BUbiNG: Massive Crawling for the Masses

Paolo Boldi  Andrea Marino  Massimo Santini  Sebastiano Vigna
Dipartimento di Informatica
Università degli Studi di Milano, Italy
{boldi,marino,santini,vigna}@di.unimi.it

ABSTRACT
Although web crawlers have been around for twenty years by now, there is virtually no freely available, open-source crawling software that guarantees high throughput, overcomes the limits of single-machine tools and at the same time scales linearly with the amount of resources available. This paper aims at filling this gap.

Categories and Subject Descriptors
H.3.4 [Information storage and retrieval]: Systems and software—World Wide Web (WWW)

1. INTRODUCTION
A web crawler is a system that downloads systematically a large number of web pages starting from a seed and following hypertextual links. In this paper we describe the design and implementation of BUbiNG, our new web crawler built upon the experience with UbiCrawler [1] and on the last ten years of research on the topic. BUbiNG main features are the following:

- It is pure Java and open source, released under the GNU GPLv3+ and available at the LAW web site.¹
- It is fully distributed: multiple agents perform the crawl concurrently and handle the necessary coordination without the need of any central control; given enough bandwidth, the crawling speed grows linearly with the number of agents.
- Its design acknowledges that CPUs and OS kernels have become extremely efficient in handling a large number of threads, and that large amounts of RAM are by now easily available at a moderate cost.
- It is very fast: on a 64-core, 64GB workstation it can download hundreds of million of pages at more than 9000 pages per second respecting politeness, analyzing, compressing and storing more than 140 MB/s of data.
- It guarantees that politeness intervals are satisfied both at the host and at the IP level, that is, that two data requests to the same host or IP are separated by at least a specified amount of time. The two intervals can be set independently, and, in principle, customized per host or IP.
- It guarantees that hostwise the visit is breadth first, and that also the global behavior is as close as possible to a breadth-first visit, taking politeness limits into account; moreover, the global policy can be easily customized.

For more details about previous works or the main issues in the design of crawlers, we refer the reader to [5].

2. DESIGN HIGHLIGHTS
BUbiNG stands on a few architectural choices which in some cases contrast the common folklore wisdom. We took our decisions after carefully benchmarking several options and gathering the hands-on experience of similar projects.

- The fetching logic of BUbiNG is built around thousands of identical fetching threads performing essentially only synchronous (blocking) I/O. Experience with recent Linux kernels and increase in the number of cores per machine shows that this approach consistently outperforms asynchronous I/O.
- Lock-free [3] data structures are used to “sandwich” fetching threads, so that they never have to access lock-based data structures. This approach is particularly useful to avoid direct access to synchronized data structures with logarithmic modification time, as contention between fetching threads can become very significant. Such structures are accessed by a single thread that enqueues the result of the slow-access operation to a lock-free queue, where any fetching thread can pick it up quickly.
- URL storage (both in memory and on disk) is entirely performed using byte arrays. While this approach might seem anachronistic, it pays off in terms of footprint (a String instance can occupy three times the memory of the corresponding byte array) and in terms of number of created objects.

¹The authors were supported by the EU-FET grant NA-DINE (GA 288956).
²http://law.di.unimi.it/


<table>
<thead>
<tr>
<th>Crawler</th>
<th>Machines</th>
<th>Resources (Millions)</th>
<th>Resources/s total</th>
<th>Resources/s per agent</th>
<th>Speed in MB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nutch (ClueWeb09)</td>
<td>100 (Hadoop)</td>
<td>1.200</td>
<td>430</td>
<td>4.3</td>
<td>10</td>
</tr>
<tr>
<td>Heritrix (ClueWeb12)</td>
<td>5</td>
<td>2.300</td>
<td>300</td>
<td>60</td>
<td>19</td>
</tr>
<tr>
<td>IRLBot</td>
<td>1</td>
<td>0.380</td>
<td>1790</td>
<td>1790</td>
<td>40</td>
</tr>
<tr>
<td>BUbiNG (iStella)</td>
<td>1</td>
<td>0.380</td>
<td>5400</td>
<td>5400</td>
<td>135</td>
</tr>
<tr>
<td>BUbiNG (in vitro)</td>
<td>4</td>
<td>1.000</td>
<td>36.600</td>
<td>9.150</td>
<td>584</td>
</tr>
</tbody>
</table>

Table 1: Comparison between BUbiNG and the main existing open-source crawlers. Resources are HTML URLs that have been emitted by the sieve. The main task of the distributor is to dequeue iteratively a URL from the sieve, checking whether it belongs to a host for which a visit state already exists, and then either creating a new visit state or enqueuing the URL to an existing one. If a new visit state is necessary, it is passed to a set of DNS threads that perform DNS resolution and then move the visit state on the workbench. Since, however, breadth-first visit queues grow exponentially, and the workbench can use only a fixed amount of in-core memory, it is necessary to virtualize the workbench, that is, writing on disk part of the URLs coming out of the sieve. In the first versions of BUbiNG, we tried designs inspired by the BEAST module of IRLbot [2], which however is only vaguely specified; moreover, BEAST-based implementations performed poorly unless we discarded all sites generating errors. Currently, BUbiNG uses a sophisticated memory-mapped system that can handle millions of on-disk FIFO queues by appending elements in a log-like fashion and periodically collecting unused space. Alternatively, if page-level prioritization is necessary, BUbiNG can virtualize the workbench using the Berkeley DB.

3. EXPERIMENTS

We ran two kinds of experiments: one batch was performed in vitro with a HTTP proxy simulating network connections towards the web and generating fake HTML pages. Four agents (with IP delay 500 ms and host delay 4 s) downloaded on average 36 600 pages per second. A second batch of experiments was run at iStella, an Italian commercial search engine that kindly provided us with a 48-core, 512 GB RAM machine with a 2 Gb/s link that we were able to fully saturate, downloading 5 400 pages per second using a single agent. Table 1 reports comparison with public data about other crawlers.

4. REFERENCES