

# UbiCrawler

## Scalability And Fault-Tolerance Issues

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

Bruno Codenotti  
Istituto di Informatica e Telematica  
Consiglio Nazionale delle Ricerche  
Via Moruzzi 1  
I-56010 Pisa, Italy  
codenotti@imc.pi.cnr.it

Massimo Santini  
Dipartimento di scienze sociali, cognitive e quantitative  
Università di Modena e Reggio Emilia  
Via Fratelli Manfredi  
I-42100 Reggio Emilia, Italy  
msantini@unimo.it

Sebastiano Vigna  
Dipartimento di Scienze dell'Informazione  
Università degli Studi di Milano  
Via Comelico 39/41  
I-20135 Milano, Italy  
vigna@acm.org

### **ABSTRACT**

We report on the implementation of UbiCrawler, a scalable distributed web crawler, analyze its performance, and describe its fault tolerance.

### **Keywords**

distributed crawling, consistent hashing, fault tolerance, web graph

### **1. INTRODUCTION**

In this poster we report on the implementation of UbiCrawler, a scalable, fault-tolerant and fully distributed web crawler, and we give some data on its performance. The overall structure of the UbiCrawler design was preliminarily described in [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). Full details about the implementation of UbiCrawler can be found in [2], and at <http://ubi.imc.pi.cnr.it/projects/ubicrawler/>.

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.

## 2. 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 at a time (no host is ever visited by two threads at the same time). Assignment of hosts to agents takes into account the mass storage resources and bandwidth available at each agent.

With the above used term "autonomously" we mean that when a new agent is started, it can briefly communicate with another living agent to get the list of currently living agents, and announce to them that it has been started. However, after this event, *all agents must run independently, without any centralized control*; they can just communicate some URLs that should be fetched, and no other information. This constraint is instrumental in achieving fault tolerance: should an agent crash, the remaining agents will be able to decide who should fetch a certain host *without exchanging information and without using a central coordinator*. This means that the only information available to them is the set of living agents. In distributed systems language, we assume *crash faults*, in which agents can abruptly die; no behaviour can be assumed in the presence of such a fault, except that the agent stops communicating (in particular, one cannot prescribe actions to a crashing agent, or recover its state afterwards).

Thus, the most relevant theoretical ingredient in the design of UbiCrawler is the *assignment function*, that is, the function that, knowing just the set of living agents, assigns each host to a living agent.

## 3. THE ASSIGNMENT FUNCTION

Let  $A$  be our set of agent identifiers (i.e., potential agent names), and  $L$  be the subset of living agents: we have to assign hosts to agents in  $L$ . More precisely, we have to set up a function  $\delta$  that, for each nonempty set  $L$  of living agents, and for each host  $h$ , delegates the responsibility of fetching  $h$  to the agent  $\delta_L(h)$  in  $L$ .

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 by each agent knowing just the set of living agents. However, what happens when an agent crashes? The assignment function can be computed again, giving however a different result for most hosts. 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. 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.

To solve this problem, we need an assignment function that handles gracefully changes in the set of living agents. We isolated the fundamental property of such a function:

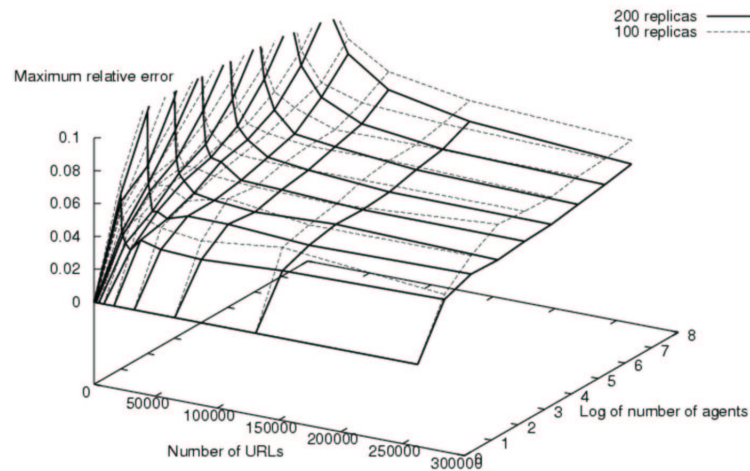
An assignment function is *contravariant* if the portion of the web crawled by each agent shrinks whenever the set of living agents grows. More precisely, if  $L$  is a subset of  $M$  then  $\delta_M^{-1}(a)$  must be a subset of  $\delta_L^{-1}(a)$ .

Contravariance has a fundamental consequence: if a new set of agents is added, no previously running agent will ever lose an assignment in favour of another previously running agent; more precisely, if  $L$  is a subset of  $M$  and  $\delta_M(h)$  is in  $L$  then  $\delta_M(h) = \delta_L(h)$ ; this guarantees that at any time the set of agents can be enlarged with minimal interference with the current host assignment.

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 the set of possible agents. 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 randomly permuting 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 ones wants to keep large).

Recently, a new type of hashing called *consistent hashing* [3,4] has been proposed for the implementation of a system of distributed web caches. The idea of consistent hashing is very simple, yet profound: each agent is replicated a fixed number  $k$  of times, and each copy (we shall call it a *replica*) is mapped randomly on the unit circle. When we want to hash a host, we compute in some way from the host a point in the unit circle, find its nearest replica, and take the corresponding agent.

UbiCrawler uses consistent hashing, but computes the random placement of the replicas of an agent using a pseudorandom generator seeded with some bits extracted from the (unique) identifier of the agent. In this way, all agents are always able to compute the same replica placement, even if no communication occurs after the startup phase. For more details, we refer the reader to [2].



**Figure 1: Experimental data on consistent hashing. Once a substantial number of hosts have been crawled, the deviation of consistent hashing from perfect balancing is less than 6% when  $k=100$  and less than 4.5% for  $k=200$ .**

#### 4. 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 account up to now. 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, standard connection-timeout mechanisms let the other agents know that something bad has happened. However, thanks to the properties of the assignment function, the fact that different agents have different views of the set of living agents does not interfere with the crawling process.

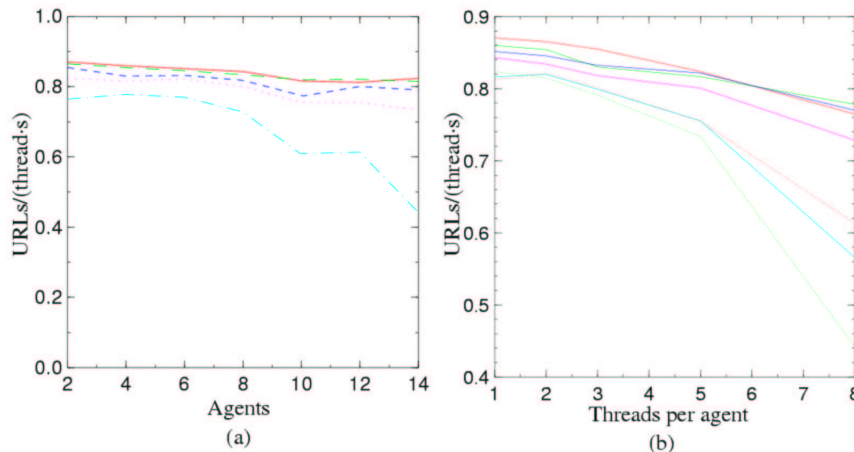
Suppose, for instance, that  $a$  knows that  $b$  is dead, whereas  $a'$  does not. Because of contravariance, the only difference between  $a$  and  $a'$  in assignments of hosts to agents is the set of hosts pertaining to  $b$ . Agent  $a$  correctly dispatches URLs from these hosts to other agents, 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 URL dispatched correctly. Thus,  $a$  and  $a'$  will never dispatch URLs from the same host to different agents.

## 5. 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. 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 2 shows some data we have obtained.

Graph (a) shows how work per thread changes when the number of agents increases. In a perfectly scalable system, all lines should be horizontal. 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 (we had to multiplex several agents on the same machine, causing disk thrashing).

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. The reader should take with a grain of salt the lower lines, which show the artifact already seen in graph (a).



**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).**

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 (this includes HTML parsing and word extraction), 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.

## 6. REFERENCES

1. Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Trovatore: Towards a highly scalable distributed web crawler. In *Poster Proc. of Tenth International World Wide Web Conference*, Hong Kong, China, 2001, pages 140–141.
2. Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. Technical Report, Università degli Studi di Milano, Dipartimento di Scienze dell'Informazione, 2002. Submitted for publication.
3. 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 *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, 1997.
4. David Karger, Tom Leighton, Danny Lewin, and Alex Sherman. Web caching with consistent hashing. In *Proc. of WWW8*, Toronto, Canada, 1999.