STUDIA UNIV. BABEŞ-BOLYAI, INFORMATICA, Volume XLIII, Number 2, 1998

MULTI-TIERED CLIENT-SERVER TECHNIQUES FOR DISTRIBUTED DATABASE SYSTEMS

DARABANT SERGIU ADRIAN

Abstract. Information explosion across all areas has determined an increase in hardware requirements for application that provide data to the users. As hardware evolvement is quite susceptible to be bound after a top barrier is reached, new technologies must be developed in the software area in order to keep up with the requirements. We present here such a technique for improving access to data by means of distribution and by using client-server multi-tiered techniques. The main idea is to, transparently and efficiently, distribute data in multiple places using a client-server multi-tiered system. We model the system by introducing a broker between clients and servers in a client-server system. This additional level in the communication layer between clients and server will handle things as data distribution and will participate in the query processing stage.

1. Introduction

Large databases are, lately, more and more a quite common thing. High amounts of user data collections are needed in almost all today's applications. In order to keep up with the performance needs and with this high amount of data, new technologies must be used. Hardware performance has increased a lot in the last years but so did the requirements in the software area (amount of data, speed requirements). As software needs evolve almost in parallel with hardware performance we will reach, someday, a top performance that will hardly be improved after that, in other words a top limit in hardware performance. This is happening as we tend to reach the light speed. To solve this problem research in software area is done and is aimed to find new software solutions. We deal in this paper with a very simple, yet highly effective, method to improve performance in a distributed system. We will apply this model to a distributed database technology although it can be used with some modifications in any distributed system.

A distributed system is a collection of sites connected in a network together by some kind of communication lines that runs an application on the entire system and makes transparent to the user the fact that we are dealing with more than one computer. A distributed system can be easily used to reduce query time response, if properly used, in a database system that is highly overloaded when used in a traditional way.

In a traditional client-server model, a client, or more clients, usually communicate with a server that provides services. There is no link between the way one server processes a request from a client related to the others similar servers in the system. Most of the time the servers are unaware one of each other or they are aware just by means of mutual exclusion when accessing common resources. In a such multi-server or single application server system we can improve performance by using an additional intelligent broker that assures a global state between servers or replicates data from one server to multiple servers and manages the client-to-server

requests/responds.

One example of such system is the Internet Search Engines. Such an engine is composed by an Information Retrieval (IR) system that manages text collections indexed from the Internet or local text and clustered document collections. Dynamic IR systems, where data and documents are collected from the Internet continuously are highly eligible for the distributed technique we present in this paper. In the same time, the system must quickly respond to a large number of user queries. A system like this might be an Internet search engine like *Altavista*[1] or *Lycos*[2], etc. They provide a very easy way to search for information on the Internet using a friendly interface. There is a specific need for returning, as a result, from a query, the latest information available to the user because, statistically, this will be the most relevant to the end user. This problem did not exist in the past, when existing information was refreshed once or a few times a day. In these days, however, information is gathered form the network at speeds as high as hundreds of MB/hour. In this case, most of the time, new information is added dynamically to the system in large quantities, in parallel with processing queries from the end users.

2. General System architecture

The difference between the architecture proposed here and the classic client-server architecture is highlighted in the *fig. 2.1.*

As we can see in the traditional architecture each client must connect to every server, and more it has to know all the available servers and services. In the second model, the broker is used to provide links to all the servers in domain. The connection between clients and servers is managed by the broker. The architecture presented in *figure 2.1 b)* has the following components:

1. **Clients** - a user interface for accessing services. Usually is a process specific to the system or a general client for multiple services. In a database distributed system it can be an interface to the services provided by the database: update, query, data definition, etc. It's not very complicated but rather simple and has to know just how to connect with the broker. The rest is application specific and is implemented in the application: sending commands to the database engine. A client does not connect directly to a specific database server.
2. **Connection Server** – is the broker between the clients and servers. Handles requests from clients by forwarding them to the appropriate servers and merging back the results if necessary. Since this is the key for the whole new system architecture, it has some properties that help to improve performance. The connection server is usually a lightweight process that manages all the messages between clients and servers i.e. client interfaces and database engines. These messages are placed in queues that are served using some priorities or simple in a *first-in-first-served* order.
3. **Servers** – Processes or just front-ends that implement a standard communication interface between the connection server and database engines. In many cases, the single database engines don't have interfaces to access them remotely.

[1] Altavista – http://www.altavista .digital.com
[2] Lycos – http://www.lycos.com

Consequently we need a way to communicate with the database either locally (on the same host), or remotely as needed. Again these interfaces are quite simple and they only need to implement a standard communication protocol, and post the commands from the connection server to the database engine in an understandable language (specific to the attached database engine). This is another way to improve qualitative performance. We can build a heterogeneous system made up from multiple kinds of database engines, provided that we can come with a proper interface to that requested by the connection server; i.e. database engines must support some minimal facilities. (ex. SQL support, transactions, etc).

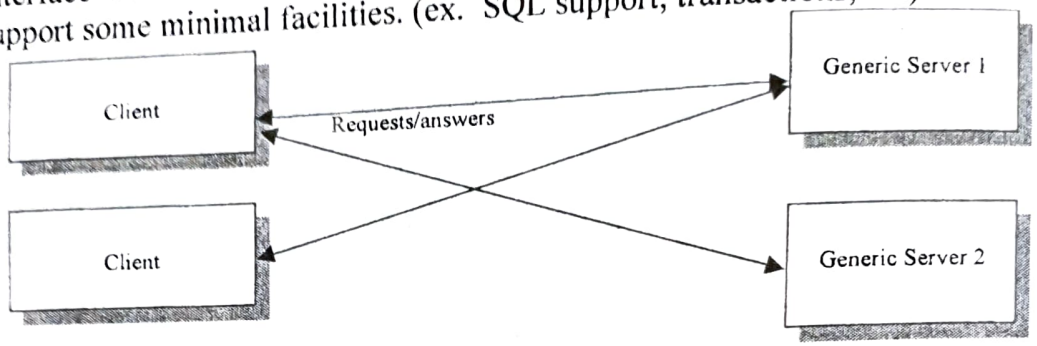

Fig.1 a) Traditional

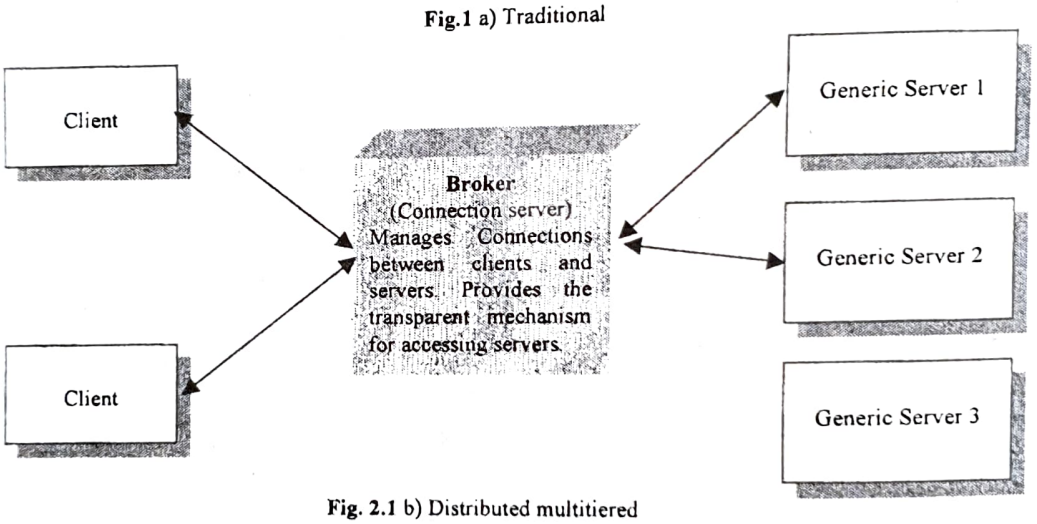

Fig. 2.1 b) Distributed multitiered

1. System simulation

In order to demonstrate that we can achieve a performance boost by using a system like this we will simulate such architecture for a distributed Information Retrieval system. We will use a simulation rather than a specific implementation because implementing such a system can be time costing. Such a simulation can model the real behavior of the system under different circumstances and enables different aspects of a distributed system to be tested prior to implementation. We use different parameters in the simulation to model different number of users accessing the system, different queries and number of term per query, different topologies and *on load adaptation* techniques.

In the simulation, we used some results of a similar model developed by Brandon Cahoon and Kathryn S. McKinley **[1]**. Our work relates, mainly, to their measurements of a distributed system architecture for Information Retrieval systems. They proposed a distributed Information Retrieval were clients connect to

document retrieval engines using a single or a fixed number of brokers. Each architecture was measured in terms of query time response, document retrieval time, network latency, client-processing time. In their experiments, they used a static topology under different workload coefficients and using different document collections. They varied some parameters of the topology, like number of connection servers and/or Retrieval engines and studied the results in order to find out the bottlenecks of the system on different topologies.

As a simulation environment, we used YACSIM **[5]**, a simulation library. YACSIM is a discrete–event, process oriented simulation language based on the C programming language. We model, using YACSIM, all the major components of the system including clients, connection servers and retrieval engines. We used as a retrieval and query engine Inquery, a probabilistic retrieval model that is based upon a Bayesian inference network. Inquery accepts natural language or structured queries. For query operations, the system outputs a list of documents ranked by relevance. Internally, the system stores the text collections as an inverted file. We used Inquery in order to keep results obtained by their study related to our work and keep their relevance intact. We will use in our experiment different text collections of those used in the above-mentioned work. This can influence some of the system measurements. There is a strong relation between query evaluation time and the number of terms in the query and the frequency of each of the terms. In the TIPSTER 1, a collection of full-articles and abstracts used by Cahoon, the correlation between query length and query evaluation time is .96 and for query term frequency is 0.95. The time to evaluate a single term ranges from 0.5 seconds for a term that appears only once to 17 seconds for a term that appears 554,658 (maximum term frequency in TIPSETR 1). Our collection has close parameters to those mentioned above. They are 0.90 for correlation between query time and query length, respectively 0.85 for query term frequency. We divide the query response time into CPU and disk access time. The simulator computes the evaluation of the query response time by adding the evaluation times of the individual terms in query like in previous work we relating to.

Like in their study, we approximate constant the document retrieval time for an Inquery server and we take an average retrieval time from a number various set of retrieval operation on documents with different sizes. We repeated for a fixed number of times the set of retrieval operations with different randomly selected documents. The average response time is used in the simulation. Connection server times are different from that presented in **[1]** because we try to improve performance of system by making it self-adaptable. We do this by using the results gathered from studies made in **[1]**, **[2]**, **[3]** to develop a simulation for a self adapting system to a different workload pattern and using a different number of servers and services, having information replicated at different sites. The system also supports fragmentation. Our results might a bit different than those presented in **[1]** because of the different sources of data used, but we found it in a ±19% of the real implemented system. This is because we used some of the results presented in **[1]** that do not accurately model all out results and because some of the caching and data distribution techniques we use alter the in a various manner the response time of the system.

4.1 Simulation model

The simulation for our adaptive model is working, as we said, using some results from previous work in domain. In addition, it uses some other parameters in order to demonstrate de correctness of our assumptions about system workload. These assumptions are:

1. The system is transparently distributed.
2. We have stable links between all the components in the system. If some links fail, the results would be inaccurate. In the same time, the algorithms, used here for communication between brokers, for synchronization and for exchanging hints and cached data do not work correctly in the case of broken links. The system might go in an unstable status in the case of communication failure. Anyway, our aim was to demonstrate that we could improve performance by adapting to a particular system load in time. There are various studies about recovery from communication errors in distributed systems that can be applied in order to improve the system.
3. Information is spread across a network of computers on servers, using different topologies, replication and fragmentation techniques **[4], [6], [7]**. The replication and fragmentation strategies are dictated by the possible number of clients, amount of data in a text collection, amount of update operations, etc.
4. We can have one or more distributed text collections. They are spread out on the network, replicated and fragmented. In the case of small text collections, between the others we have, we can choose not to fragment it. We can still replicate it if it's a very intensive searched database. Otherwise, this can cause major bottlenecks in the system.
5. We have multiple brokers in the system. We say the system has a dynamic topology, as we have a minimum number of brokers registered at a particular moment. This is the starting configuration when the load coefficient is zero. They are aware by each other and they communicate with each other, as we will see. They serve clients until workload in the system reaches a peak point. At this moment a new broker is spawned in the system and registered, i.e. made known to all the others. This always happens on a different free (of brokers) site.
6. Each broker knows the server topology and data distribution and replication schemes, i.e. it has information about fragments and information present in each node. This information is obtained from the servers when they are started. When a new broker is started, it receives from the initiator all the information about the current system topology and data distribution.
7. Each broker analyzes each query received from a client and splits the query in such a manner that only sites that contain information about those terms in query are interrogated.
8. A user can choose to interrogate just one or multiple collections. If multiple collections are interrogated it is the broker's task to decide which data servers to query. This information is extracted from the data it has about system topology and the geographically collection distribution.
9. Each client has a local cache where documents retrieved are stored.
10. Each connection server (broker) has it's own cache used together with the other broker's caches to improve speed when common information is asked by multiple clients, or repeatedly, by the same clients.

4.2 Simulation Parameters

We used some parameters to vary the conditions in which the simulation runs. Some of them are taken from studies made in **[1]** and others are new necessary because of the improvements of the system behavior and open architecture:

- *Number of clients/servers (C/IS)* – used to model small and large configurations. It is different used comparing to work in **[1]** as we need to constantly vary the number of clients to make the system adapt itself to the specific workload at some particular moment in time. We do not use this parameter to identify bottlenecks in the system but, merely, to simulate the necessary conditions (especially by varying the number of clients) for the system to adapt to an increase in the system workload.
- *Terms per Query(TPQ)* - is also used to increase or decrease the workload in the system
- *Distribution of Terms in Queries (QTF)* – we used a local observer distribution of terms in queries, calculated for our text collection.
- *Number of Documents that Match Query (AR)* – is the list of documents returned to the clients. This parameter affects the network traffic and processing time in the brokers.
- *Client Cache Hit (CCH)* - is the client cache hit percent/ coefficient
- *Broker Cache Hit (BCH)* – a value for each broker cache hit frequency. This is used in order to use cached data when this is possible between brokers. They exchange information when a new query is issued so that if information already exists, it is given to the appropriate broker and after that to the appropriate client.
- *Max. Broker Queue Length (MaxBQL)* – is the maximum queue length for a broker. When this value is reached that broker is going multithreaded until another peak value is reached and after that, the broker tries to spawn another broker on a foreign site in the system. This is useful for moments when the system is not very loaded and we have the facility to run multithreaded brokers. We choose another measured value for going to spawn remote processes when reached. If chosen in a proper way, this can lead to an improved level of network traffic, because of load balancing as one machine does not get overloaded with too many requests that it cannot handle.
- *Max. Broker Childs (MaxBC)* – this is the other value, we, previously mentioned. It is used as a maximum load value for a site. When reached, a site tries to spawn an additional remote broker on a free site.
- *Maximum Number of Brokers (MaxNB)* – this is the maximum number of brokers permit in the system. When reached no other brokers will be spawned remotely, even if one of the existing ones is overloaded. This is useful because we do not have an infinite system as number of sites. If proper chosen it will not be reached sooner than the moment when all the existing data servers are properly loaded. After that no other increase in performance can be achieved, whatsoever.
- *Minimum Number of Brokers (MinNB)* – is the minimum number of existing brokers in the system. Even if they are idle, they don't stop execution. Properly chosen can handle sudden, large, workload increase after an idle period.
- *Maximum Idle Time (MIT)* – is the amount of time that, a remotely spawned broker, lives idle. When this value is reached that broker stops execution and dies.

- *Maximum Server Queue Length (MSQL)* – is the maximum number of entries in a server's waiting queue. When this number is reached, brokers that try to use it receive back specific hints. If another server with the same data exists in the system, all the brokers will choose that one. If not, they will continue to use this one which will, soon, become overloaded.
- *Maximum Pre-analyzed Queue Length (MPQL)* – the length of the pre-analyzed queue at each broker. When receiving queries each broker tries to pre-analyze a number of queries in the queue an exchange information with the other brokers. This is useful for caching purposes, as we will see. If the requested data is already present somewhere in a broker's cache it will be returned to the appropriate handling broker and to the client without having to send all the query (eventually split up by fragment purposes) to servers.

4.3 System architecture

Our system architecture differs a bit from the original one presented at the beginning of this paper. This is, mainly, because we permit a dynamic topology of the system. Brokers can dynamically be spawned or closed (automatically, as we saw). In the *fig. 4.3.1,* we present the actual architecture of the system we simulate.
The system is following the next algorithm:

```
Initialize algorithm is:
Initialize system
        For each initial broker b do Run_Broker(b,true);
End;
Procedure Run_Broker(broker runb, boolean initial) is:
register with all remote brokers;
if not initial then
        Gather information received from initiator;
else
        gather information from the system ;
end if
while true do
        while Incomming_connection()=false do
                if not initial then
                    compute_idle_time;
                    if Max_Idle_Time >= MIT then
                            unregister with all remote brokers;
                            exit;
                    end if
                end if
        end while
```

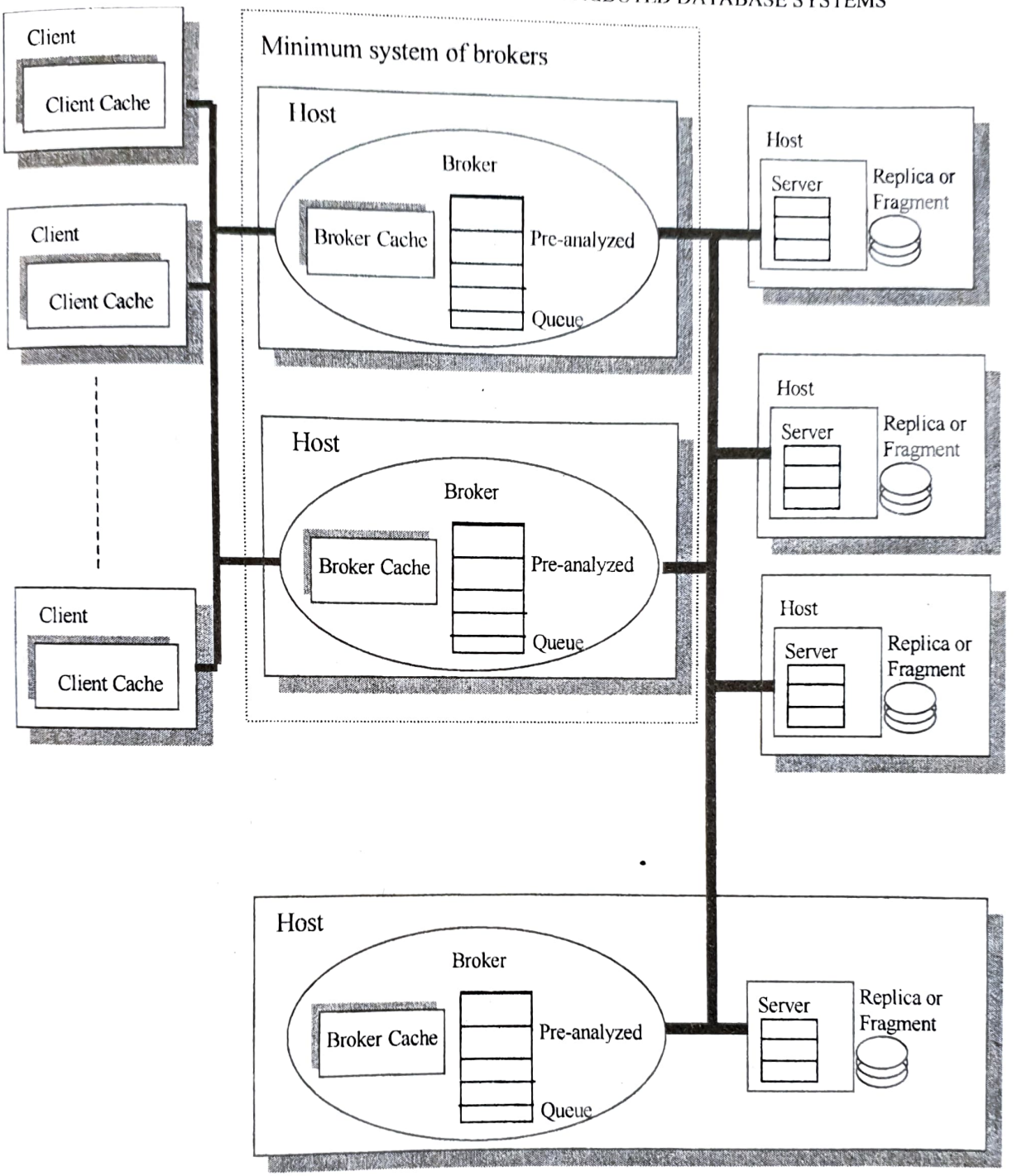

Fig. 4.3.1 System architecture

```
insert into queue request
if Queue_length >= MaxBQL then
        exchange hints with all existing remote brokers;
        if exists idle broker b1 then
                pass hint to client to connect to idle broker b1
                remove from queue;
        else
                if Child_Number < MaxBC then
```

```
                        spawn local child blnew;
                        Run_local(blnew);
                        pass hints to client to connect to blnew;
                        remove from queue
                    else
                        if Max_Number_of_Brokers >= MaxNB then
                            if exists partially free broker bfree then
                                pass hints to client to connect to
bfree;
                                remove from queue;
                            else
                                postpone request;
                            end if
                        else
                            spawn remote broker bnew;
                            Run_Broker(bnew, false);
                            pass hints to client to connect to bnew;
                            remove from queue;
                        end if;
                    end if;
                end if;
            else
                extract request req from queue;
                analyze request;
                pass request hints to all remote brokers;
                pre-analyze requests from queue(MPQL);
                while available_remote_data(remote_broker) do
                    retrieve_data(remote_broker);
                if full_processed(req) then send result to client;
                else
                    identify data servers;
                    take load hint form server;
                    if Max_Server_Queue_Length>=MSQL then
                        identify another server;
                        if found snew then server=snew; end if
                    end if;
                    send partial query to server;
                    wait for results;
                    send results to the client;
                end if;
                pass hints to all remote brokers queue(MPQL);
                if available_remote_data(remote_broker) do
                    mark reordering operation on queue;
                    identify request reqcache in the queue;
                    reorder requests in the queue;
                end if
```

```
        while available_remote_data(remote_broker) do
                retrieve_data(remote_broker);
        if full_processed(reqcache) then send result to client;
        else
                identify data servers;
                take load hint form server;
                if Max_Server_Queue_Length>=MSQL then
                        identify another server;
                        if found snew then server=snew; end if
                end if;
                send partial query to server;
                wait for results;
                send results to the client;
        end if;
End procedure Run_Broker;
```

Procedure Run_local runs a local child on the same site. The only difference from a remote broker is that the local child only receives information the parent process (initial broker) and terminates as soon as serves a request to a client.

An initial broker waits for a connection and inserts the request into a queue. If the queue does not get longer than MaxBQL the broker processes the request. It first analyzes the query and sends hints to the other remote brokers. If there are brokers that have related information in the cache, they send back the data. If the request is fully processed, we return result to the client, otherwise we send the rest of the partial query for evaluation to the identified servers and after we receive the result we combine it with the cached information and send it back to the client. Meanwhile the queue is pre-analyzed and identify requests for which some brokers have cached data, we reorder the queue if possible, i.e. no other unresolved request has gone back in the queue (because of multiple reordering). If we have such requests we take cached data and send it to the client. Anytime when a broker is much more loaded than others requests are routed to the other brokers. When a broker expects to send a query for evaluation to a server, it takes hints about server load from the server. If the server is overloaded than the broker tries to identify another valid server. If found sends the query to the second server else to the initial one which will become overloaded.

5. Experiments and results

In this section, we present the results of our experiments. We tested both cases in which clients only query a single document collection distributed and fragmented over the entire network, with different number of servers. In the other case clients query multiple text collections, some of them distributed, fragmented and replicated.

5.1 Distributing a single text collection

In this section we examine the performance of the system when we divide a single, large (0.8 GB), text collection among all servers. The size of the text collection

managed by each server depends upon the number of servers.

For up to about 6-8 servers, a single connection-server can handle all requests. If we raise the number of servers from 8 to more the single broker rapidly tends to go to multithreaded mode or passes connections to the others. In this case we have a continuously improve in performance with some pauses when the system adapts itself to the increasing load. For up to 300 clients we improve performance by 12.34. The cause of this result is that each Inquery server processes the data more easily. Documents are most likely distributed over all the servers. This highly improves performance. As we increase the number of clients the load on the connection servers, increases and they tend to redirect some queries between them. If we choose the maximum number of brokers as one we have almost the same results as those presented in [1]: an increase in performance for 1 to 20 servers and up to 400 clients. This is because we use multithreading in our connection server. Experiments in [1] do not use multithreading, fact that is the cause of their results (not as good as these ones). They increase performance for 1 to 8 servers and up to 256 clients. We reach just 400 clients and 20 servers because after that the network latency increases as the connection server becomes increasingly utilized as number of clients increases. In this case, Inquery servers perform quite well. If we replicate data on additional servers, we do not increase performance. In this case, the network and the connection servers become, more and more, a bottleneck. When the number of servers increases over 40 the connection server cannot process s as much results as the servers send back.

If we increase the number of Inquery servers above 20, the connection servers gets overloaded much more faster and becomes a bottleneck. Performance degrades, for 40 servers, 4 times when increasing the number of clients from 1-400. The result is better in this case, also, than that from [1]. Performance degrading is 4 times, in their case, for just 8 servers and 256 clients. In multithreaded mode, connections servers can process as much as 16-32 requests per second. Above that, they became over utilized.

If we increase MaxNB (maximum number of brokers) the system scales much more better as this is the point were the self adapting property becomes utilized. With MaxNB=2 the system performs better with up to 64 servers. We see that the system performs better as the number of clients and Inquery servers increases. We get a speedup of 3.65 over the single connection-server model using the same number of clients. For MaxNB=4 improves even more performance for large configurations. Again, if we distribute just a single document collection we do not gain improvements for more than 16 Inquery servers if we replicate some fragments. This is because Inquery servers can respond immediately to queries. If we use large queries, we gain speedup if the number of Inquery servers is under 128. Over 128, they can respond even to large queries. Under that number, they become over utilized and replication helps, and helps a lot if each fragment is replicated. In this case, we can improve results by doubling speed. This is happening because of the hints passed between connection servers and Inquery servers before evaluating a query.

In these cases, if we use caching, especially on the connection server side we can improve performance. We observed that many times people tend to send similar queries many times, or, they change some terms in a query and resubmit it. Again, we observed that in a large group of clients with the same "preferences" in terms of query similitude, caching can improve with a factor of aprox. 2. In the later case, this is

happening because people tend to search for similar words. This fact leads to a high percent of reordering close transactions. Therefore, many queries are not sent to the servers but evaluated by the connection servers.

For MaxNB>4 and a number of Inquery servers between than 128-240 the system performs increasingly good in the same time with the increase in the clients number. This is because the system scales well at the connection server level. At this point self-adaptation facility of the system works at maximum, provided that we have enough sites to run the system. We can only observe short time bottlenecks on the periods when the system switches between different strategies. For a large number of clients (more than 600), however, the number of messages exchanged over the network increases a lot, due to the necessity for passing hints between components. At this level, caching techniques tend to be unutilized because the amount of different, not similar, queries exceeds the caching capacity of each node. The amount of messages exchanged over the network is very high and the network becomes a bottleneck source. For some configurations of 8-10 connection servers and more than 240 Inquery servers, both components, have idle periods because of the network latency. At this level of high number in Inquery servers and clients, our experiment shows that the simplified algorithm without hints works better. This result comes from the fact that the connection servers spend a lot of time exchanging messages between them before each query and with Inquery servers in order to determine server's load.

For a even high number of Inquery servers and if no replication is present, connection servers spend a lot of time to determine the location (identify servers) of data and to process hints and messages from the other brokers. However, if the number of Inquery servers is not so high, the system performs better in any situation. We also think that a number of more than 400 Inquery servers is unrealistic and will not be soon implemented in any system because of the high demand in the number of different computers to run the configuration.

5.2 Distributing multiple text collections

In this case, the system maintains multiple text collections. In this configuration, each client selects a random subset of the available collections to search for. Statistics show that a client searches half of the available collections. The workload increases both as a function of the number of clients and the number of Inquery servers. For MaxNB=1 the system performs better, in our case, due to the multithreaded connection server. For MaxNB>2 performance increases, if we use small queries, together with the number of Inquery servers. For TPQ=2 we improve performance for more than 32 Inquery servers because transaction time decreases due to parallelism. The same result is obtained by experiments in **[1]**. As we add Inquery servers they respond faster and for a MaxNB>2 the system performs quite well for 200 clients. The system begins to adapt to the workload increase for more than 128 and 180 servers. Caching techniques do not scale very well at this level.

6. Future Work

The system scales well for quite a lot of situations. However, we must assume

stabile links between computers. Otherwise, the algorithms may loop forever. We did not anticipate the case where a component in the system cannot respond to a message. This happens when the system variables are updated due to system self-adapting capability. There are a lot of techniques and distributed algorithms that take in account these situations. Another issue is in the caching techniques used here. We think that they can be improved in order to give better results even in the situations presented, were they seemed inopportune.

7. Conclusion

In this paper, we present a model of a distributed system that tries to alleviate some of the problems encountered in the traditional IR system. In addition, the system is simulated in a IR model. We think that the same solution or, a similar one, can be applied in any other traditional system that do not scale well when overloaded. Databases scale very well to such a technique, even traditional non-distributed databases, or non-relational databases. We develop a flexible simulation based on results previously obtained on similar research and use those results in order to make a distributed system adapt to the particular workload of a moment. Our results show that the system we proposed provides scalable performance on almost all kind of architectures. Replication, caching and fragmentation are ones of the methods rarely used in IR, and many times even in traditional systems. Together with data distribution on a system that adapts itself to the specific workload, they can improve speed in gathering information.

REFERENCES

[1] Brandon Cahoon, Kathryn S. McKinley, Performance evaluation of a Distributed Architecture for Information Retrieval, SIGIR 1996 conference paper.
[2] Forbes J. Burkowski, Retrieval performance of a distributed text database utilizing a parallel process document server. In 1990 International Symposium on Databases in Parallel and Distributed Systems, pages 71-79, Trinity College, Dublin, Ireland, July 1990.
[3] Z. Lin and z. Zhou, Parallelizing I/O intensive applications for a workstation cluster, a case study. Computer Architecture News, 21(5):15-22. December 1993.
[4] Antony Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared nothing distributed text document information retrieval systems. Technical report STAN-CS-92-1456, Stanford Univ.
[5] J. Robert Jump. YACSIM Reference Manual. Rice University, 1993
[6] Charles L. Viles and James C. French. Dissemination of collection wide information in a distributed information retrieval system. In Proceedings of the 18th International Conference on Research and Development in IR, Seattle, WA, 1995
[7] Ian A. Macleof, T. Patrick Martin, Brent Nordin, and John R. Philips, Strategies for building distributed IR systems, Information Processing & Management, 23 (6):511-528, 1987

Babeş-Bolyai University, Faculty of Mathematics and Informatics, RO 3400 Cluj-Napoca, str. M. Kogalniceanu 1, România.

E-mail address: dadi@cs.ubbcluj.ro