The Design and Implementation of Erachnid: an Extensible, Scalable Web Crawler in Erlang

David Sheets (dsheets@mit.edu)

6.UAP

Lalana Kagal

May 22, 2009

1 Introduction

With the explosive growth of the World Wide Web in the last 15 years, the task of processing and analyzing Web-scale resources has become much more complex. As of March 5, 2009, the most popular Web search engine, Google, has indexed approximately 17 billion active web resources. Fortunately, web-scale processing, storage, and bandwidth resources have become available in the form of "cloud computing" such as Amazon's Elastic Compute Cloud (EC2). Because these resources are able to be provisioned dynamically, smaller players can pay for only what they need, when they need it. To take full advantage of this capability, however, software must be designed with Web-scalability in mind. To accomodate niche use cases that are now possible with cloud resources, this Web-scalable software must be extensible.

This research project produced a web crawler, Erachnid, that can easily be scaled up or down and run on any combination of cloud and local resources. Additionally, the design of

Erachnid is sufficiently modular so that it can be adapted to a variety of uses in production and research environments.

2 Crawling Challenges

2.1 Overfetching/Refetching

To prevent the accidental duplicate fetching of a resource, a fast resource status look-up service is required. Many resources and clusters of resources contain multiple identical URLs. A naive crawler would request a resource for every URL it extracts. This results in increased bandwidth cost. Additionally, overfetching incurs an opportunity cost as newly discovered content stays unfetched while already fetched content is reacquired.

Refetching content in a timely manner is important if the consumers of the crawler's data need up-to-date information. Search engine indexes, web site monitoring systems, and event tracking systems all require fresh information. To schedule refetches, crawlers can either fetch the least recently fetched (naive) or adjust refetch frequency according to the change velocity and perceived value of content. The refetch algorithm is typically highly dependent on the nature of the crawl. The Erachnid prototype does not contain a refetch algorithm.

2.2 Request Throttling

As a web crawler consumes more aggregate bandwidth in its operation, it becomes increasingly important that its request frequency is reasonable. With no request throttling, a distributed web crawler pointed at a single site is essentially a denial of service attack. The greater the request throughput capability of the crawler, the more easily the crawler can take sites offline through careless requests. To be a respectful web bot, all crawlers must take steps to limit the maximum number of requests issued to a single site in a given period

of time.

2.3 Dynamic Content

As the Web evolves, HTTP, the Web's network protocol, is increasingly used to request programmatically generated content instead of simple static resources. For crawlers, this dynamic content can be difficult to handle because of its variable URL structure. Many dynamic resources link to other dynamic resources of the same type. If a crawler uses only simple URL canonicalization, it may generate an exponential number of resources to be crawled when faced with a simple web application.

2.4 Prioritizing Downloads

Crawler bandwidth is a limited resource and the Web is constantly expanding and changing. To make the most of a crawler's bandwidth, prioritization of resource requests is required. In many cases, this prioritization algorithm is specific to the crawl application and a highly tuned algorithm is typically complex.

3 Crawler Design

To meet the challenges of the modern Web crawler, Erachnid was implemented in Erlang for its lightweight processes and rich networking libraries.

3.1 Erlang

Developed in 1986 at Ericsson, Erlang is a functional programming language and runtime system designed to support distributed, fault-tolerant, soft-real-time, non-stop applications such as telecommunications switches. At a low level, the language supports concurrency

via the Actor model. Processes and asynchronous message passing are primitives in Erlang. These features make Erlang an ideal language for construction of a Web-scale crawling system.

Unfortunately, Erlang has a number of shortcomings. For example, Erlang code may be difficult for a procedural or imperative programmer to understand because of its shared-nothing semantics. Therefore, to gain the power of Erlang's runtime system and process primitives while maintaining ease of extensibility, Erlang may be used as the "glue" of the web crawler and simple interfaces were designed for extensibility. C, Python, Ruby, and the JVM all have libraries for the creation of Erlang nodes.

3.2 System Layout

Erachnid consists of three primary components: the queue server, the fetcher, and the extractor. Any number of queue servers and fetchers may be started manually or via plugin.

3.3 Queue Server

Each queue server maintains a queue of pending resource request queues. Each resource request queue corresponds to a single domain name and there exists only a single resource request queue anywhere in the system for a given domain. Additionally, these pending request queues are tagged with timestamps to provide request throttling. As fetchers request resources to retrieve, values are popped off of request queues in least recently retrieved order. If a request queue for a domain empties, the queue server will spawn extractors to process fetched but unextracted resources from the database. If all resource request queues have been accessed more recently than the throttle timeout, new and updated queues are merged in from a dictionary that collects resource requests pushed from extractors.

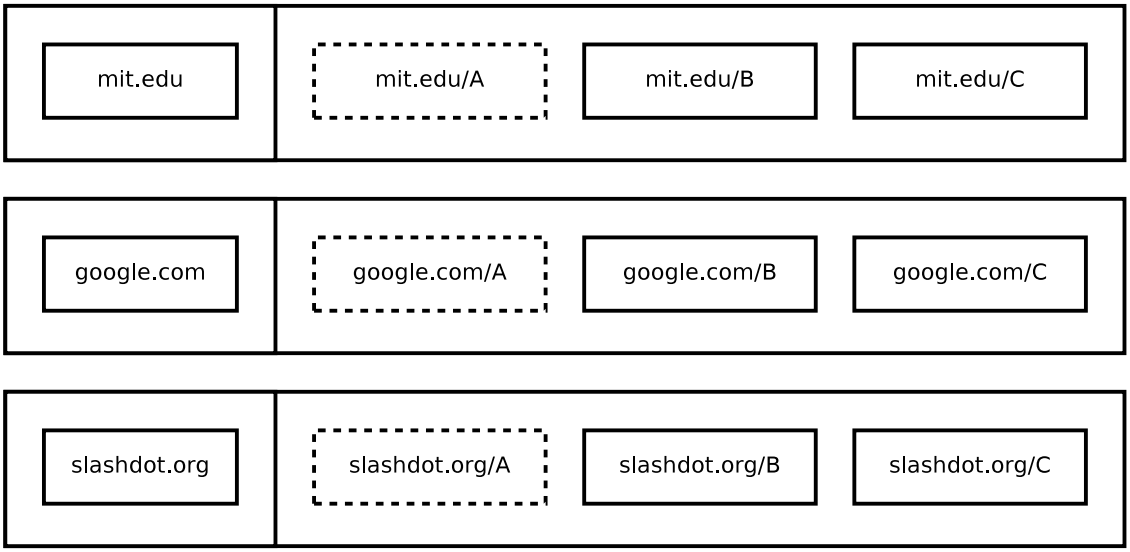

Figure 1: A simplified representation of the queue server's data structure. Here, the mit.edu queue is at the head of the domain queue and the head of the mit.edu queue, mit.edu/A, will be popped first when a request is made to the queue server. After mit.edu/A is popped, the mit.edu queue will move to the back of the domain queue and google.com/A will be popped next. Only the dashed elements may be popped simultaneously in this system due to the throttling policy.

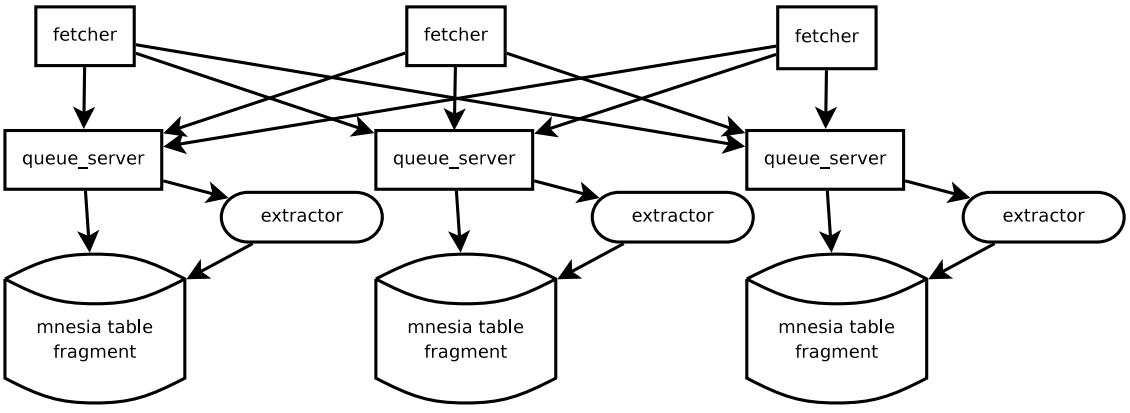

Figure 2: A 3-node crawl system with fragmented Mnesia tables.

3.4 Fetcher

The fetcher drives the crawl system. At frequent intervals, the fetcher requests new resources to fetch from a queue server. The fetcher attempts to maintain a constant number of pending network requests. Manual or automatic adjustment of this number is necessary to maximize bandwidth utilization. As network requests return, the fetcher loads the resources into the store and notifies the relevant queue servers.

3.5 Extractor

Whereas the fetch system is driven by a timer and response pump, the extraction system is subordinate to the queue server (and thus the fetchers). When a queue server begins to run out of valid response requests to dispatch, it spawns extractor processes. These processes take a list of domains that are in demand, query the database for resources belonging to those domains, and scan through the binary resource data looking for URLs. After the extractor has processed a few resources for each requested domain, it builds resource records, inserts them into the database, and notifies the relevant queue servers.

3.6 The Store

To manage crawl state, Erlang/OTP's Mnesia distributed database was employed. Mnesia supports table fragmentation, online data migration, and ACID transactions, making it very well suited for the large persistent storage system backing a web crawler.

The database schema used by the crawl system is very simple with only two tables. The 'resource' table (Table 1) describes web resources the crawler will retrieve, is currently processing, or has retrieved. A 'resource_flags' record in the resource table indicates the resource's status using a set of booleans.

Field	Type
Domain	String
URL	String
Flags	ResourceFlags
Depth	Integer
Body	Binary

Table 1: The ‘resource’ table in Mnesia

Field	Type
Domain	String
Server	GlobalID

Table 2: The ‘queue’ table in Mnesia

Because Mnesia stores arbitrary Erlang terms and does not require schema definition, the ‘resource’ table can be extended by any plugin without breaking compatibility with core systems.

The other table in the crawler’s database is the ‘queue’ table which maintains a mapping from resource hostname (domain name) to queue server (Table 2). This table is used to coordinate queue servers and fetchers so every domain has a unique queue server in charge of it.

4 Scalability

Because of Erachnid’s separation of concerns, it should be very scalable. Due to time and resource constraints, scalability tests were only performed on one, two, and three multicore machines. All machines were running 64-bit Ubuntu Linux 8.10 between 2 and 2.4 GHz. On all machines, average CPU utilization remained below 60% during testing. Each machine was connected to MIT’s network via 100 Mbps ethernet. Request speeds were measured by averaging retrieved page counts over three consecutive 10 second intervals. Undistributed, Erachnid averages 78.3 resource retrievals/sec. Distributed, Erachnid scales with $m = 0.94$

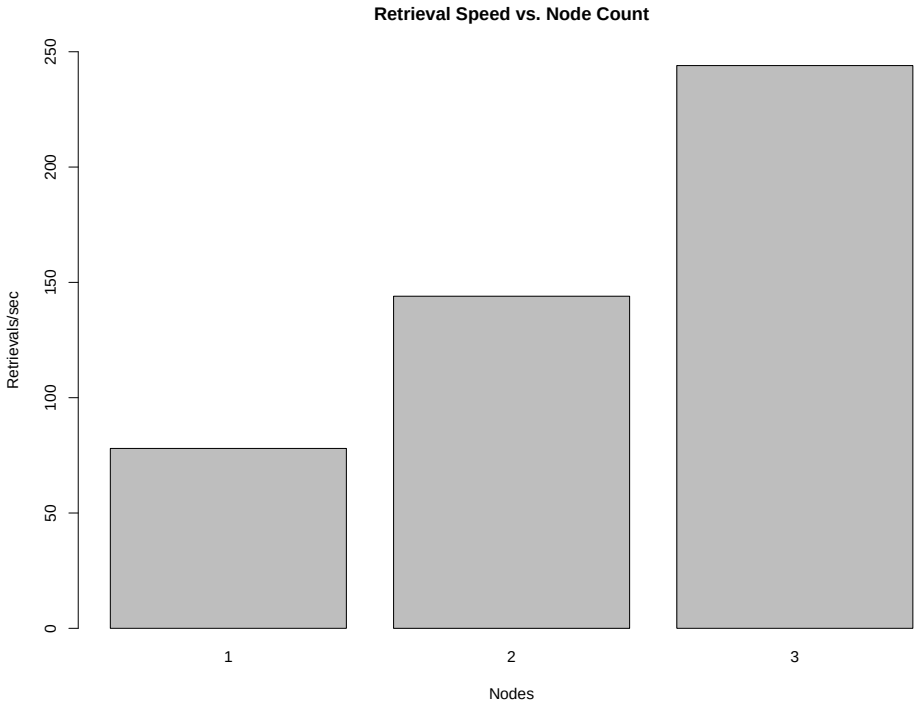

Figure 3: Adding more nodes yields a near-linear speedup (m=0.94)

on average. Crawlers were seeded with the URLs in Appendix B.

In this limited configuation, however, testing did yield near linear (m=0.94) speed-up (Fig. 3). Unfortunately, to make conclusive statements about the performance of this design, future testing of the system on more local and remote nodes is required. The author is confident, however, that similar speed-ups should be achievable within at least an order of magnitude of nodes.

5 Extensibility

Erachnid was designed to be modular and to easily allow future users to customize it. To this end, the API (Appendix A) for each crawler component was kept simple and abstract. Unfortunately, no plugin components were developed as part of this crawler design prototype.

6 Limitations

Erachnid is currently prototype-quality code and contains a number of bugs. Profiling has not been done on the system so request speeds are not indicative of what would be achievable in a production scenario. Erachnid does not contain many of the features of a generic crawler such as refetch policy, sophisticated URL canonicalization, or load balancing. If Erachnid were to be developed further, these features would be implemented as plugins on top of Erachnid's basic design.

7 Conclusion

Erlang's shared-nothing concurrency and lightweight threads are very well suited to developing Web crawling systems. The Mnesia distributed relational database system provides a solid and adequately performant storage solution for crawl tasks. With a focused and extended development effort, a crawl system written in Erlang with a similar design could easily become the standard tool a researcher uses when he or she needs to analyze large amounts of Web data.

A API

Component	Function
Queue Server	pop(count) push(domain, [resource]) split(count)
Fetcher	fetch(resource) complete(count)
Extractor	extract([domain])

Table 3: The API for the major crawler components

B Seed URLs

http://news.google.com/
http://web.mit.edu/dsheets/www/
http://en.wikipedia.org/

Table 4: Seed URLs used for benchmark

C Distribution

A copy of the current Erachnid source code is available at:

http://web.mit.edu/dsheets/Public/erachnid.tar.gz

No warranty is made or implied for this code – use at your own risk.

To run the code, an Erlang/OTP distribution newer than R12B is required. crawler.erl contains simple set-up functions for debugging. crawler:seed can be used to start a basic, single node crawl. To run on multiple nodes, the nodes must use Erlang name services, register with each other, and share secret cookies. Information about configuring and installing fragmented Mnesia tables can be found at:

http://www.trapexit.org/Mnesia_Table_Fragmentation