

Data Gathering system

Stephen Wattam

January 13, 2011

1 Requirements

The system to gather data for the document attrition data set must ensure a few simple properties are adhered to.

Firstly, the integrity of the network is likely to affect data gathering in a number of ways. Though measuring network performance statistics for websites might be useful, any variation therein is likely to be eclipsed (or at least inseparable from) local variations caused by the process of downloading data itself. The inherent instability of routing online makes reaching a fully working site a matter of best-effort, and the whole study is working under the assumption that the one connection at Lancaster University is representative of most clients' experiences online.

Client-profiling and restricting measures by end servers are also likely to make a big difference to the result. For this reason, the client doing the downloading will have to impersonate a commonly used internet browser. As of the time of writing (30th November 2010) this means Internet Explorer on Windows, Firefox on Windows, or Chrome on Windows. Switching content on user-agent is not a common technique (except when detecting web crawlers) and this is not expected to affect the contents of the corpus significantly.

Variation in character set will require modification either to the storage format (storing in the original format) or to the data itself (blanket conversion to utf-8). The latter of these is simpler to process during later stages, and is unlikely to introduce error on the assumption that a page's content-encoding header is set correctly when downloaded.

Storage of non-text page resources (and supplementary resources such as CSS and JavaScript) would result in very large storage requirements, and arguably add little to any resulting analyses. As such they shall be omitted, with only the requested HTML being stored. The HTTP request in full shall be stored, including all header fields, as these may be extracted at a later date and used as appropriate.

1.1 Process

The system must:

1. Download a large list of URLs in as short a time as possible (will require parallel requests)
2. Store these URLs in a hierarchical folder structure for easy retrieval (/yyyy/mm/dd/page or /week/page)
3. Retry and queue requests as a client's browser would
4. Appear to the server as a web browser on a "new visit" to the page (without cookies or sessions)
5. Store HTTP headers and request data for later processing

2 Architecture

The system itself must repeatedly complete a simple task: downloading a copy of every web page in a list as fast as possible, and storing the result in a file (or database). This lends itself to a client-server, highly parallelised architecture, with the list maintaining global state through ACID transactions.

The list itself must be capable of very high throughput due to the large number of URLs available, and should be accessible through a system-wide (if not network agnostic) interface. SQLite3 is ideally suited to this style of processing, and is unlikely to be damaged by high-concurrency workers accessing it.

2.1 Parallelism

Downloading large amounts of data is a relatively common problem, and as such has been solved already in a number of libraries. The best of these appears to be Typhoeus (<https://github.com/pauldix/typhoeus>), which is a ruby wrapper around the cURL library for downloading. Typhoeus can handle SSL, and is capable of making many requests simultaneously.

Use of a library able to parallelise requests reduces the need for an architecture that can easily handle concurrency. As such, the sqlite3 database will be maintained by a single thread, which uses typhoeus to parallelise its requests to servers. This primarily has the benefit of avoiding competition between worker threads for the state of the server.

2.2 Storage

Two types of data must be stored for the project: the URL list itself (hereafter referred to as 'metadata', since it must contain a list for every download of every URL), and the contents of the HTTP request, including the body of the document (simply, 'data'). Updates of both must be ordered so that no data from sample n is updated after the start of sample $n + 1$, etc. This is easily accomplished with one download agent by simply killing the existing task prior to starting a new one (likely a job for cron).

Metadata must be stored in a fast, random-access format. As mentioned above, SQLite3 is well suited to this task due to its ability to be transparently memory-resident. A list of URLs must be stored, along with lists of sample sets and the success or failure of each.

Data will be significantly larger (perhaps terabytes), and is best stored in a folder hierarchy on a conventional file system. This also simplifies the process of backups. Data will contain the document body AND its HTTP headers. Whilst storing the headers in the metadata would also make sense (and make the data easier to export), it would make the metadata database grow significantly in size as the process runs.