Slug: A Semantic Web Crawler

Leigh Dodds, leigh@ldodds.com, February 2006

Abstract

This paper introduces "Slug" a web crawler (or "Scutter") designed for harvesting semantic web content. Implemented in Java using the Jena API, Slug provides a configurable, modular framework that allows a great degree of flexibility in configuring the retrieval, processing and storage of harvested content. The framework provides an RDF vocabulary for describing crawler configurations and collects metadata concerning crawling activity. Crawler metadata allows for reporting and analysis of crawling progress, as well as more efficient retrieval through the storage of HTTP caching data.

Introduction

In concept a semantic web crawler differs from a traditional web crawler in only two regards: the format of the source material it is traversing, and the means of specifying links between information resources. Whereas a traditional crawler operates on HTML documents, linked using HTML anchors, a Semantic Web crawler (or "scutter" [Scutter]as they're known within the community) operates on RDF metadata with linking implemented using the `rdfs:seeAlso` relationship [Brickley2003].

However, in practice, the aggregation and processing of semantic web content by a scutter differs significantly from that of a normal web crawler. A normal crawler must only contend with extracting text from, possibly invalid, content and subsequent link extraction, whereas a semantic web crawler must carry out additional processing tasks: merging of information resources via Inverse-Functional-Properties; tracking provenance of data; harvesting schemas and ontologies in addition to source data; extraction of embedded metadata (e.g. EXIF, XMP), etc. Matt Biddulph's 2004 paper "Crawling the Semantic Web" [Biddulph04] provides an excellent introduction to these issues and summarizes implementation experience gained whilst constructing a scutter.

Despite this promising initial research, and the growing availability of rich sources of freely available metadata, there are still very few open source semantic web crawling frameworks [RDFCrawl][Biddulph03][Elmo].

Although large RDF datasets are readily available in a number of specific domains, e.g. bioinformatics, there are as yet no scutters actively aggregating semantic web data for the purposes of enabling easier application development and further research. The creativity surrounding "mashups" [Mashup], which mix data and functionality across domains, highlights the need for easily accessible cross-domain RDF data sets. Semantic web research into areas such as storage and query performance, federated queries, etc. will be enabled by the provision of a wider variety of data sources. For example recent experiences have shown that the structure of an RDF data set can greatly effect triple store performance [Broekstra].

To address these concerns and encourage additional research, the author has written Slug [Slug], an open source, configurable and modular scutter. The emphasis of the framework is on modularity and ease of use rather than speed (hence the name!), although performance is currently acceptable.

The rest of this paper describes the design, and features of the framework.

Summary of Features

The current release ("alpha-2") of the Slug crawling framework includes the following functionality:

- Multi-threaded retrieval of data via HTTP
- Support for HTTP Conditional GETs [Biddulph03b], enabling optimised retrieval of previously fetched content
- Internal collection of statistics on crawler state, e.g. number of active workers, to facilitate monitoring of the crawler
- Crawling from a fixed list of starting points (a "Scutter Plan" [ScutterPlan]) and/or refreshing previously crawled data sources
- A persistent "memory" for capturing and persisting crawl related metadata. E.g. HTTP status codes from responses, location of local copies of data, location from where a source was found, etc.
- Configurable crawler behaviour including crawling "depth", blacklisting of URLs based on regular expressions, and avoidance of crawling loops
- Automatic processing and traversal of `rdfs:seeAlso` links between RDF documents
- Creation of a local cache of retrieved data
- Storage of retrieved RDF data using Jena persistent models
- Configurable processing and filtering pipelines enabling the scutter to be customised for specific tasks
- Creation of scutter "profiles" using a custom RDF vocabulary for describing the scutter configuration

Crawler Architecture

The UML diagram provided in Figure 1 summarizes the key components and relationships present in the Slug framework. Central to the design of the framework are variations of the Master-Slave and Producer-Consumer design patterns.

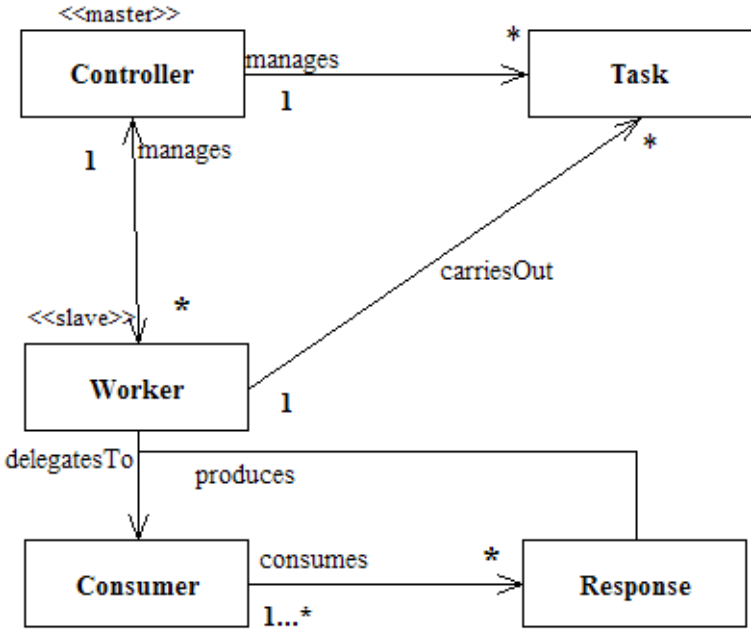

Figure 1: Summary of Slug Architecture

The Controller is responsible for managing a list of Tasks which are carried out by a number of Worker instances. The Controller implementation is agnostic to the exact nature of each Task and

the implementation of the Workers. A factory class is used by the Controller to create worker instances at application startup and on demand during application processing; each Worker runs as a separate thread within the application, co-ordinated by the single Controller instance.

The core Controller-Worker framework is therefore similar to the Master-Slave design pattern in which a central component delegates tasks to a number of slave instances. The variation imposed here is that the Controller itself does not aggregate the results of completing each Task; neither do the workers.

The processing of task results, which may involve multiple processing stages, is abstracted from the task itself by applying the Producer-Consumer pattern. Each Worker *produces* results which are handled by a Consumer instance. This allows alternate processing behaviour to be substituted into the framework without changing the core functionality that deals solely with multi-threaded retrieval of content via HTTP.

Finally, to allow for modular extension of the consuming of task results, the framework provides an implementation of the Consumer interface that co-ordinates several other Consumer instances. This delegation model allows multiple Consumer instances to be involved in processing a single Response generated by each Worker.

A typical Consumer configuration might consist of:

- A ResponseStorer that stores retrieved RDF data in a local file system cache, treating the response as a series of bytes
- An RDFConsumer that parses the retrieved data, extracting `rdfs:seeAlso` links in order to generate new Tasks, adding them to the Controllers job queue.
- A PersistentResponseStorer that parses the retrieved data, storing the resulting triples in a persistent Jena model

Custom Consumer implementations can be implemented and used within the framework by declaring them within the crawler configuration (see below). E.g. A custom consumer might discover additional RDF data to be crawled by inspecting more properties than just `rdfs:seeAlso` links.

Another key component in the architecture are “task filters”. These components are used by the Controller to filter any new tasks encountered, in order to ensure they meet arbitrary criteria. For example: that the URL associated with the new task does not match a given regular expression, allowing blacklisting of specific URLs. As with the Consumer interface, the framework allows for multiple task filters to co-operate in the decision to admit a new task to the queue. Complex filtering criteria can therefore be assembled from fine-grained filter implementations.

The final aspect to the Slug framework is the concept of a “memory”. A Memory instance, maintained by the Controller, but accessible by all framework components, provides a persistent RDF model for storing crawl related metadata. E.g. HTTP response codes and headers. See “The Scutter Vocabulary” for more details. All components have access to the working Memory so can inspect it to query recent activity or to store custom metadata.

All of the components in the Slug framework can be configured using a custom RDF vocabulary that allows extension of its capabilities by integrating custom components, e.g. new TaskFilters or Consumers.

Crawler Configuration

A Slug scutter instance is configured via a proprietary RDF vocabulary, in the namespace “`http://purl.org/NET/schemas/slug/config/`”. A sample crawler configuration file is provided in Appendix 1.

The classes and properties in the configuration vocabulary echo the main application architecture:

- A Scutter resource has properties configuring its initial number of workers, its memory, and the required consumer and task filter components. The latter are both modelled as RDF sequences.
- A Memory resource is configured with properties indicating a file name (in the case of a simple file system based memory) or database connection parameters.
- Consumer and Filter components are configured using an "impl" property that indicates the fully-qualified Java class name of the implementation. Java reflection is used to automatically instantiate these components at runtime. Component instances are responsible for configuring themselves further from the available metadata: each instance is passed a reference to its "peer" RDF resource in the configuration file immediately after instantiation.

An RDF vocabulary was chosen as the means for configuring the framework rather than a custom XML vocabulary or Java properties file for several reasons.

Firstly the ability to add arbitrary additional properties makes it easy to extend the configuration for specific component instances. E.g. a custom Consumer instance may need to be configured with the location and authentication credentials for a remote HTTP service to which it passes newly fetched data. This same flexibility can also be exploited to improve readability of the configuration by using Dublin Core properties to associate a title and description with each component.

Secondly, it was desirable to allow for the specification of different crawling "profiles": alternate configurations of the framework specialised for different crawling strategies. Using an RDF vocabulary it was easier to relate together individual components and the Scutter instance that uses them.

For example a configuration file may contain the specification of a "fast, shallow" scutter that defines a Scutter instance with a large number of worker threads, but a task filter that ensures that the depth of crawling is quite shallow. The same configuration might define a "slow, deep" scutter than uses a smaller number of workers, allows for a large depth when following links, but additionally blacklisting any LiveJournal URLs encountered during the crawl. These two scutter profiles may share the same configuration relating to memory storage, and the consumer pipeline.

These configuration options allow the basic framework to be used to specify a number of different crawling profiles by reconfiguring the components for specialised purposes.

The Scutter Vocabulary

The persistent memory created by the Slug framework is an initial implementation of the ScutterVocab specification [ScutterVocab]. This specification is still under development so details may vary; the current implementation is summarised below. Refer to Appendix 2 for sample data.

A Representation is an anonymous resource that describes a source URL encountered by the crawler. A Representation should have a `source` property that indicates the original web resource it describes. Additionally a Representation may have any number of `origin` properties. These properties identify documents that refer to the specific Representation, i.e. the origin of the `rdfs:seeAlso` link(s) that lead the scutter to discover and then retrieve this URL.

Therefore over subsequent runs of the scutter the `origin` properties associated with each Representation will end up producing a "map" of the semantic web network.

A given Representation may be retrieved any number of times, resulting in multiple `fetch` properties. Each Fetch is annotated with properties describing the results of the fetch, including the date of retrieval, number of triples it contains (if parsed), and additional HTTP metadata such as the status code and Last-Modified and Etag headers. These headers are used by the workers to carry out

Conditional GETs avoiding repeated downloads of the same content.

If a given Fetch results in an error then this too is recorded, along with a suitable error message. The ScutterVocab allows Representations to be annotated with a skip property that indicates that it should be ignored in subsequent crawls. The Slug framework automatically generates these properties whenever it encounters specific errors, e.g. “Unknown Host”, or the HTTP status codes “404 Not Found”, “410 Gone”, or “403 Not Authorised”.

Over time, the Fetch metadata produces a complete history of crawler activity, including error reporting and reasons for blacklisting.

See the later section “Analying Crawler Activity with SPARQL” for examples of manipulating the crawler memory.

Working With Slug

The following sections outline how to work with the Slug framework for particular purposes. The first example discusses reporting on scutter activity, e.g. for the purposes of mapping or disagnose crawling problems. The second example illustrates how to use Slug to create a local cache of resources and then make them available via the Jena FileManager.

Analysing Crawler Activity With SPARQL

The Jena framework bundled with Slug includes the ARQ SPARQL library. This means that SPARQL queries can be used to generate reports from the scutter's memory. The Slug distribution includes the following useful queries:

Query Name	Description
list-errors.rq	Lists the errors associated with the most recent Fetch of each Representation. I.e. The most recently encountered errors
list-fetches.rq	List the crawl history for a given resource
list-largest.rq	List the 10 resources that generated the most triples
list-mimetypes.rq	List all mimetypes encountered by the scutter from successful fetches
list-origins-of-resource.rq	List which resources point to a given resource. E.g. which documents include rdfs:seeAlso links to this document?

The following SPARQL query shows the contents of the file list-fetches.rq:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX scutter: <http://purl.org/net/scutter/>
SELECT ?date ?status ?contentType ?rawTripleCount
WHERE
{
  ?representation scutter:fetch ?fetch.
  ?fetch dc:date ?date.
  OPTIONAL { ?fetch  scutter:status ?status. }
  OPTIONAL { ?fetch scutter:contentType ?contentType. }
  OPTIONAL { ?fetch scutter:rawTripleCount ?rawTripleCount. }
  ?representation scutter:source <http://www.ldodds.com/ldodds.rdf>.
```

```
}
ORDER BY DESC(?date)
```

Queries such as those listed above are important for administering a scutter instance. It should be possible to discover errors, trace the history of interactions with a given resource, etc. The crawl metadata may also be useful for identifying problems with the publishing of semantic web data, e.g. broken links or incorrect mime types in responses.

Cross-referencing crawl metadata with that aggregated by the scutter (e.g. into a persistent store) provides other possibilities, e.g. tracing usage of particular RDF schemas and/or properties.

Creating a Local Data Cache

The Jena FileManager [FileManager] provides a simple factory class for loading RDF data from the local disk or network. The FileManager delegates locating of individual resources to a LocationMapper instance. A LocationMapper may read a local configuration file for mapping of remote resources to local cached instances. E.g. to avoid network fetches when the desired data is available locally. The Jena documentation contains further details on the operation of the FileManager and LocationMapper components.

This basic functionality is very similar in nature to the XML Catalogs [XMLCatalogs]used by many XML application frameworks. Whereas the XML Catalogs use a standard XML vocabulary for declaring mapping rules, the Jena LocationMapper uses an RDF vocabulary.

The following example illustrates a sample mapping entry from a LocationMapper configuration file. In this case a remote RDF document is mapped to a locally cached copy:

```
[] lm:mapping
   [ lm:name "http://www.ldodds.com/ldodds.rdf" ; lm:altName
"file:etc/ldodds.rdf" ]
```

A basic feature of the Slug framework is the capability of creating a local cache of remote RDF data. The crawler memory (see above) includes properties that map source HTTP URLs to local file system URIs. It is therefore possible to apply some simple inferencing rules to convert from one vocabulary to another. The following example uses the Jena rules syntax [Inference].

```
@prefix scutter: <http://purl.org/net/scutter/>.
@prefix lm: <http://jena.hpl.hp.com/2004/08/location-mapping#> .

[inferMappingFromLocalCopy:

   (?R rdf:type scutter:Representation),
     (?R scutter:source ?source),
      (?R scutter:localCopy ?local),
         makeTemp(?mapping),
            makeLiteral(?source, ?literalSource)

     -> (?mapping lm:name ?literalSource),
        (?mapping lm:altName ?local)]

[createMapping:
   (?mapping lm:name ?source), makeTemp(?x)
      -> (?x lm:mapping ?mapping)]
```

The rules assume the availability of a bespoke inferencing primitive that binds the String value of a variable to a newly created literal. This is required to map the `source` properties from the ScutterVocab (which are RDF Resources) to the literal values used in the location mapping vocabulary. An implementation of this primitive is included in the Slug distribution.

Jena based applications may apply these rules to generate a location mapping configuration which

can then be used to optimise how the application fetches remote resources. One specific use case for this feature is in the implementation of a SPARQL Protocol endpoint which may deal with arbitrary specified SPARQL Datasets.

Planned Future Work

Additional future work is planned on the Slug framework, including:

- Additional Consumer implementations, e.g. for publishing retrieved data to remote data sources
- Additional TaskFilter implementations, to allow white-listing of URLs
- Improvements to the Worker implementations to support additional HTTP operations, e.g. content negotiation, the Robot Exclusion Protocol, HTTP Authentication, etc.
- Refactoring of the Consumer components to implement a complete Pipe-And-Filters architecture. This will allow handling and extraction of alternative content formats, e.g. extracting metadata from HTML documents using GRDDL [GRDDL], microformats, etc.

Appendix 1: Sample Scutter Configuration

```
<slug:Scutter rdf:about="slug">
  <slug:hasMemory rdf:resource="memory"/>

  <slug:workers>10</slug:workers>

    <slug:consumers>
      <rdf:Seq>
       <rdf:li rdf:resource="storer"/>
       <rdf:li rdf:resource="rdf-consumer"/>
      </rdf:Seq>
    </slug:consumers>

    <slug:filters>
      <rdf:Seq>
        <rdf:li rdf:resource="single-fetch-filter"/>
        <rdf:li rdf:resource="depth-filter"/>
        <rdf:li rdf:resource="regex-filter"/>
      </rdf:Seq>
    </slug:filters>
</slug:Scutter>

<slug:Memory rdf:about="memory">
  <slug:file>c:\eclipse\workspace\slug\memory.rdf</slug:file>
</slug:Memory>

<slug:Consumer rdf:about="storer">
  <dc:title>ResponseStorer</dc:title>
  <slug:impl>com.ldodds.slug.http.ResponseStorer</slug:impl>
  <slug:cache>c:\temp\slug-cache</slug:cache>
</slug:Consumer>

<slug:Consumer rdf:about="rdf-consumer">
  <dc:title>RDFConsumer</dc:title>
  <slug:impl>com.ldodds.slug.http.RDFConsumer</slug:impl>
</slug:Consumer>

<slug:Filter rdf:about="regex-filter">
  <dc:title>Block URLs based on Regex</dc:title>
  <slug:impl>com.ldodds.slug.http.RegexFilter</slug:impl>
```

```
    <!-- regular expression, if matches, then url not included -->
    <slug:filter>livejournal</slug:filter>
   </slug:Filter>
```

Appendix 2: Scutter Memory Fragment

```
<scutter:Representation>
  <scutter:source rdf:resource="http://www.ldodds.com/ldodds.rdf"/>
  <scutter:origin rdf:resource="http://ldodds.com/ldodds-documents.rdf"/>
  <scutter:origin rdf:resource="http://ldodds.com/ldodds-favbooks.rdf"/>
  <scutter:origin rdf:resource="http://ldodds.com/ldodds-events.rdf"/>
  <scutter:origin rdf:resource="http://ldodds.com/ldodds-projects.rdf"/>
  <scutter:origin rdf:resource="http://ldodds.com/ldodds-toread.rdf"/>
  <scutter:origin rdf:resource="http://ldodds.com/ldodds-knows.rdf"/>
  <scutter:origin rdf:resource="http://ldodds.com/ldodds-reading.rdf"/>
  <scutter:origin rdf:resource="http://ldodds.com/ldodds-read.rdf"/>

  <scutter:latestFetch>
     <scutter:Fetch>
       <dc:date>2006-02-27T14:46:29+0000</dc:date>
       <scutter:status>304</scutter:status>
       <scutter:etag>"192033-1e38-44217fc0"</scutter:etag>
     </scutter:Fetch>
  </scutter:latestFetch>

  <scutter:fetch>
     <scutter:Fetch>
       <dc:date>2006-02-27T14:14:49+0000</dc:date>
       <scutter:status>304</scutter:status>
       <scutter:etag>"192033-1e38-44217fc0"</scutter:etag>
     </scutter:Fetch>
  </scutter:fetch>

  <scutter:fetch>
     <scutter:Fetch>
       <dc:date>2006-02-27T14:05:12+0000</dc:date>
       <scutter:status>200</scutter:status>
       <scutter:contentType>text/xml</scutter:contentType>
       <scutter:lastModified>Thu, 16 Jun 2005 12:47:03
GMT</scutter:lastModified>
       <scutter:etag>"192033-1e38-44217fc0"</scutter:etag>
       <scutter:rawTripleCount>107</scutter:rawTripleCount>
     </scutter:Fetch>
  </scutter:fetch>
```

```
  <scutter:localCopy>c:\temp\slug-
cache\www.ldodds.com\ldodds.rdf</scutter:localCopy>

</scutter:Representation>
```

References

[Biddulph03] Matt Biddulph (2003) An RDF Crawler
http://www.hackdiary.com/archives/000030.html
[Biddulph03b] Matt Biddulph (2003) Using HTTP conditional GET in java for efficient polling
http://www.hackdiary.com/archives/000028.html
[Biddulph04] Matt Biddulph (2004) Crawling the Semantic Web
http://www.idealliance.org/papers/dx_xmle04/papers/03-06-03/03-06-03.html
[Brickley2003] Dan Brickley (2003) RDF Hyper-linking
http://www.w3.org/2001/sw/Europe/talks/xml2003/Overview-6.html
[Broekstra] Jeen Broekstra (2006) Pitfalls in Benchmarking Triple Stores
http://jeenbroekstra.blogspot.com/2006/02/pitfalls-in-benchmarking-triple-stores.html
[Elmo] Peter Mika () The Elmo Scutter
http://www.openrdf.org/doc/elmo/users/userguide.html#d0e229
[FileManager] () The Jena FileManager and LocationMapper http://jena.sourceforge.net/how-to/filemanager.html
[GRDDL] Dominique Hazaël-Massieux, Dan Connolly (2005) Gleaning Resource Descriptions from Dialects of Languages (GRDDL) http://www.w3.org/TeamSubmission/grddl/
[Inference] Dave Reynolds (2005) Jena 2 Inference Support
http://jena.sourceforge.net/inference/index.html#rules
[Mashup] () Mashups http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)
[RDFCrawl] Siegfried Handschuh (2000) RDF Crawl http://ontobroker.semanticweb.org/rdfcrawl/
[Scutter] () Scutter http://rdfweb.org/topic/Scutter
[ScutterPlan] () Scutter Plan http://rdfweb.org/topic/ScutterPlan
[ScutterVocab] Morten Frederiksen (2003) Scutter Vocab http://rdfweb.org/topic/ScutterVocab
[Slug] Leigh Dodds (2006) Slug: A Semantic Web Crawler http://www.ldodds.com/projects/slug
[XMLCatalogs] (2005) XML Catalogs http://www.oasis-open.org/committees/download.php/14041/xml-catalogs.html