# SPINO: A DISTRIBUTED ARCHITECTURE FOR MASSIVE TEXT STORAGE

José Guimarães, Paulo Trezentos
*Edif. ISCTE, Av. Forças Armadas,1600-082 Lisboa, Portugal*
*Email: Jose.Guimaraes@iscte.pt, Paulo.Trezentos@iscte.pt*

Keywords: distributed, cluster, load balancing, database, information retrieval, inverted file index

Abstract: In this paper we introduce a framework for text data storage and retrieval. As an alternative for proprietary solutions we designed a distributed architecture based on a *Linux Beowulf Cluster* and *open source* tools. In order to validate the proposed solution a prototype (*Spino (Serviço de Pesquisa INteligente ou Orientada)*) has been built. The prototype is oriented towards retrieving *USENET* articles through a Web interface. Most of the points described can be applied for scenarios not necessarely related with Internet. The framework proposed is within Databases and Information Systems Integration area and describes simultaneously the conceptual and pratical aspects of its implementation. Suggested solution is presented in three different perspectives: web serving, off-line processing and database querying.

## 1 INTRODUCTION

The need for massive data storage is not a recent problem. However, with the recently massifyed use of information systems it is becomming more relevant, not only for large corporations and government departments, but also for almost all companies managing information systems.
Massive storage itself is not a difficult problem to solve. Nowadays technology (disk arrays, backup streams, data tapes...) are making affordable to store Terabytes ($10^{12}$ bytes) or even Petabytes ($10^{15}$ bytes) of data. However, this huge increase in data storage requirements redefines the problem managing and processing very large amounts of information in run-time.
There are proprietary solutions proposed by manufacturers providing performant answers to this problem. Retrieving information from large databases can become expensive: the cost of high performance computing hardware plus the price of the managing software itself.

Databases can comprise information about clients, documents, sales registering,... the only common point is that they are large and we have to retrieve information with a short and effective time delay.
Our study has been restricted to text retrieval but could be applied to other domains of data retrieval.
Our starting point was a restrictive budget and the need to store large amounts of text. The application, we intended to provide was a *USENET*[1] web interface for browsing and searching *news* articles and their content. It is well known that some *USENET* newsgroups have an intensive daily traffic resulting in a massive storage problem. In this case the system only stored permanently a subset of the newsgroups summing up to a total of 20 GB/year[2].
To address the problem, we decided to design and build a *Linux Beowulf Cluster* architecture. *Beowulf* clusters are a set of computers connected through a network[3] and build exclusively from off-the-shelve components (Pfi98).

## 2 SPINO FRAMEWORK

We start by analyzing the requirements and later the physical architecture that will support it.

### 2.1 Requirements

This system has two major requirements: **Feeding** - since the system had to be fed by a *USENET* server 24 hours a day, seven days a week and **Searching and browsing** capabilities - responsible for giving the user what he is looking for. Basically, a set of features had to be provided as well as some non-functional requirements: system needed to be stable, fast when answering user queries and scalable with the increase of data.

### 2.2 Physical Architecture

To study this problem we developed a complete framework addressing three main perspectives:

- **Load balancing** - as a solution for exhaustive and concurrent access through a HTTP connection. It is

[1] *USENET* is a network of servers that provide access to newsgroups using the *NNTP* protocol.

[2] 1 GB for articles database plus additional auxiliary databases.

[3] We used *Fast Ethernet NIC*s and a level 2 switch.

represented by shadowed blocks in the left corner of Figure 1.

- **Databases optimization** - in order to minimize database query latency, which could result in delays compromising the work on-line. Database servers are also present in Figure 1.
- **Off-line Processing** - handling of information in batch jobs that could be run at non-peak hours. Batch processing servers are represented in Figure 1 by the block at the bottom.

Each of these perspectives could be handled independently although they all are interconnected and fed by each other with content.

The *Beowulf* cluster[4] that supports the system was assembled from the off-the-shelve components. It is constituted by a master node: a Pentium II at 450 Mhz with a Ultra-Wide SCSI disk and 512 MB of RAM. The master node is only used for providing clients with required services such as *NFS* (*Network File System*) root file system, *DHCP*, *DNS*, ...

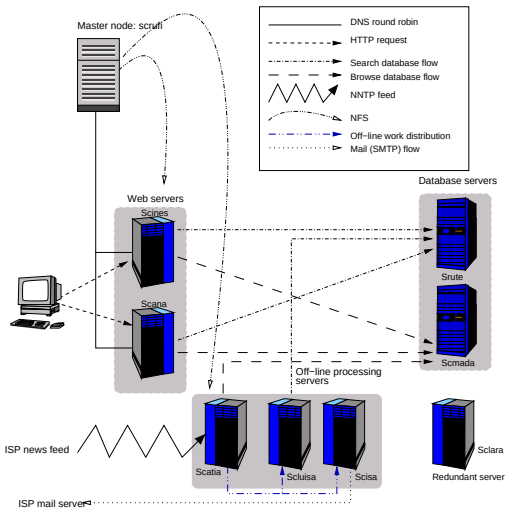


Figure 1: Logical view of client and server nodes

The eight diskless node clients all are Pentium II at 350 Mhz having 128 MB of RAM.

All nodes have *Fast Ethernet* cards and are connected to a level 2 ethernet *switch*. This architecture was structured having in mind previous experiences in this field well documented in (Kon98).

Two of the clients have installed SCSI disks and are used to store databases.

The servers for diskless machines use the following approach to boot each client: floppy disks[5] containing the necessary information for issuing a *bootp*[6] request are used; the request is catched by the master node; from the configuration information returned in the *bootp* answer, the client node retrieves the root file system from the server using *TFTP*[7].

Once the root file system has been retrieved and installed in memory, the kernel is then loaded from the file system. An alternative approach was retrieving the kernel also through *TFTP* (when using "tagged files" on server side).

## 2.3 Load Balancing

As mentioned before, one of the non-functional requirements was the system should be very fast. Should we invest only in database optimization, the bottleneck would move into the web server since all access to the system passed through it. Moreover, almost all HTTP access was accomplished by executing CGI[8] programs that were easily replicated. Thus, we decided to use a simple round-robin solution that could balance traffic between two servers.

In this section we analyze several points related with load-balancing and their effects in the overall structure.

### 2.3.1 *DNS* Round Robin

There are several different approaches for load-balancing between HTTP servers, nevertheless all can be grouped in: hardware based or software based.

Hardware solutions choose which server will handle the connection using dedicated hardware that takes decisions using *SMNP* data. Software solutions are provided from a wide range of possible choices, where we point out *Eddie* and *BIND* (*Berkeley Internet Name Domain*) round-robin.

*Eddie* is an open-source project[9] sponsored by Ericsson delivering load-balancing and automatic discovery of failed servers and subsequent re-direction of traffic. Maybe due to the fact that it is written in the functional programming language Erlang it showed to be quite slow in our stress tests with WebBench tool. When submited to highly intensive traffic, it's performance gradually degradated.

The previous handicaps of Eddie's solution lead us to *BIND* round robin: the most simple solution from software set. *BIND* is an implementation of the *Domain Name System* (*DNS*) protocol that in the newer versions can be configured to rotate IP addresses returned to a *DNS* request chosen from a set previously defined (Liu98).

Our *DNS* round robin consists in configuring the *DNS*

[4] Our cluster was named *Vitara* and more extensive information on its performance and assembling can be found at http://vitara.adetti.iscte.pt.

[5] We could have used EPROMs instead, that would result in a more straightforward process.

[6] Bootstrap protocol (also know as *bootp*) is defined in RFC 951 and RFC 1532.

[7] Trivial File Transfer Protocol (*TFTP*) is defined by RFC 783 and uses *UDP* as its transport protocol.

[8] CGI *(*Common Gateway Interface) are programs executed by the web server and whose output is present to the browser.

[9] You can visit it at: `http://eddie.sourceforge.net`.

server with two IP addresses for the same canonical name. In this case, *http://www.spino.org*. *BIND* would then rotate the returned IP address between `scines` server and `scana` server (see Figure 1).
*DNS* round robin became a good choice since the load was roughly divided between the two front-ends. There are good articles pointing out weaknesses and strengths of Round Robin but we identify clearly one: IP address was cached in browsers and proxies and that led to a situation where a client would use always the same IP address for the all session. This would also happen if the client ignored the TTL(*time-to-live*) in the IP datagram.
Some advanced but useful features are also not present with Round Robin: it does not have the ability to differentiate by port, has no awareness if servers are off-line and cannot take into account existing workload parameters using SMNP.

#### 2.3.2 Replicated static content

Since HTTP requests are balanced between two servers, we should have a method to replicate information in both servers.
In our prototype all the information (articles, newsgroup list, ...) is dynamic. Dynamic in the sense that it is stored in databases and is not archived in static HTML pages. Therefore we only need to care with CGI programs update.
*NFS* (*Network File-System*) was then used as the medium where the files were kept. As said before, our clients are diskless and they mount the root file-system through *NFS*.

#### 2.3.3 Online processing: browsing Vs querying

Two different types of client requests were identified: one related with queries by term and other with browsing throughout the newsgroups.
The first group takes a longer time, since it has to search the entire database for a boolean expression. Browsing was a less consuming operation. Both types of client requests were answered by CGI programs written in C language.

### 2.4 Databases

Databases are critical for information retrieval applications. We decided to use an open-source RDBMS (*Relational Data-Base Management System*) that would provide a good performance for simultaneous large databases and concurrent access. Postgres[10] fits in this category.
Postgres has advanced features like triggers, foreign keys, rules, subselects and views. At the same time its performance is very competitive when comparing with commercial options.
We also tested *MySQL*[11] but it does not scale very well when under a more intensive load.

[10] `http://www.postgresql.org`
[11] `http://www.mysql.com/`

#### 2.4.1 Database and Tables Structure

The design of the databases was done carefully. We soon discover that having two types of requests (browsing and querying) it would be also better to separate databases in two different servers: one would be allocated with browsing information and the other with querying information. With this architecture, complex boolean queries having inherent delays do not have any impact in browsing information databases.

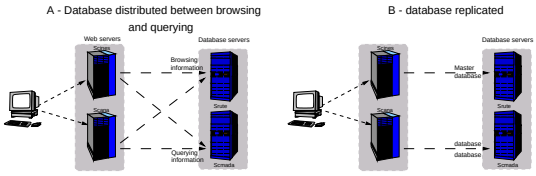


Figure 2: Logical view of client and server nodes

This scenario (A in figure 2) was compared with an hypothetical one that has same data replicated over the two database servers (scenario B in figure 2). These two databases have different roles. The client database would only be used for retrieve information (SELECT statement) where updating (INSERT and UPDATE) should always take effect on the master database for integrity assuring purposes.
This B scenario could be explored but also had some disadvantages, since it depends on the good load balancing of the web servers and the client database are not permanently updated.
For this effect, it is possible to replicate a Postgres database using Perl / Shell scripts but the method is dangerous because the sincronization is done outside RDBMS.
Users database is kept together with browsing database. Presently we have 4000 registered users in the system so the database does not have performance problems.
Database indexes were used in order to accelerate data retrieve. We choose hash indexes when wildcards were not used in SQL statements. For instance, when choosing a specific article using num (id) as reference. All other indexes are btree (binary trees).

#### 2.4.2 Optimizing access (inverted file index)

Some mechanisms were implemented in order to optimize database access.
As we can see further, in section 2.5.2, only keywords are stored in the database where we will search. Our first approach was storing keywords in an extra field of the table “articles”, together with the subject, name of the author,...
After 350.000 articles inserted, querying this database was extremely slow. This was due to two main reasons: keywords of all articles records had to be searched and we had to use wildcards in the search because keywords of articles were all together in one field separated by a “—” delimiter.

The solution for this problem happened to be the use of an *inverted file* index. Inverted files make association between a word (vocabulary) and references for the documents where that word occur (occurrences). That way, when searching for a specific term such as "football" vocabulary is searched and we directly know in which documents it is present. This turned out to be extremely efficient because we could use an hash index in the database since wildcards were not already need it. In the field "occur" we also have to include the article ID and the newsgroup due to the reasons that we mentioned before. This field of the database can have a large size in case of frequent words. Since Postgres[12] only support records (tuples) with a maximum 8KB size this resulted in a problem. Postgres development team foresee that version 7.1.0 will not have a limit for tuples size but we couldn't wait for that. Through some hacking to Postgres source code we successfully re-compile it with support for 32KB tuple size. This was enough for us to carry on using Postgres.
It is curious that when we tried to implement inverted file index, it became very hard to build a new inverted file with the first algorithm approach. The approach was: read the keywords of all articles and check if they were in the inverted file index. If they were present they would be update with the new occurrence. If not, we would insert a new vocabulary entry. At that time, 350.000 articles were already stored and after seven days of trying to generate the inverted file in a Pentium II at 350 Mhz we gave up.
Was then time to improve the code source. Our first guess (with the help of `strace` command and `gdb`) was that the database query for existing vocabulary was introducing a significant delay. And our guess was correct. When we read all the dictionary (vocabulary plus occurrences) into memory and work it from there, we only spent 17 hours building the new inverted file. Of course in the end of the program we flushed the dictionary into the database.

The process was monitored and the graph present in figure 3 generated from log files. Dotted line represents the documents retrieved from the database which are used to extract vocabulary that is later inserted in dictionary database ("invertree" table).
X-axis represents time and Y-axis is the number of documents reviewed (that have an equivalence with the total of new words inserted in the database).
It is interesting to note that until 10.000 seconds[13] (equivalent to about 100.000 documents) the number of words in the database increased faster than the number of the documents retrieved. This means that in average, each document contributes with more than a new word. For a large number of words (100.000 words) the growth of the words line is not so high as the documents reviewed.
Since the documents retrieved have a linear growth we can infer that the growth of the dictionary database does not interfere with the making of the invert file process. This would was not happenning in the first approach, where for each document the entire vocabulary database had to be searched.
The total size of the vocabulary happened to be much smaller than the database itself. As we can see in Table 1, vocabulary database is 25% of the plain approach database where each document has its keywords associated.
Some benchmarks related with average query times were implemented with the use of three different kind of queries. Note that query times presented are somehow slow -even for inverted files index- because the database server is a Pentium II at 350 Mhz and programs for accessing database are not optimized.

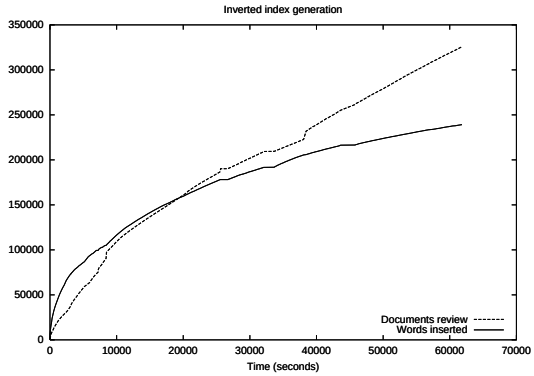


Figure 3: Evolution of making of the inverted file process

Table 1: Invert file implementation results

| | Plain approach | Invert file index |
|---|---|---|
| Total size | 144 MB | 31 MB |
| Simple query time | 13.41s | 1.14s |
| Boolean query time | 16.41s | 3.94s |
| Web query time | 21.2s | 4.35s |

## 2.5 Off-line Processing

As we described in the previous sections, some tasks are performed off-line.
The **Statistical processing** is one of them. It consists on eliminate stopwords and choose the more representative words. Words are chosen based on their frequency. All keywords are associated with a value (tf-idf based) that could be used for ranking responses to queries containning a certain word.
Another task is the **Thread processing**. It is needed for identify "parent" and "children" of the article and insert in the database. **Sending articles** is also done off-line, since articles that are written using Spino's web inter-

[12] The last version released at the time this article is been written is Postgres 7.0.3.

[13] 10.000 seconds are 2 hours and 56 minutes.

face must be sent to other news servers. This is done quite often (every two minutes). The other tasks that are not done in run time are: **House keeping** - some articles are truncated after fifteen days [14], **Notify users** - users can request that a notify is issued when a specific article arrives and **RDF update** - Spino also provides RDFs[15] of the newsgroups.

### 2.5.1 Article arriving process

Figure 4 represents the off-line processing block of Figure 1. When the article arrives to the system three of those tasks are triggered: the thread processing, RDF update and notify users.

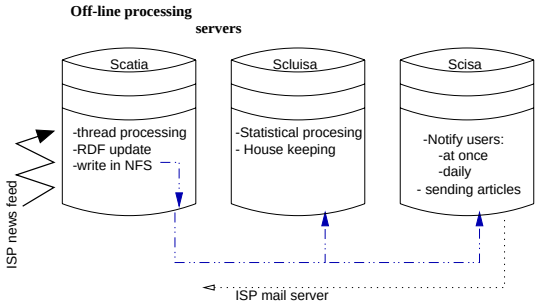


Figure 4: Off-line servers and their tasks

Thread processing was described before in section 2.4.2 and will result in an update of the database. At the same time, the RDF file for that newsgroup is update in the *NFS* directory as well as a reference that the article arrive. These actions take place in `scatia` server.
Once `scisa` detects that the article as arrived, from the reference passed by `scatia`, it starts to look in the users database to know if any user establish a request for notification. Users can request to be notified if an article of a thread arrives or if that article was written by a pre-defined author.
Notification requests can be issued once the article arrives or in a diary digest. This last operation is done by a different program that is also executed in `scisa` but at 3:00am when traffic is low.

### 2.5.2 Statistical processing

The body of the article is stored only for display purposes. Querying for some term in database is done over article keywords. Keywords are the most frequent words in an article. The distribution of different words inside each document is the subject of study in some models like *Zipf's Law*(Zip49) but are out of the scope of this document.
Stopwords[16] are eliminated when calculating those keywords. This task is very CPU consumer. We used cron *Vixie cron* daemon to execute this task and all other events that are not triggered by article arrival.
We mentioned before that inverted files were used to index terms.
This task is done right after the keywords are calculated.

[14]The article itself (first 8K) is kept in the database and the remainning information is removed.

[15]Resource Description Framework are metadata that can be interchanged between servers and that are written in XML. In this case is used to provide headline of each newsgroup to other Internet sites.

[16]Since stopwords are terms like articles and pronouns that does not carry meaning in natural language is very difficult to eliminate them in foreign languages. Presently our prototype only eliminates portuguese pronouns.

## 3 CONCLUSIONS

Our prototype is now running and has a representative base of users. It is relevant to note that in Portugal it is the only client proposing a web interface for *USENET* news.
This fact is not a guarantee that the framework we propose is the best appraoch, or even useful, but we can name some advantages and, naturally, disadvantages.
The main advantage is stability. One of our database servers has presently an uptime of 76 days. The all system now works without human intervention and it very seldom need a reboot.
Another positive aspect is price/performance ratio. With a cluster of nine off-the-shelve machines we get a performance competitive with expensive proprietary solutions of medium size. We used *PVM* (*Parallel Virtual Machine*)[17] to perform some tests that were very stimulating.
Open source tools have proven to be very stable and powerful. Postgres, *BIND*, crond and Linux *NFS* implementation as well as other GNU tools are probably not the most user friendly tools available, but are very well engineered towards performance and bug free.
The main aspect we can consider as a disadvantage concerns the know-how that is required in order to implement a solution using these tools. Expertise resources are fundamental to tune the system.
From all that has been said, we can draw the conclusion that a *Beowulf* cluster system could provide the most adequate solution to the problem, and the use of open source tools aid to build a relatively inexpensive distributed architecture for efficient large scale text storage and retrieve.
Some work has yet to be done, namely a more thorough performance analysis on aspects like: diskless clients, *NFS* overhead and Postgres SQL optimization.

## ACKNOWLEDGMENTS

The authors would like to thank ADETTI (Associação para o Desenvolvimento das Telecomunicações e Técnicas de Informática) by their support to this work, and the means that were put at our disposal.

## REFERENCES

Manu Konchady. Parallel computing using linux. *Linux Journal*, 45:78–81, January 1998.

Paul Albitz Cricket Liu. *DNS and BIND*. O'Reilly, third edition, 1998.

Gregory F Pfister. *In Search of Clusters, the ongoing battle in lowly parallel computing*. Prentice Hall, second edition, 1998.

G. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.

[17]`http://www.epm.ornl.gov/pvm/`