UbiCrawler: A Scalable Fully Distributed Web Crawler

Paolo Boldi[*] Bruno Codenotti[†] Massimo Santini[‡] Sebastiano Vigna[§]

Abstract

We present the design and implementation of UbiCrawler, a scalable distributed web crawler, and we analyze its performance. The main features of UbiCrawler are platform independence, fault tolerance, a very effective assignment function for partitioning the domain to crawl, and more in general the complete decentralization of every task.

1 Introduction

In this paper we present the design and implementation of UbiCrawler, a scalable, fault-tolerant and fully distributed web crawler, and we evaluate its performance both *a priori* and *a posteriori*. The overall structure of the UbiCrawler design was preliminarily described in [2][1] and [1].

It has been recognized that *as the size of the web grows, it becomes imperative to parallelize the crawling process, in order to finish downloading pages in a reasonable amount of time* [5]. Nonetheless, little published work actually investigates the fundamental issues underlying the parallelization of the different tasks involved with the crawling process. Some features of Google have been presented in [3], where the crawling mechanism is described as a two stage process:

- a URL server distributes individual URLs to multiple crawlers, which download web pages in parallel;
- the crawlers then send the downloaded pages to a central indexer, on which links are extracted and sent via the URL server to the crawlers.

In contrast, when designing UbiCrawler, we have decided to decentralize every task, with obvious advantages in terms of scalability and fault tolerance.

Essential features of UbiCrawler are

- platform independence;
- full distribution of every task (no single point of failure and no centralized coordination at all);
- tolerance to failures: permanent as well as transient failures are dealt with gracefully;
- scalability.

As outlined in Section 2, these features are the offspring of a well defined design goal: fault tolerance and full distribution (lack of any centralized control) are assumptions which have guided our architectural choices. For instance, while there are several reasonable ways to partition the domain to be crawled if we assume the presence of a central server, it becomes harder to find an assignment of URLs to different agents which is fully distributed, does not require too much coordination, and allows us to cope with failures.

In Section 3 we introduce the reader to the design of Ubicrawler, and make clear how it meets the above requirements.

Ubicrawler has been used to explore various portions of the WEB, e.g., `.it`, `.eu.int`, and the African domain; the corresponding data are available at `http://ubi.imc.pi.cnr.it/projects/ubicrawler/` (see also [1] for an analysis of the African Web).

[*]Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, via Comelico 39/41, I-20135 Milano, Italy. `boldi@dsi.unimi.it`

[†]Istituto di Matematica Computazionale, Consiglio Nazionale delle Ricerche, Via Moruzzi 1, I-56010 Pisa, Italy. `codenotti@imc.pi.cnr.it`

[‡]Dipartimento di Scienze Sociali, Cognitive e Quantitative, Università di Modena e Reggio Emilia, via Fratelli Manfredi I-42100 Reggio Emilia, Italy. `msantini@unimo.it`

[§]Contact author. Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, via Comelico 39/41, I-20135 Milano, Italy. `vigna@acm.org`. Phone: +39-0258356324. Fax: +39-0258356373.

[1]At the time, the name of the crawler was *Trovatore*, later changed into UbiCrawler when the authors learned about the existence of an Italian search engine named Trovatore.

2 Design Assumptions, Requirements, and Goals

In this section we give a brief presentation of the most important design choices which have guided the implementation of Ubicrawler. More precisely, we sketch general design goals and requirements, as well as assumptions on the type of faults that should be tolerated.

In the next sections, we will describe the architecture of UbiCrawler and show its adherence to the features outlined here.

Full distribution. In order to achieve significant advantages in terms of programming, deployment, and debugging, a parallel and distributed crawler should be composed by identically programmed agents, distinguished by a unique identifier only. This has a fundamental consequence: each task must be performed in a fully distributed fashion, that is, no central coordinator can exist.

We also do not want to rely on any assumption concerning the location of the agents, and this implies that latency can become and issue, so that we should minimize communication to reduce it.

Balanced locally computable assignment. The distribution of URLs to agents is an important issue, crucially related to the efficiency of the distributed crawling process.

We identify the three following goals:

- At any time, each URL should be assigned to a specific agent, which is solely *responsible* for it.
- For any given URL, the knowledge of its responsible agent should be locally available. In other words, every agent should have the capability to compute the identifier of the agent responsible for a URL, without communicating.
- The distribution of URLs should be *balanced*, that is, each agent should be responsible for approximately the same number of URLs.

Scalability. The number of pages crawled per second per agent should be (almost) independent of the number of agents. In other words, we expect the throughput to grow linearly with the number of agents.

No host overload. A parallel crawler should never try to fetch more than one page at a time from a given host.

Fault tolerance. A distributed crawler should continue to work under *crash faults*, that is, when some agents abruptly die. No behaviour can be assumed in the presence of this kind of crash, except that the faulty agent stops communicating; in particular, one cannot prescribe any action to a crashing agent, or recover its state afterwards[2]. When an agent crashes, the remaining agents should continue to satisfy the "Balanced locally computable assignment" requirement: this means, in particular, that URLs will have to be redistributed.

This has two important consequences:

- It is not possible to assume that URLs are statically distributed.
- Since the "Balanced locally computable assignment" requirement must be satisfied *at any time*, it is not reasonable to rely on a distributed reassignment protocol after a crash. Indeed, during the protocol the requirement would be violated.

3 The Software Architecture

UbiCrawler is composed by several agents that autonomously coordinate their behaviour in such a way that each of them scans its share of the web. An agent performs its task by running several threads, each dedicated to the visit of a single host. More precisely, each thread scans a single host using a breadth-first visit. We make sure that different threads visit different hosts at the same time, so that each host is not overloaded by too many requests. The outlinks that are not local to the given host are dispatched to the right agent, which puts them in the queue of pages to be visited. Thus, the overall visit of the web is breadth first, but as soon as a new host is met, it is entirely visited (possibly with bounds on the depth reached or on the overall number of pages), again in a breadth-first fashion.

[2]Note that this is radically different from milder assumptions, as for instance saying that the state of a faulty agent can be recovered. In the latter case, one can try to "mend" the crawler's global state by analyzing the state of the crashed agent.

More sophisticated approaches (which can take into account suitable priorities related to URLs, such as, for instance, their rank) can be easily implemented. However it is worth noting that several authors (see, e.g., [11]) have argued that breadth-first visits tends to find high quality pages early on in the crawl. A deeper discussion about page quality is given in Section 5.

Assignment of hosts to agents takes into account the mass storage resources and bandwidth available at each agent. This is currently done by means of a single indicator, called *capacity*, which acts as a weight used by the assignment function to distribute hosts. Under certain circumstances, each agent a gets a fraction of hosts proportional to its capacity C_a (see Section 4 for a precise description of how this works). Note that even if the number of URLs per host varies wildly, the distribution of URLs among agents tends to even out during large crawls. Besides empirical statistical reasons for this, there are also other motivations, such as the usage of policies for bounding the maximum number of pages crawled from a host and the maximum depth of a visit. Such policies are necessary to avoid (possibly malicious) *web traps*.

Finally, an essential component in UbiCrawler is a *reliable failure detector* [4], that uses timeouts to detect crashed agents; reliability refers to the fact that a crashed agent will eventually be distrusted by every active agent (a property that is usually referred to as *strong completeness* in the theory of failure detectors). The failure detector is the only synchronous component of UbiCrawler (i.e., the only component using timings for its functioning); all other components interact in a completely asynchronous way.

To achieve platform-independence, we have chosen Java$^{\text{TM}}$ 2 as implementation language. Currently UbiCrawler consists of more than ninety classes and interfaces, made of more than 16 000 lines of code and generating almost a megabyte of bytecode.

4 The Assignment Function

In this section we describe the assignment function used by UbiCrawler, and we explain why it makes it possible to decentralize every task and to achieve our fault tolerance goals.

Let $\mathscr{A}$ be our set of agent identifiers (i.e., potential agent names), and $\mathscr{L} \subseteq \mathscr{A}$ be the set of alive agents: we have to assign hosts to agents in $\mathscr{L}$. More precisely, we have to set up a function δ that, for each nonempty set $\mathscr{L}$ of alive agents, and for each host h, delegates the responsibility of fetching h to the agent $\delta_{\mathscr{L}}(h) \in \mathscr{L}$.

The following properties are desirable for an assignment function:

1. *Balancing.* Each agent should get approximately the same number of hosts; in other words, if m is the (total) number of hosts, we want that $\left|\delta_{\mathscr{L}}^{-1}(a)\right| \sim m/|\mathscr{L}|$ for each $a \in \mathscr{L}$.
2. *Contravariance.* The set of hosts assigned to an agent should change in a contravariant manner with respect to the set of alive agents across a deactivation and reactivation. More precisely, if $\mathscr{L} \subseteq \mathscr{L}'$ then $\delta_{\mathscr{L}}^{-1}(a) \supseteq \delta_{\mathscr{L}'}^{-1}(a)$; that is to say, if the number of agents grows, the portion of the web crawled by each agent must shrink. Contravariance has a fundamental consequence: if a new set of agents is added, no old agent will ever lose an assignment in favour of another old agent; more precisely, if $\mathscr{L} \subseteq \mathscr{L}'$ and $\delta_{\mathscr{L}'}(h) \in \mathscr{L}$ then $\delta_{\mathscr{L}'}(h) = \delta_{\mathscr{L}}(h)$; this guarantees that at any time the set of agents can be enlarged with minimal interference with the current host assignment.

Note that satisfying partially the above requirement is not difficult: for instance, a typical approach used in non-fault-tolerant distributed crawlers is to compute a modulo-based hash function of the host name. This has very good balancing properties (each agent gets approximately the same number of hosts), and certainly can be computed locally by each agent knowing just the set of alive agents.

However, what happens when an agent crashes? The assignment function can be computed again, giving however a different result for almost all hosts. The *size* of the sets of hosts assigned to each agent would grow or shrink contravariantly, but the *content* of those sets would change in a completely chaotic way. As a consequence, after a crash most pages will be stored by an agent that should not have fetched them, and they could mistakenly be re-fetched several times[3].

Clearly, if a central coordinator is available or if the agents can engage a kind of "resynchronization phase" they could gather other information and use other mechanisms to redistribute the hosts to crawl. However, we would have just shifted the fault-tolerance problem to the resynchronization phase—faults in the latter would be fatal.

4.1 Background

Although it is not completely obvious, it is not difficult to show that contravariance implies that each possible host induces a total order (i.e., a permutation) on $\mathscr{A}$; more precisely, a contravariant assignment is equivalent to a function that assigns an element of $S_{\mathscr{A}}$ (the symmetric group over $\mathscr{A}$, i.e., the set of all permutations elements of $\mathscr{A}$, or equivalently, the set of all total orderings of

[3]For the same reason, a modulo-based hash function would make it difficult to increase the number of agents during a crawl.

elements of $\mathscr{A}$) to each host: then, $\delta_{\mathscr{L}}(h)$ is computed by taking, in the permutation associated to h, the first agent that belongs to the set $\mathscr{L}$.

A simple technique to obtain a balanced, contravariant assignment function consists in trying to generate such permutations, for instance, using some bits extracted from a host name to seed a (pseudo)random generator, and then permuting randomly the set of possible agents. This solution has the big disadvantage of running in time and space proportional to the set of possible agents (which one wants to keep as large as feasible). Thus, we need a more sophisticated approach.

4.2 Consistent Hashing

Recently, a new type of hashing called *consistent hashing* [8, 9] has been proposed for the implementation of a system of distributed web caches (a different approach to the same problem can be found in [6]). The idea of consistent hashing is very simple, yet profound.

As we noted, for a typical hash function, adding a bucket (i.e., a new place in the hash table) is a catastrophic event. In consistent hashing, instead, each bucket is replicated a fixed number κ of times, and each copy (we shall call it a *replica*) is mapped randomly on the unit circle. When we want to hash a key, we compute in some way from the key a point in the unit circle, and find its nearest replica: the corresponding bucket is our hash. The reader is referred to [8] for a detailed report on the powerful features of consistent hashing, which in particular give us balancing for free. Contravariance is also easily verified.

In our case, buckets are agents, and keys are hosts. We must be very careful, however, if we want the contravariance (2) to hold, because mapping randomly the replicas to the unit circle each time an agent is started will not work; indeed, δ would depend not only on $\mathscr{L}$, but also on the choice of the replicas. Thus, *all* agents should compute the same set of replicas corresponding to a given agent, so that, once a host is turned into a point of the unit circle, all agents will agree on who is responsible for that host.

4.3 Identifier–Seeded Consistent Hashing

A method to fix the set of replicas associated to an agent and try to maintain the good randomness properties of consistent hashing is to derive the set of replicas from a very good random number generator seeded with the agent identifier: we call this approach *identifier-seeded consistent hashing*. We have opted for the Mersenne Twister [10], a fast random generator with an extremely long cycle that passes very strong statistical tests.

However this solution imposes further constraints: since replicas cannot overlap, any discretization of the unit circle will incur in the Birthday paradox—even with a very large number of points, the probability that two replicas overlap will become non-negligible. Indeed, when a new agent is started, its identifier is used to generate the replicas for the agent. However, if during this process we generate a replica that is already assigned to some other agent, we must force the new agent to choose another identifier.

This solution might be a source of problems if an agent goes down for a while and discovers a conflict when it is restarted. Nonetheless, some standard probability arguments show that with a 64-bit representation for the elements of the unit circle there is room for 10^4 agents with a conflict probability of 10^{-12}.

We remark that a theoretical analysis of the balancing produced by identifier-seeded consistent hashing is most difficult, if not impossible (unless, of course, one uses the working assumption that replicas behave as if randomly distributed). Thus, we report experimental data: in Figure 1 one can see that once a substantial number of hosts have been crawled, the deviation from perfect balancing is less than 6% for small as well as for large sets of agents when $\kappa = 100$, that is, we use 100 replicas per bucket (thin lines); if $\kappa = 200$, the deviation decreases to 4.5% (thick lines).

We have implemented consistent hashing as follows: the unit interval can be mapped on the whole set of representable integers, and then replicas can be kept in a balanced tree whose keys are integers. This allows us to hash a host in logarithmic time (in the number of alive agents). By keeping the leaves of the tree in a doubly linked chain we can also easily implement the search for the next nearest replica.

As we already mentioned, an important feature of UbiCrawler is that it can run on heterogeneous hardware, with different amount of available space. To this purpose, the number of replicas generated for an agent is multiplied by its capacity, and this guarantees that the assignment function distributes hosts evenly with respect to the mass storage available at each agent.

Moreover, the number of threads of execution for each agent can be tuned to suit network bandwidth or CPU limitations. Note however that an excessive number of threads can lead to contention on shared data structures, such as the store, and to excessive CPU load, with corresponding performance degradation.

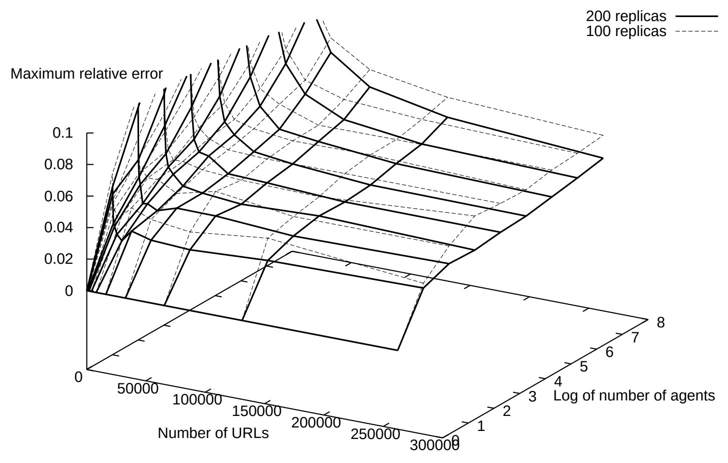

Figure 1: Experimental data on identifier-seeded consistent hashing. Deviation from perfect balancing is less than 6% with 100 replicas (thin lines), and less than 4.5% with 200 replicas (thick lines).

5 Performance Evaluation

The goal of this section is to discuss UbiCrawler in the framework of the classification given in [5], and to analyze its scalability and fault-tolerance features. In particular, we consider the most important features identified by [5] (degree of distribution, coordination, partitioning techniques, coverage, overlap, and communication overhead) and contrast UbiCrawler against them.

Degree of distribution. A parallel crawler can be intra-site, or distributed, that is, its agents can communicate either through a LAN or through a WAN. UbiCrawler is a distributed crawler which can run on any kind of network.

Coordination. In the classification of [5], agents can use a different amount of coordination: at one extreme, all agents crawl the network independently, and one hopes that their overlap will be small due to a careful choice of the starting URLs; at the other extreme, a central coordinator divides the network either *statically* (i.e., before the agents actually start) or *dynamically* (i.e., during the crawl).

As for UbiCrawler, the assignment function gives rise to a kind of coordination that does not fit the models and the options suggested above. Indeed, the coordination is dynamic, but there is no central authority that handles it. Thus, in a sense, all agents run independently, but they are at the same time tightly and distributedly coordinated. We call this feature *distributed dynamic coordination*.

Partitioning techniques. The web can be partitioned in several ways; in particular, the partition can be obtained from URL-based hash, host-based hash or hierarchically, using, for instance, Internet domains. Currently, UbiCrawler uses a host-based hash; note that since [5] does not consider consistent hashing, some of the arguments about the shortcomings of hashing functions are no longer true for UbiCrawler.

Coverage. It is defined as $\frac{c}{u}$, where c is the number of actually crawled pages, and u the number of pages the crawler as a whole had to visit.

If no faults occur, UbiCrawler achieves coverage 1, which is optimal. Otherwise, it is in principle possible that some URLs that were stored locally by a crashed agent will not be crawled. However, if these URLs are reached along other paths after the crash, they will clearly be fetched by the new agent responsible for them.

Overlap. It is defined as $\frac{n-u}{u}$, where n is the total number of crawled pages and u the number of *unique* pages (sometimes $u < n$ because the same page has been erroneously fetched several times).

Even in the presence of faults, UbiCrawler achieves overlap 0, which is optimal. However, if an agent crashes and it is restarted after a while with part of its state recovered (i.e., crawled pages), we cannot guarantee the absence of duplications, and thus overlap can be greater than zero. Nevertheless, note that after such an event UbiCrawler autonomously tries to converge to a state with overlap 0 (see Section 5.1.1). This property is usually known as *self-stabilization*, a technique for protocol design introduced by Dijkstra [7].

Communication overhead. It is defined as $\frac{e}{n}$, where e is the number of URLs exchanged by the agents during the crawl and n is the number of crawled pages.

Assuming, as reported in [5], that on the average every page contains just one link to another site [5], we have that n crawled pages will give rise to n URLs that must be potentially communicated to other agents[4]. Due to the balancing property of the assignment function, at most

$$n\frac{\sum_{a\neq\bar{a}} C_a}{\sum_a C_a} < n$$

messages will be sent across the network, where a ranges in the set of alive agents, and $\bar{a}$ is the agent that fetched the page (recall that C_a is the capacity of agent a). By the definition of [5], our communication overhead is thus less than 1, a fact that has been confirmed by our experimental analysis. Another interesting feature is that the number of messages is *independent of the number of agents*, and depends only on the number of crawled pages. In other words, a large number of agents will generate more network traffic, but this is due to the fact that they are fetching more pages, and not to a design bottleneck.

Quality. It is a complex measure of 'importance" or "relevance" of crawled pages based on ranking techniques; an important challenge is to build a crawler that tends to collect high-quality pages first.

As we already mentioned, currently UbiCrawler uses a parallel per-host breadth-first visit, without dealing with ranking and quality of page issues. This is because our immediate goal is to focus on scalability of the crawler itself and on the analysis of some portions of the web, as opposed to building a search engine. Nonetheless, since a breadth-first single-process visit tends to visit high-quality pages first [11], it is natural to ask whether our strategy works well or not.[5]

We have used a very rough yet reasonable quality measure, namely, the number of incoming links, and computed how the average quality of visited pages changes in time. Figure 2 shows for example the average indegree during a crawl of the domain `.eu.int`; somehow surprisingly, UbiCrawler has a very good performance, even though our strategy is far from being a plain breadth-first visit (see Section 3).

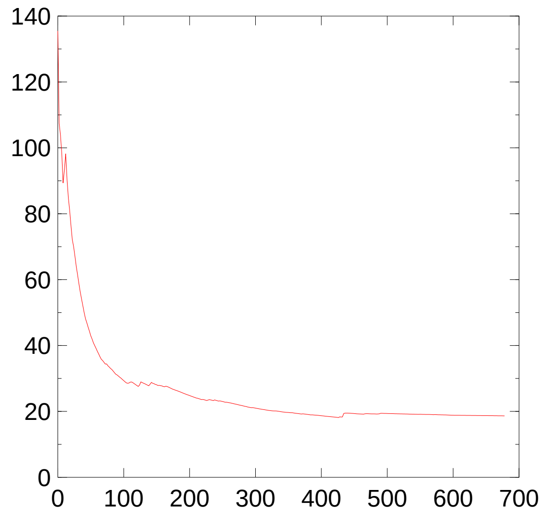

Figure 2: Quality of crawled pages (average indegree) as a function of the number of crawled URLs (× 1 000).

5.1 Fault Tolerance

To the best of our knowledge, no commonly accepted metrics exist for estimating the fault tolerance of distributed crawlers, since the issue of faults has not been taken into serious consideration up to now. It is indeed an interesting and open problem to define

[4]Note that in principle not all URLs must be necessarily communicated to other agents; one could just rely on the choice of a good seed to guarantee that no pages will be lost. Nonetheless, in a worst-case scenario, to obtain coverage 1 all URLs not crawled *must* be communicated to some other agent.

[5]Of course, it will be possible to order pages according to a ranking function, using, for instance, backlink information, at a later stage of this project.

a set of measures to test the robustness of parallel crawlers in the presence of faults. Thus, we give an overview of the reaction of UbiCrawler agents to faults.

UbiCrawler agents can die or become unreachable either expectedly (for instance, for maintenance) or unexpectedly (for instance, because of a network problem). At any time, each agent has its own view of which agents are alive and reachable, and these views do not necessarily coincide.

Whenever an agent dies abruptly, the failure detector discovers that something bad has happened (e.g., using timeouts). Thanks to the properties of the assignment function, the fact that different agents have different views of the set of alive agents does not disturb the crawling process. Suppose, for instance, that a knows that b is dead, whereas a' does not. Because of contravariance, the only difference between a and a' in assignments of host to agents is the set of hosts pertaining to b. Agent a correctly dispatches these hosts to other agents, and agent a' will do the same as soon as it realizes that b is dead, which will happen, in the worst case, when it tries to dispatch a URL to b. At this point, b will be believed dead, and the host dispatched correctly. Thus, a and a' will never dispatch hosts to different agents.

Another consequence of this design choice is that agents can be dynamically added during a crawl, and after a while all pages for which they are responsible will be removed from the stores of the agents that fetched them before the new agent's birth. In other words, making UbiCrawler self-stabilizing by design gives us not only fault tolerance, but also a greater adaptivity to dynamical configuration changes.

5.1.1 Page Recovery

An interesting feature of contravariant assignment functions is that they allow to guess easily who could have fetched previously a page for which an agent is responsible in the present configuration. Indeed, if a is responsible for the host h, then the agent responsible for *h before a was started* is the one associated to the next-nearest replica. Indeed, this allows us to implement a *page recovery protocol* in a very simple way. Under certain conditions, the protocol allows to avoid re-fetching several times the same page even in the presence of faults.

The system is parametrized by an integer t: each time an agent is going to fetch a page of a host for which it is currently responsible, it first checks whether the next-nearest t agents have already fetched that page. It is not difficult to prove that this guarantees page recovery as long as *no more than t agents were started since the page was crawled*. Note that the number of agents that crashed is completely irrelevant.

This approach implies that if we want to accept t (possibly transient) faults without generating overlap, we have to increase by a linear factor of t the network traffic, as any fetched page will generate at least t communications. This is not unreasonable, as in distributed system it is typical that at a number of rounds linearly related to the maximum number of faults is required to solve, for instance, consensus.

5.2 Scalability

In a highly scalable system, one should guarantee that the work performed by every thread is constant as the number of threads changes, i.e., that the system and communication overhead does not reduce the performance of each thread. The figures given in Section 5 show that the amount of network communication grows linearly with the number of downloaded pages, a fact which implies that the performance of each UbiCrawler thread is essentially independent of the number of agents. We have measured how the average number of pages stored per second and thread changes when the number of agents, or the number of threads per agent, changes. Figure 3 plots the resulting data for the African domain.

Graph (a) shows how work per thread changes when the number of agents increases. In a perfectly scalable system, all lines should be horizontal (which would mean that by increasing the number of agents we could arbitrarily accelerate the crawling process). There is a slight drop in the second part of the first graph, that becomes significant with eight threads. The drop in work, however, is in this case an artifact of the test, caused by our current limitations in terms of hardware resources: to run experiments using more than seven agents, we had to start two agents per machine, and the existence of so many active processes unbearably raised the CPU load, and led to hard disk thrashing. We have decided to include anyway the data because they show almost constant work for a smaller number of threads and for less than eight agents.

Graph (b) shows how work per thread changes when the number of threads per agent increases. In this case, data contention, CPU load and disk thrashing become serious issues, and thus the work performed by each single thread reduces. The drop in work, however, is strongly dependent on the hardware architecture and, again, the reader should take with a grain of salt the lower lines, which manifest the artifact already seen in graph (a).

Just to make these graphs into actual figures, note that a system with sixteen agents, each running four threads, can fetch about 4 500 000 pages a day, and we expect these figures to scale almost linearly with the number of agents, if sufficient network bandwidth is available. Further tests with more than fifty agents showed that this is certainly true when the number of threads per agent is small.

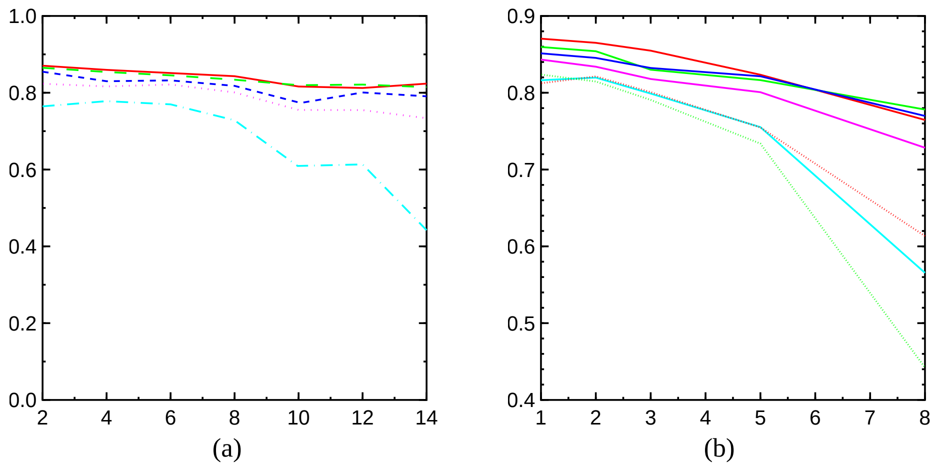

Figure 3: Average number of pages crawled per second and thread, that is, work per thread. Graph (a) shows how work changes when the number of agents changes; the different patterns represent the number of threads (solid line=1, long dashes=2, short dashes=3, dots=5, dash-dots=8). Graph (b) shows how work changes when the number of threads changes; the different lines represent a different number of agents (from 2 to 14, higher to lower).

6 Conclusions

We have presented UbiCrawler, a fully distributed, scalable and fault-tolerant web crawler. We believe that UbiCrawler introduces new ideas in parallel crawling, in particular the use of consistent hashing as a mean to completely decentralize the coordination logic, graceful degradation in the presence of faults, and linear scalability.

We plan to continue the development and deployment of UbiCrawler, and to test its performance in very large web domains.

References

[1] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Structural properties of the african web. submitted as poster to the 11th International World–Wide Web Conference.

[2] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Trovatore: Towards a highly scalable distributed web crawler. In *Proc. of 10th International World–Wide Web Conference*, Hong Kong, China, 2001. Poster session (Winner of the Best Poster Award).

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1/7):107–117, 1998.

[4] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[5] Junghoo Cho and Hector Garcia-Molina. Parallel crawlers. In *Proc. of the 11th International World–Wide Web Conference*, 2002.

[6] Robert Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In David B. Lomet, editor, *Proc. Foundations of Data Organization and Algorithms, 4th International Conference, FODO'93*, volume 730 of *Lecture Notes in Computer Science*, pages 101–114, Chicago, Illinois, USA, 1993. Springer–Verlag.

[7] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[8] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, 1997.

[9] David Karger, Tom Leighton, Danny Lewin, and Alex Sherman. Web caching with consistent hashing. In *Proc. of 8th International World–Wide Web Conference*, Toronto, Canada, 1999.

[10] M. Matsumoto and T. Nishimura. Mersenne twister: A 623–dimensionally equidistributed uniform pseudo–random number generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998.

[11] Marc Najork and Janet L. Wiener. Breadth-first search crawling yields high-quality pages. In *Proc. of 10th International World Wide Web Conference*, Hong Kong, China, 2001.