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 9 000 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.

*The authors were supported by the EU-FET grant NADINE (GA 288956).
[1] `http://law.di.unimi.it/`

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2577304.

Crawler	Machines	Resources (Millions)	Resources/s		Speed in MB/s	
			overall	per agent	overall	per agent
Nutch (ClueWeb09)	100 (Hadoop)	1 200	430	4.3	10	0.1
Heritrix (ClueWeb12)	5	2 300	300	60	19	3.9
IRLBot	1	6 380	1 790	1 790	40	40
BUbiNG (iStella)	1	500	5 400	5 400	135	135
BUbiNG (*in vitro*)	4	1 000	36 600	9 150	584	146

Table 1: Comparison between BUbiNG and the main existing open-source crawlers. Resources are HTML pages for ClueWeb09 and IRLBot, but include other data types (e.g., images) for ClueWeb12. For reference, we also report the throughput of IRLbot [2], although the latter is not publicly available.

- Following UbiCrawler's design [1], BUbiNG agents are identical and autonomous. The assignment of URLs to agents is entirely customizable, but by default we use *consistent hashing* as a fault-tolerant, self-configuring assignment function.

We now provide a few highlights on data structures that are novel and central in the design of BUbiNG.

Workbench. It is an in-memory data structure that contains the next URLs to be visited. It is one of the main novel ideas in BUbiNG's design.

URLs associated with a specific host are kept in a structure called *visit state*, containing a FIFO queue of the next URLs to be crawled for that host along with a `next-fetch` field that specifies the first instant in time when a URL from the queue can be downloaded, according to the host politeness configuration. Visit states are further gathered by IP address in *workbench entries*: a workbench entry contains a queue of visit states sharing a common IP, prioritized by their `next-fetch` field, and an IP-specific `next-fetch`, containing the first instant in time when the IP address can be accessed again, according to the IP politeness configuration. The *workbench* is the queue of all workbench entries, prioritized on the `next-fetch` field of each entry *maximized* with the `next-fetch` field on the top element of its queue of visit states. In other words, the workbench is a priority queue of priority queues of FIFO queues. Due to our choice of priorities *there is a host that can be visited without violating host or IP politeness if and only if the first URL of the top visit state of the top workbench entry can be visited*. This approach improves significantly over IRLBot's two-queues technique [2], as it can detect in constant time the next URL to process.

Cache and sieve. To keep track of already-seen URLs, every time a URL is discovered it is checked against a high-performance approximate LRU cache containing 128-bit fingerprints: more than 90% of the URLs discovered are discarded at this stage. The cache has also another important goal: it avoids that frequently found URLs assigned to another agent are retransmitted several times. URLs that pass the cache check are enqueued to a *sieve*, a data structure originally used by Mercator [4] that stores fingerprints of the set of seen URLs on disk and merges them periodically with a set accumulated in RAM, emitting new URLs that must be crawled. We tested an alternative sieve described in [2], the *DRUM* (a extension of the Mercator sieve) but DRUM destroys the breadth-first order of the visit, and we found no performance advantages with respect to a standard Mercator sieve coupled with our cache.

Distributor. It is a high-priority thread that processes 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

[1] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[2] H. Lee, D. Leonard, X. Wang, and D. Loguinov. Irlbot: Scaling to 6 billion pages and beyond. *ACM Trans. Web*, 3(3):8:1–8:34, July 2009.

[3] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM PODC*, pages 267–275. ACM, 1996.

[4] M. Najork and A. Heydon. High-performance web crawling. In James Abello, Panos M. Pardalos, and Mauricio G. C. Resende, eds., *Handbook of massive data sets*, pages 25–45. Kluwer Academic Publishers, 2002.

[5] C. Olston and M. Najork. Web crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010.