adhere to in order for the components to work in different archives. The fewer the interfaces, and the simpler each interface is, the better.

It was suggested that the WG investigated what interfaces may be needed by developing mock-up prototypes of the functional components suggested by the use cases. A prototype need not actually perform (even part of) its intended function; a mock-up prototype for a text search engine might, for instance, always return the same list of results regardless of its input. What is important is that the programming interface seems reasonable and is well described (e.g. how do you ask for only the top 10 results?) including a rationale (e.g. why should search terms be UTF-8 encoded?).

Each mock-up prototype (along with descriptions) was not supposed to take more than a few days to develop, and the implementers were not expected to spend their time on coordinating interfaces (so that the prototypes integrate at this point of time). Once these prototypes had been worked out there should be available excellent material for discussing a set of common interfaces.

It was strongly recommend that the prototypes be written as actual, compiling code even if the code does no real work. This ensures that all aspects of an interface down to the concrete level have been considered. If everybody developed in the same programming language (e.g. Java) that would be splendid, but at the prototype stage it is not necessary to have a strictly mandatory, common platform.

The selected prototypes in many cases proved to be very difficult to implement and some have not been implemented at all. Despite this it was of great value to make the effort because it provoked valuable deliberations and discussions that greatly helped in defining many of the concepts and issues that the IIPC members are looking at.

This applies to all aspects of web archiving, i.e. collection policy, harvesting and access.

This document is the result of work and discussions by the members of the IIPC Access Group. The draft was written by Niels H. Christensen of the Royal Library of Denmark.

Table of Contents

1 Introduction 1
2 Components to be prototyped 3
3 Prototype A: Select a List of Dates and Present Difference 4
3.1 Some questions to be addressed 4
3.2 Goals 4
3.3 Prerequisites 5
3.4 Comparator Interface 5
3.5 Difference Report 6
3.6 Comparator Deployment and Use 8
3.7 Specific Comparators 9
3.8 Simple bitwise comparators (hashing) 10
3.9 "Roughly the same" comparators 10
3.10 Support tools 11
3.11 Summary of comparators 11
3.12 Future directions 12
4 Prototype B: Show linking information 13
4.1 Some questions to be addressed: 13
4.2 Item Link Information 13
4.3 Link Extractor 13
4.4 Link Analyzer 14
4.5 Query Tool 14
4.6 Some questions to be addressed 15
5 Prototype C: Define Segment 17
5.1 Some questions to be addressed: 17
5.2 Definitions 17
7. Prototype D 20
6 Prototype D: Statistical Information about a Segment 22
6.1 Some questions to be answered 22
7 Prototype E: Process Objects 24
7.1 Some questions to be addressed: 24
7.2 Implementation 24
8 Prototype F: Format Distribution in a Segment 27
8.1 Some questions to be addressed 27
8.2 Solution 1 27
8.3 Solution 2 29
9 Prototype G 30
9.1 Some questions to be addressed 30
9.2 Method 30
10 Prototype H: On demand harvesting 31
10.1 Some questions to be addressed 31
10.2 Definitions 31

2 Components to be prototyped

Like many other systems, the access tools described in the use cases can be divided into three layers: The presentation layer, the logical layer and the data layer. The use cases mostly describe features of the presentation layer (e.g. entering search terms and pressing a search button) but some features of the other layers are easily inferred (e.g. grouping URL lists by domain in the logical layer or returning the byte size of an object in the data layer). The prototypes focus on the logical layer of a feature rather than the presentation layer (direct user interaction, boxes, text fields, buttons etc.) or the data layer (reading ARC files, reading record metadata from file etc.).

In the detailed description of the prototypes the acronym API (Abstract Programming Interface) is used extensively. This is because the prototypes are all about interface decisions rather than actual implemented functionality.

The prototypes are listed in the following table:

Task-ID	Identified Tasks	Based on Use Cases
A	Select a List of Dates and Present Difference	4.3, 4.8.2 and 4.8.3
B	Show linking information	4.3 and 5.1.2
C	Define Segment	7.1.1 7.1.2, 7.1.3 and 7.2.1
D	Statistical Information about a Segment	7.1, 7.2, 7.3 and 7.4
E	Process Objects	7.1.2, 7.1.3, 7.2.1, 7.2.2 and 7.4.1
F	Format Distribution in a Segment	7.2.1, 7.2.2 and 7.3.1
G	Save Search Results for Reuse	4.5
H	On Demand Harvesting	4.11

3 Prototype A: Select a List of Dates and Present Difference

This prototype is motivated by use cases 4.3, 4.8.2 and 4.8.3.

The provided prototype should have an API that represents a list of dates. The dates are associated with a given URL. Conceptually, these are the ingest dates of the archived versions of that URL object. The prototype should also have an API representing the differences between two versions of a given URL object. It should be possible to get a difference representation by picking two dates from a list.

3.1 Some questions to be addressed

What is the most convenient level of granularity for dates in a web archive (days, seconds, milliseconds, variable)? Should the date list API tell if the given URL object actually changed between two consecutive dates (like the *'s in the Wayback Machine)? Should the date list API include functionality for finding the "nearest date" to a given date (so the one can look up the version nearest to a wanted date)? How should differences between two "binary objects" be represented? How should differences between two text objects be represented?

3.2 Goals

The intent is to define requirements for a basic set of tools for comparing documents in web archive collections. The output of these comparisons can be used in access tools in a number of ways, including:

- Displaying only distinct versions of a repeatedly harvested document, instead of redundant identical items.
- Providing insight into a document's changes.
- Providing a base for more complex analysis of change across a site, or a repeated crawl.

The scope is limited to defining a core set of utilities, which can be used in support of an access tool suite, not addressing any issues of user interface or of presentation of the changes once detected.

3.3 Prerequisites

A prerequisite for this set of tools is some low-level access to an archive, providing a way to retrieve two items for comparison. Neither the selection of the two particular items to compare, nor the mechanism for their retrieval from the archive, is within the scope.

The existence of some interface for obtaining items from the archive is assumed, for which a trivial pseudo code1 interface might resemble:

```
interface ArchiveItem {
    // the item's metadata, as key/value pairs
    List<Pair<String, String>> getMetadata();

    // read the item's data
    InputStream    getContents();
}
```

The specific interface is immaterial, as long as it provides some way of accessing the item and its metadata, either of which might be important for comparison.

3.4 Comparator Interface

A comparator is a tool that, given two items in a collection, returns a report describing the changes between them. By defining a standard interface to comparators, we may implement various comparison approaches, and select the most appropriate implementation depending on need.

Any particular comparator may use any approach at all in comparing the two items, and its particular algorithm will likely not be equally applicable to every kind of items. A comparator can indicate whether or not it is able to compare two given items, and how well it is likely to perform the job. Specific comparators would make the decision based on attributes of the items such as their content type; a comparator best suited for plain text documents might return an indicator of "HIGH" applicability for two files of MIME type "text/plain", but only "MEDIUM" applicability for two XML documents, and "NOT APPLICABLE" if provided with two JPEG images.

No particular guarantee is made regarding what aspects of an object the comparator considers; one comparator may examine the entire body of the items, while another comparator examines only the first 100KB of the items, and a third might only consider a checksum found in the item's metadata.

[1] "Pseudo code" implying it's neither Java nor C++, including some of the less appealing features of both while remaining un-compileable.

After a comparison is performed, the comparator is able to provide a difference report in some machine-readable form, suitable for post-processing.

A comparator's interface, in pseudo code, might resemble:

```
interface Comparator {
    // distinguish this comparator with a unique name describing its
approach
    String getName();

    // provide comparator-specific configurations
    void configure(String name, String value);
    List<pair<String, String>> getConfiguration();

    enum Applicability { NOT=0, LOW, MEDIUM_LOW, MEDIUM, MEDIUM_HIGH, HIGH
}
    // indicate how effective this comparator believes it can be at
    // comparing the given items
    Applicability isApplicable(ArchiveItem firstItem,
                               ArchiveItem secondItem);

    // by examining the body and/or metadata of the supplied items,
    // describe the changes from the first item to the second
    DifferenceReport compare(ArchiveItem firstItem,
                             ArchiveItem secondItem);
}
```

3.5 Difference Report

The result of a comparator's operation is a report of the differences between the two supplied documents. The contents of this report will contain some standard forms of information, so that clients of the comparators may present the same interface to users, regardless of the particular comparison approach used. In addition, the report will contain whatever information the particular comparator provides about its specific approach.

The common aspects to all difference reports will include:

An enumerated indication of the degree of change between the first item and the second, normalized to one of the following values:

- No change (identical items)
- Minor change
- Moderate change
- Major change
- Complete change (no relation)

A comparator-specific quantified measurement of change, normalized to a decimal in the range of [0..1] (or, synonymously, 0-100%), as well as a textual description of the kind of change. The numeric indicator will not necessarily be comparable between

different comparators, other than that the specific value 0 should indicate absolute confidence of identicality and that the value 1 should indicate complete change, according to the comparator's particular criteria. The scale may vary according to whatever criteria the comparator uses for assessing change; it need not be linear, nor must it distinguish with any particular level of precision. 2

If possible, a list of regions where changes occurred from the first document to the second, indicated by the corresponding byte ranges within the two items. The meaning of these ranges would depend on the type of comparator, and may not be applicable for some comparator approaches.

As an example of the regions of change, consider the trivial document "ABC", a changed version "ADC", and a simple text comparator. The changed region would be at offset 1, of length 1, in both the source and the original document (representing "B" -> "D"). Depending on the change algorithm, note that the regions in two documents may not have the same starting offset or length. The idea of a list of changed regions may not be applicable to a given comparator, and so may be empty, or may indicate, with the same meaning, that the entire document is changed.

Identification of the comparator's name and configuration, for documentation.

In addition, a report of any additionally available comparator-specific information may be supplied, if relevant.

Thus a reasonable interface might be:

```
interface DifferenceReport {

    enum ChangeLevel { NONE=0, MINOR, MODERATE, MAJOR, COMPLETE }
    // indicate how much change occurred, normalized
    ChangeLevel getChangeAmount();

    // a comparator-specific measure of change, constrained to range
[0..1]
    float getChangeFactor();
    // a comparator-specific description of the change type
    String getChangeType();

    // a portion of a document, starting at the offset, of the given
length
    class ByteRange {
       long int offset, length;
    }

    // the corresponding regions in two documents being compared
```

[2] In other words, should two different comparators both indicate a change level of 50%, the only certainty is that neither comparator was able to definitively state that the items were either identical or entirely different.

```
    class ChangedRegion {
       ByteRange from, to;
    }

    // if applicable, a list of corresponding zones of change
    List<ChangedRegion> getChangedRegions();

    // a comparator-specific report of change details
    String getChangeReport();

    // handles to the items being compared
    ArchiveItem from();
    ArchiveItem to();

    // the comparator that generated this report
    String getComparatorName();

    // the configuration for the comparator that generated this report
    List<Pair<String, String>> getComparatorConfiguration();
};
```

3.6 Comparator Deployment and Use

Given the above description of aspects common to all comparators, an analysis environment or access tool should support deploying multiple comparators that provide this interface, each comparator best suited for particular content types or comparison approaches. By some means, a system would detect the desired set of comparator implementations installed within an environment, and load any supplied configuration for each.

Having loaded all appropriate comparators, a system would, upon being invoked for a comparison between two specific items, query each comparator for its applicability to those two items. A system would then select one or more of the most applicable comparators, run a comparison using those comparators, and retrieve the difference report from each comparator.

In some cases, only a portion of the difference report may actually be useful. In the example of a simple change indicator flag, used when displaying two consecutive documents in an access tool, the difference report might be used solely to determine whether or not two instances of a document were different, producing a single Boolean flag and ignoring any of the report details. Another context, however, might use the changed ranges contained in the report to produce a "redlined" view, displaying changed sections in-line with deletions crossed out or additions highlighted. Still another case might use known features of a particular comparator to generate a custom report, and then integrate that report into another context.

An interface for a dispatcher that selects appropriate comparators might resemble:

```
interface ComparatorDispatcher {
```

```
    // returns the single most applicable comparator for the given two
    // items, ordered by applicability
    Comparator getBestComparator(ArchiveItem from, ArchiveItem to);

    // returns, ordered by their applicability, all comparators able to
    // handle the given items with an Applicability greater than or
    // equal to the given level.
    List<Comparator> getRelevantComparators(
        ArchiveItem from,
        ArchiveItem to,
        Comparator.Applicability applicability=Comparator.LOW);

    // add a new comparator, located by some means, to the list of
    // known comparators
    void installComparator(Comparator newComparator);
}
```

3.7 Specific Comparators

There are essentially two, somewhat related, problems when it comes to evaluating changes between two documents. The first relates to the nature of change. That is, exactly when has a document changed? The simplest form of change detection is a straight bit wise comparison, usually implemented via strong hashing. This assumes that any change in the sequence of bits compromising a document equates to a change in the document.

While this is, to some degree, true, it fails to address the fact that most modern document contain a significant amount of irrelevant, redundant and/or superficial data. Most notable, is layout information, such as that used in HTML documents. This layout information can frequently account for the majority of the document's size and can often be modified significantly without affecting the actual document content at all. More specialized document formats, such as MS-Word documents, may also contain all sorts of extraneous information such as revision history etc.

To tackle this, it is necessary to either abstract the content before doing any comparison evaluation, or use algorithms that are capable of overlooking insignificant changes. This then leads us to the second problem, which is that different tools will invariably be needed for different document formats. I.e. we will need tools that are specifically tailored for common document formats in order to gain the best possible results. The following is a, very limited, overview of some of the existing cases and possible approaches to them.

3.8 Simple bitwise comparators (hashing)

As noted above, comparing strong hashes of two versions of a document can reveal if they are identical. The problem with this approach is twofold. As noted above it does not take into consideration that the document may contain a significant amount of irrelevant data. This could, however, be handled by passing the document through a supporting tool, prior to hashing, that removes the irrelevant data or extracts the actual content.

The other problem lies in the fact that this approach will always render a binary, changed or not changed, verdict. It does evaluate the degree of change! In many cases, that may be unimportant, but this may however also be of some interest, especially since we know that HTML pages (in addition to their layout) often contain dynamically generated sections that change frequently (such as ads and clocks) but do not contribute much to the actual content.

This hashing can be implemented (and already has been in Heritrix) using the SHA-1 hashing algorithm. There are also numerous other algorithms for this purpose. As time goes by, better algorithms, with a lower probability of collision, will undoubtedly be introduced.

The standard Unix-style "diff" tool provides a model for a slightly more informative comparison than the pure "yes/no" result of a hash-based comparison, one that identifies corresponding identical regions ("longest common substrings") in the two documents, and thus also can indicate where the regions of change are. This approach can ignore irrelevant details such as character case or whitespace, but remains unable to distinguish significant change (of intellectual content) from insignificant changes to formatting in more rich content types such as HTML.

3.9 "Roughly the same" comparators

Some, more advanced algorithms exist for calculating the relative amount of change. The approaches vary, but a common one is to divide the document up into 'shingles.' The number of identical versus different shingles between two documents then dictates the amount of change.

Many such algorithms presume the existence of a parser that can divide the document into a canonical sequence of tokens. I.e. ignoring layout, format and other information we choose to ignore, so that two documents formatted differently are reduced to the same sequence of tokens. A shingle is then a continuous subsequence of such tokens.

By evaluating these shingles the algorithm can evaluate the resemblance of the documents on the grade 0-1. Documents whose resemblance is very high are considered 'roughly the same' and assumed to be unchanged or identical.

It is worth noting that the efficiency and accuracy of these algorithms vary based on some parameters, such as shingle sizes.

The use of comparators based on these algorithms can significantly improve change detection, but it also requires some knowledge of the underlying document. Without a good parsing tool, the layout and formatting will continue to affect the results, much as they affect the hashing comparators.

3.10 Support tools

The most essential support tools for comparators will therefore be parsers, capable of extracting the content out of documents and ignoring the formatting. Clearly, different parsers will be needed for different document types.

Most notably, parsers will be needed for HTML documents, as they are the most common web resource and they also are known to change with high frequency. PDF parsers are also likely to be useful, as are parsers for any number of other document types including Microsoft Word and XML documents.

3.11 Summary of comparators

The following is a list of likely comparators:

A generic hashing comparator. Capable of handling all documents, if only in a primitive manner.

A byte-oriented text comparator. Analogous to the standard "diff" tool; correlates changed regions in textual documents without special handling of formatting markup (beyond perhaps ignoring whitespace or letter-case changes).

HTML formatting-insensitive basic comparator. Essentially the same as #1, but pre-processes the document with a tool that extracts only the content.

HTML formatting-insensitive advanced comparator. Uses a 'roughly the same' algorithm for evaluating HTML documents. A parser support tool is needed.

For most document types, variations on #3 or #4 could be created. For non-text based content, such as images, #1 will probably be sufficient as it is both (relatively) unlikely that they will change and it is very difficult to evaluate the amount of change in such documents in a meaningful manner.

For text-based formats, it will be necessary to evaluate how common the document type is and the frequency with which it generally changes. As creating more sophisticated comparators is somewhat expensive we will naturally want to emphasize those documents that are most common and change frequently.

3.12 Future directions

The following topics are enumerated to indicate possible future directions, without commenting on their value.

a) Comparisons of broader scope (page + embeddings, site, crawl).

b) Presentation approaches (how to display text deltas).

c) A generalized pure-text document format, allowing format-independent comparisons (e.g. HTML and PDF versions of the same doc).

d) Describing change over >2 items.

4 Prototype B: Show linking information

This prototype is motivated by use cases 4.3 and 5.1.2.

The provided prototype should have an API that given a reference to a specific URL object returns a representation of known links to and from the URL object.

4.1 Some questions to be addressed:

Should the reference to the specific URL object be to a specific version too (i.e. including date stamp)? Should the links returned be associated with specific dates? Should to-links and from-links have a common representation, or should they be represented differently? What is the relevant metadata of a to/from-link? Should the API allow requests for to/from-links within a given time period?

4.2 Item Link Information

This set of tools provides linking information about the items in an index. This task requires three tools:

- The link extractor, which identifies links in a single piece of content.
- The link analyzer, which pre-computes in- and out-linking for each item in a given segment of content.
- The reporting tool, which queries and presents the computed link information.

4.3 Link Extractor

A prerequisite for the link analyzer is a set of tools for link extraction. The Heritrix link extractors perform this role during the crawl process, and they can be used in post-crawl tools after some minor refactoring.

The link extractor tool, given a single archive object, extracts and categorizes the URLs found within it; if using Heritrix's model, the categories might be:

- HTML links
- CSS links
- JavaScript file links
- HTTP header links (redirects)
- META tag links
- BASE tag links
- HTML embeds

- Speculative embeds
- Prerequisites

The results should be generated in such a form as to be usable by future tools in the tool chain; integration at the code level would obviously be the most straightforward, but the link information is sufficiently simple in structure that we may easily define textual output formats.

4.4 Link Analyzer

Given the link extractor and an index, the link analyzer generates a complete set of in- and out-link information for the items in the index. It first visits every item in the index, runs the link extractor on the content, and saves the out-link information to the persistent store – for each item, what other items it references. It then inverts the resulting data to produce the in-links for each item – for each item, what items within the index that reference it – and saves the results to the persistent store.

The result is a persistent set of additional metadata for each item in an index, which can be queried as needed or post-processed, perhaps written to a file. Note that the out-linking information may include links to content not found in the index, and that the in-linking information will only contain links found within content in the index.

Unclear at the moment is how to address transitive linkages – relationships between more than two items, such as redirections (e.g. HTTP 30x or HTML META REFRESH) or nesting of content (such as HTML frames). Should these be addressed by the link analyzer, to make explicit that if A embeds B, which redirects to C, that we also explicitly record that A embeds C?

4.5 Query Tool

Having produced the link information, and categorized links to and from each object in our index, we are able to answer the following questions:

For a given source document and one or more types of linkage, what URLs are referenced (whether or not in the index)?

For a given destination document and one or more types of linkage, what items in the index reference it?

For reasons explained below, in response to Niels Christensen's questions, the results in both of these cases are pairs of a URL and the timestamp of the given document. This pair must be handed to the index to be dereferenced, to determine whether one or more actual items in the collection matches that identity, based on some measure of close matching, defined by proximity in time and/or having been collected in the same crawl. By doing this, we are able to additionally answer the following questions:

For a given source document and a link contained within it, what are the items in the index that best represent the target of the link?

Obviously, this dereferencing, to further focus on pairs of connected items actually held in the collection rather than just expressed linkages of URLs, is necessary for many reports of interest. Thus we define a secondary query tool, built atop the primary tool, that uses the output of the primary tool as input to a lookup that provides linkage information with actual collection items within a given time span of interest.

4.6 Some questions to be addressed

Should the reference to the specific URL object be to a specific version too (i.e. including date stamp)?

We can assume that in general linked (or embedded) items are not retrieved simultaneously with the source object. Indeed, linked items may not be retrieved until long after (or before) the source object, or they may be out of scope, or simply not exist. When querying for items in the collection known to be linked from a given source, then, we can only supply a confidence factor for whether a given item is truly the linked object, based on how close its timestamp is to the source item's timestamp.

However, we have already, in Task A, assumed there will be an interface to retrieve the closest/best match for a given timestamp, and so let us assume that this interface will provide the solution, and omit this logic from this tool. Therefore, the reference to a linked object should implicitly include the timestamp of the source object, rather than identifying the nearest true match.

The same ambiguity remains when querying for source items that link to a given destination item, for the same reasons. Again, the best answer we can give is to provide the destination item's timestamp and let retrieval indicate what is the closest match in time.

Should the links returned be associated with specific dates?

If the above discussion is assumed true, and an extraction tool is responsible for determining the best archived item for matching a given URL+timestamp, then a request for references to items linked from a given source object should return linked items associated with the precise timestamp of the source object. The lookup tool will identify the closest matching objects in the index, ordered by date proximity to the source.

Should to-links and from-links have a common representation, or should they be represented differently?

Is a link a first-class entity, or just a relation between documents? We have described the different categories of linkage, but it appears they all take the simple form of (source item, source timestamp, destination item, and category). The queries may be

different (give me links from this source or to this destination) but the representation is most likely the same.

What is the relevant metadata of a to/from-link?

The four elements named above probably comprise the minimal set. Depending on the category of the link reference, it would be reasonable to additionally include some or the entire context of the link within its containing document, for example:

- The offset in the file of the URL, and/or of its containing tag.
- Other attributes of the link tag (e.g. the stated dimensions of an embedded image, or the many attributes of a plug-in reference).

Extracting this metadata is potentially complex, and not of readily apparent value.

Should the API allow requests for to/from-links within a given time period?

Subject to the above discussion of the non-obvious aspects of link time stamping, it is a straightforward enhancement to find links of the appropriate type for each of the documents matching a time period and/or URL pattern, rather than just the links for a single document.

5 Prototype C: Define Segment

This prototype is motivated by use case 7.1.1 and can be considered as a prerequisite for 7.1.2, 7.1.3 and 7.2.1

The reason for this use case is that it seems inevitable that segments must be defined in order to better understand and make a little more sense of web archives that are compiled and of course the WWW, both for the institutions and for the users of the archives. Although the prototypes refer to the archive only the segment definition will certainly be used for harvesting as well.

A segment – as described in the use case – is an object representing a combination of positive and negative criteria that define whether a given URL object belongs to the segment. The segment API should include functionality for constructing a new segment, adding criteria to existing segments, inspecting what criteria make up a given segment and for telling whether a given URL object belongs to a given segment.

5.1 Some questions to be addressed:

What are the relevant types of criteria for defining a segment? URL restrictions? Date restrictions? Size restrictions? Mimetype restrictions? Content language restrictions? Linking patterns? Should segmentation only build on metadata or should it include criteria on the "actual bit stream"? Should segments with complex logical combinations be allowed, i.e. should any AND-OR-NOT combination be allowed? Are segments categorical ("yes, the URL object belongs to this segment" vs. "no, the URL object does not belong to this segment") or fuzzy (more possibilities than "yes" or "no", e.g. "don't know whether the URL object belongs to this segment").

5.2 Definitions

Defining a segment will depend on the collection policy of the web archive administrator and it should have a name and a description. Depending on that a web archive will be divided into segments that have certain common characteristics and features like: function (e.g. a library segment), topical, (political, scholarly, etc), even certain Mime types. In fact whatever serves a valid purpose. In this context it is important to note that segments can overlap and that certain URL's can belong to more than one segment.

A segment – as described in the use case – is an object representing a combination of positive and negative criteria that define whether a given URL object belongs to the segment. The segment API should include functionality for constructing a new segment, adding criteria to existing segments, inspecting what criteria make up a given segment and for telling whether a given URL object belongs to a given segment. The

administrator of the archive will define the relevant types of criteria for a segment, including restrictions (URL, date, size, mimetype, content language linking patterns), and how the criteria are applied. E.g. are they categorical (a URL either belongs or not) or fuzzy ("don't know whether the URL object belongs to this segment").

In practice those decisions will be difficult because they are subjective and could mean extensive manual intervention to evaluate and identify the Web sites that comprise the segment (i.e. if the initial domain is not too large).

In other words a segment is a subset of a web collection, [probably] represented by a simple list of the captures that fulfil the criteria that define whether a given URL object belongs to the segment, hereafter called a "Segment Definition List (SDL)". The initial compilation of a SDL list for a particular web domain is very difficult if it has not been generally harvested and stored in a web archive. The SDL might be built (or expanded) at harvest time, but for practical purposes the use case applies to an already-existing archive, and works through the repository/archive interfaces to storage.

In order to create a SDL for a segment, application code must be written which, when applied to each item in a collection in turn, would return whether or not that item is in the segment, and thus build a list of all captures making up the segment. This code would ideally be able to consider any aspects whatsoever of the relevant capture, including the content itself (if the segment-builder is willing to incur the space/time costs involved). It would also be natural to allow computing the intersection or union of segments, or create new segments by iterating over existing segments with new conditions.

Statistical information should be created for a segment with whatever collecting/calculating code is appropriate; in some cases this would happen simultaneous with the initial creation of the segment list (as full passes over collections are expensive). See prototype D, E and F.

In this prototype it is important to note that a segment will most likely not be static, but will be redefined as time passes. Therefore best practices must be developed for maintaining the SDL's. Although the archive is a closed environment it is still very complex and several methods are needed to compile the SDL for a specific segment definition. Use of Data Mining, Search tools etc is probably needed. Therefore a certain amount of iteration and manual intervention is needed for creating and maintaining the SDL's.

Using the real WWW is more complicated, but initially probably necessary, and through time it must be verified how well the archive reflects the real Web or whatever segment of the real Web is archived.

In a set of interface prototypes, there'd be interfaces for:

- A general web collection (whole thing or any segment).
- A segment (specializes collection to include membership criteria)

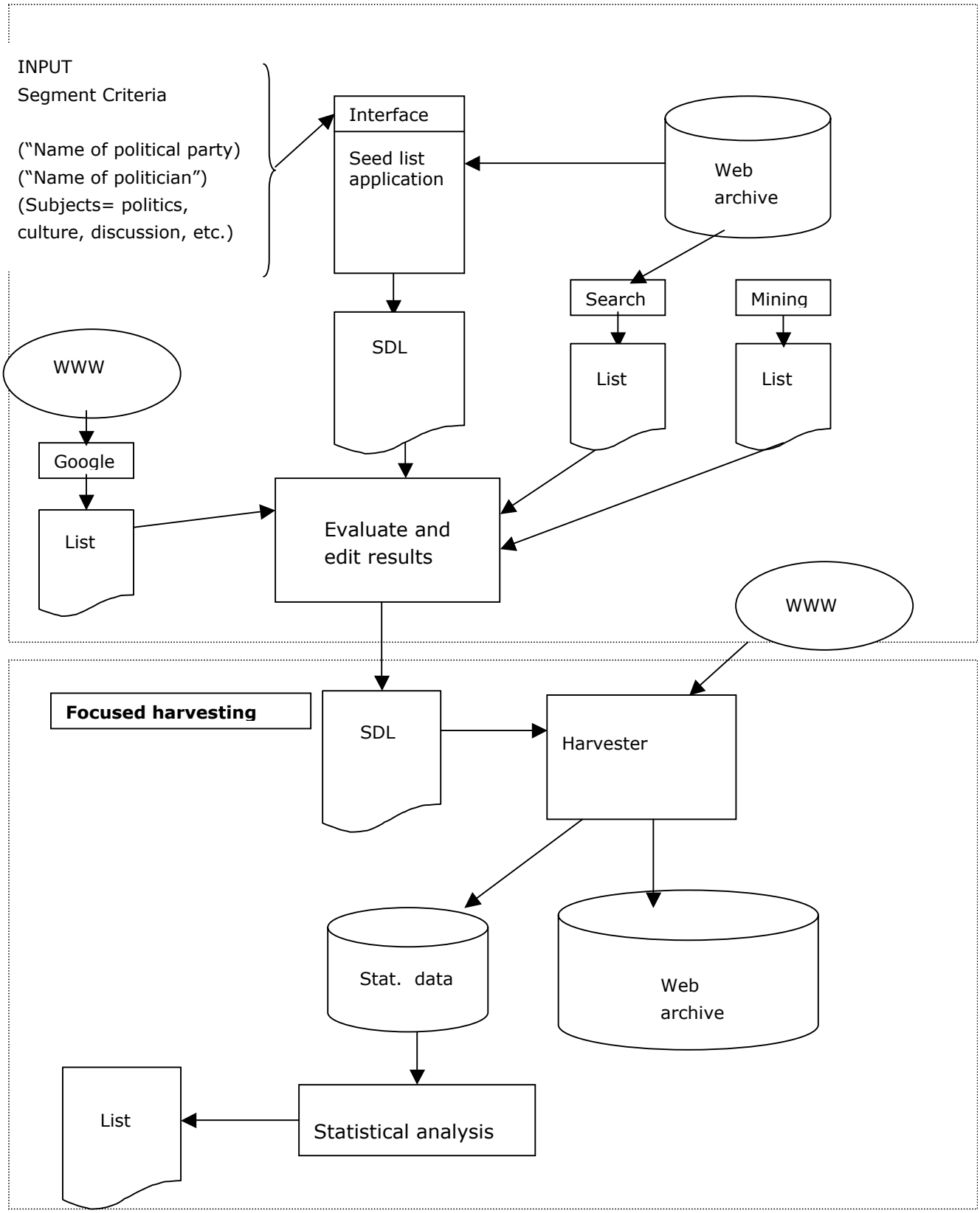
- List of including/excluding host URIs.
- List of including/excluding languages.
- List of including/excluding mime-types.
- Date ranges (including/excluding).
- File sizes (including/excluding).
- A web capture (representing one item in a collection/segment).
- A membership-test predicate (which rules a capture in or out of a segment).
- An analyzer (which applied to captures extracts/collects running statistics).

Segment data can be used:

- To manipulate statistical data in order to provide segment statistics.
- As a control mechanism (seed list) for segment harvesting.

The following diagram attempts to illustrate the scenario of an initial segment definition and the continued work that is involved when the results are used for focused harvesting using a segment seed list. The process defines a segment of political web-sites. This will be a continued activity

Initial segment definition



This prototype should provide functionality for inspecting the statistical properties of a given segment. Given a segment, the API should be able to return a representation of its statistical properties.

6 Prototype D: Statistical Information about a Segment

This prototype is motivated by use cases 7.1, 7.2, 7.3 and 7.4. This task is implemented with the combination of a flexible query tool which generates the statistics of interest from an index (as defined above), and a precomputed database of relevant metadata for the items in the collection.

Let us make the assumption that task C, responsible for defining segments, includes a tool to generate, from the definition of a given segment, a pre-computed index. Then the extraction of statistics for a segment becomes the two-step process of generating the index for that segment's definition, and then invoking the query tool against the index.

6.1 Some questions to be answered

What are the relevant properties? Number of domains? Size distribution of domains? Number of web pages/documents? Number of files? Total size? Other significant properties? How should this data be represented?

How should this data be represented? Should the API allow requests for one or more specific properties of a segment rather than the whole segment? Should the representation be serializable (i.e. suitable for storing on disk for later inspection)? Should the API include functionality for comparing the statistical properties of two or more different segments? Should the API include functionality for comparing the current statistical properties of the given segment with earlier summaries (saved on disc) of the same properties for the same segment (allowing the user to learn about the development of segment)?

What's certain is that there is no good answer to those questions yet and the answers to those questions will change. Thus, the primary requirement is to be extensible, and to allow ad-hoc statistical extraction by ensuring that as much raw metadata from the content as possible is extracted. The use of a database for metadata should provide this flexibility.

Should the API allow requests for one or more specific properties of a segment rather than the whole segment?

Again, the capability for ad-hoc queries appears more and more to be a requirement.

Should the representation be serializable (i.e. suitable for storing on disk for later inspection)?

At least for the specific case of common, well-understood reports, it makes sense to define simple textual output formats that can be post-processed as needed. The

database might be used for storing the results of a report run, in order to achieve this sort of persistence.

Should the API include functionality for comparing the statistical properties of two or more different segments?

Perhaps more generally the API should provide features for examining the change over time of a value or statistic.

Should the API include functionality for comparing the current statistical properties of the given segment with earlier summaries (saved on disc) of the same properties for the same segment (allowing the user to learn about the development of segment)?

Based on the above commentary, the change of these statistics over time will be of interest, and thus the database and API should provide for this sort of analysis.

7 Prototype E: Process Objects

This prototype is motivated by several use cases including 7.1.2, 7.1.3, 7.2.1, 7.2.2 and 7.4.1.

The prototype should allow batch jobs to run across all URL objects in the archive. The API should define how to specify the functionality of a batch job as well as how to run the job, returning its results.

7.1 Some questions to be addressed:

Should there be restrictions on batch job functionality? Should batch jobs be read-only? Should batch jobs be distributable, allowing the system to ship them to the machine where data is stored? How should a batch job return data? How does the batch handle broken files or records (does the processing stop)?

7.2 Implementation

The code consists of three files (two classes and one interface). Note that there is no actual functionality in the code - just the relevant APIs.

Prototype_E/FileBatchJob.java

```
import java.io.File;
import java.io.IOException;
import java.io.Serializable;
/**
 * Interface defining a batch job to run on a set of files.
 */
public interface FileBatchJob extends Serializable {
    /**
     * Initialize the job before runnning.
     * This is called before the processFile() calls
     */
    public void initialize();
    /**
     * Process one file.
     *
     * @param file to be processed.
     */
    public void processFile(File file);
    /**
     * Finish up the job.
     * This is called after the last process() call.
     */
```

```
    public void finish();
}
```

Prototype_E/ARCBatchJob.java

```
import org.archive.io.arc.ARCRecord;
import java.io.File;
import java.io.IOException;
/**
 * Abstract class defining a batch job to run on an ARC archive.
 * Concrete jobs inherit from this class.
 */
public abstract class ARCBatchJob implements FileBatchJob {
    /** Initialize the job before runnning.
     *  This is called before the processRecord() calls start coming.
     */
    public abstract void initialize();
    /**
     * Process one object stored in the ARC archive.
     * @param rec the object to be processed.
     */
    public abstract void processRecord(ARCRecord rec);
    /**
     * Finish up the job.
     * This is called after the last processRecord() call.
     */
    public abstract void finish();
    /**
       * Accepts only ARC and ARCGZ files. Runs through all records and
calls
       * processRecord() on every record. Any problems with reading the
file
       * or its records are reported by calling errorInRecord() and
errorInFile().
       * @param arcFile The ARC or ARCGZ file to be processed.
       */
    public void processFile(File arcFile) {
    }
    /**
    * When the org.archive.io.arc classes throw IOExceptions while reading
a specific record,
    * this is where they go. Subclasses are welcome to override the
default
    * functionality which does nothing.
    * @param e An Exception thrown by the org.archive.io.arc classes.
    */
    public void errorInRecord(IOException e) {
    }
    /**
    * When the org.archive.io.arc classes throw IOExceptions while opening
or closing a
    * file, or while moving between records, this is where they go.
    * Subclasses are welcome to override the default functionality which
does nothing.
```

```
    * @param e An Exception thrown by the org.archive.io.arc classes.
    */
    public void errorInFile(IOException e) {
    }
}
```

Prototype_E/BatchLocalFiles.java

```
import java.io.File;
/**
 * Class for running FileBatchJobs on a set of local files.
 */
public class BatchLocalFiles {
    /**
     * Constructs a new object from a list of files.
     *
     * @param files - the files that should be used for batching.
     */
    public BatchLocalFiles(File[] files) {
    }

    /**
     * Run the given job on the files associated with this object.
     *
     * @param job - the job to be executed
     */
    public void run(FileBatchJob job) {
    }
}
```

8 Prototype F: Format Distribution in a Segment

This prototype is motivated by use cases 7.2.1, 7.2.2 and 7.3.1. It may overlap with prototype C (defining a segment).

This prototype should provide functionality for inspecting the distribution of content formats in a given segment. Given a segment, the API should return a representation of its format distribution.

8.1 Some questions to be addressed

What information should be associated with each format? Total number of URL objects? Total size of URL objects? Average size? Average URL object age? From which classification is the format taken? Mimetypes? Should it be possible to choose between different format classifications? Should the API include functionality for comparing the current format distribution with earlier format distributions (saved on disc) in order to inspect the development? Should the API include functionality for looking up a given format? If so, how is a format referenced (by its name as a string or perhaps by an index number)?

There is more than one way to approach this problem. Here speed may be an issue, since, if the archive is big, choosing the wrong way of doing may give something that is unusable in practice. An important factor here is whether there is a separate metadata database or if the whole archive has to be searched. Two possible solutions are sketched. The goal is to start some discussions.

8.2 Solution 1

An object is define that is acted upon by a couple of methods. First there are three methods that define the selection criteria. Then there are three methods to extract the wanted information from the data set defined by the previous methods.

The Methods

Obviously there must be a constructor. In perl it may be named ”new”.

Set_Filter_url:
Set a filter against which the urls are matched. It takes as parameter a regular expression.
Q: do we need separate methods for protocol, host and path?

Set_Filter_Type
Sets a filter on MIME type. Parameter is a list of MIME types to accept. It should accept wildcards. I.e. it must be possible to demand ”text/*.

Set_filter_Date
Sets a filter on date-time to include. Parameters are two dates. Only setting the starting date means: everything from this day until today. Only end date means: everything up to ad including end date.

The date should be given according to iso-standard. I.e. 1998-12-01 05:50:10. The time part is optional.

All filters accept everything by default. I.e. if you don't call any of the methods you get the whole archive.

Then there are three methods to extract information

Get_Summary
Returns a summary of the data. Returns Number of, files, bytes, hosts and formats. NB, the last item is the number of formats.

Get_Format_List
Returns a list which for each host contains: the host, nb of files and nb of bytes.

I haven't specified how this list is to be given since there are several possibilities and the best way may be different in different languages. To be discussed further.

Get_Host_List
Same as above but the list contains: format, nb of files and nb of bytes.

As you can see I have no explicit ”execute” of the select, that is hidden inside the ”Get”-methods. The reason is that after you have defined your selection you don't know which information you are going to extract. If you execute the select at this point the object than has to store all information necessary to satisfy any possible demand. A disadvantage of the chosen method is that if you are going to use several ”Get”-methods for a given selection you will start a, possible time consuming, processing of the data in you archive. This should be discussed further.

Also, I have not defined how to return the data. The best way may be different for different implementation languages and is also a matter of taste. Discussions needed.

A sample program in perl

```
#!perl
# create statistics object
my $stat_obj = new IIPC::statis();
#select all kb.se hosts
$stat_obj -> Set_Filter_url('*.kb.se');
```

```
# select html documents
$stat_obj -> Set_Filter_Format('text/html');
#select 2000 onwards
$stat_obj -> Set_Filter_Date('2000-01-01'); # no second argument means no
end date. I.e. until the present.
$summary = $stat_obj -> Get_Summary();
```

The $summary entity now contains the necessary information one way or another and can be e.g. printed out.

8.3 Solution 2

Here I define a generic ”get_stat”-method. Which for a certain selection, controlled by parameters, returns the summed statistics.

Something like:

```
my ($sum_bytes,$sum_files) =
IIPC::get_stat($archive,$mime,$time_start,$time_end,\@hosts)
```

where the parameters are:
$archive: the archive in question
$mime: MIME-type to select
$time_start, $time_end: start and end date-times
@hosts: list of hosts to include

Wild cards etc works as in the first approach.

This looks very simple. However, there is a lot of logic hidden from view here.

9 Prototype G

This prototype is motivated by use case 4.5.

It considers interfaces for saving, restoring and manipulating search criteria (these may be e.g. text-based or URL based like in the Wayback Machine).The provided prototype should have an API that represents a list of search results and it should provide functions for saving (serializing) and restoring (deserializing) such a list. Each result should be associated with some kind of persistent reference to the result object along with relevant metadata.

9.1 Some questions to be addressed

What metadata is relevant for each result? URL? Title? Size? Format? Ingest date? Details on how it matched the search criteria? Should the search criteria be saved (to allow redoing that search when more objects have been ingested)? What is the best representation of criteria? String-based or more structured? What are the relevant logical operators and what are the relevant combinations? Should the system allow any combination of searches (e.g. combining metadata search with text-based) or not? Are there any relevant personalization issues?

9.2 Method

The IIPC has been involved in two access products, i.e. the WERA and the NutchVAX. Some of the issues raised for this prototype are addressed in those applications and the others will be given due consideration.

10 Prototype H: On demand harvesting

This prototype is motivated by use case 4.11.

The prototype should provide an API for requesting harvest of one or few URLs. The request should be accompanied by e.g. an email address for response. The system may respond immediately (e.g. denying the request) or later (e.g. informing that the harvest has been performed).

10.1 Some questions to be addressed

What parameters are relevant for harvest requests of the sort described in case 4.11? Should the API e.g. allow requests for repeated harvestings? For harvesting of entire sub domains? Should some sort of correspondence history be kept for each email address? What are the likely workflows around harvest requests?

10.2 Definitions

In the Heritrix harvester a *crawlRequest* is a combination of a globally unique *crawlRequestID*, a starting point, a scope specification (including a set of "sort friendly URI'S called SURTs" for *SURTPrefixScope*?), and standard harvesting/crawl parameters.

1) A *crawlRequest* can be remotely submitted at any time to any running Heritrix instance.

2) The harvester/crawler instance that receives the *crawlRequest* begins the requested harvesting immediately, independently of any other currently running harvestings/crawls in that instance.

3) The harvester/crawler instance generates ARC records tagged with the *crawlRequestI*D that is responsible for the harvesting/crawling that generates the ARC record.

4) Heritrix instances can be remotely polled to get a list of currently running harvests/crawls *crawlRequestIDs*.

5) Status reports (TBD) are remotely available for each running harvest/crawl, referred to by *crawlRequestID*.

6) Individual harvests/crawls can be paused or terminated at any time (by *crawlRequestID*) without impacting other running harvests/crawls in a given instance.