Distributing Web-based Content Management System - "FERweb"

Ivan Voras, Kristijan Zimmer, Mario Žagar
Faculty of Electrical Engineering & Computing, University of Zagreb,
Unska 3, 10000 Zagreb, Croatia
ivan.voras@fer.hr, kristijan.zimmer@fer.hr, mario.zagar@fer.hr

Abstract:. *This work explores various ways of distributing the FERweb CMS system (a web-based Content Management System of the University of Zagreb, Faculty of Electrical engineering and computing)*

Primary aim of this project is enhancing the performance of the system, within the constraint that the efforts must be based on existing technologies used in the project and with minimal impact on the existing code.

Keywords: web content management system, distributed web applications, open source, PHP, Apache, PostgreSQL

1. Introduction

The "FERweb system" is a colloquial name for the custom-made web CMS (Content Management System), developed at the University of Zagreb, Faculty of Electrical engineering and computing. The system [9] is the result of over three years of development and it has successfully been deployed as the Faculty's web site (where it's used for collaboration and coordination between students and the staff), the Croatian Academic and Research Network (CARNet) public web site and was sold to Pliva d.d. (a leading pharmaceutical company) for extended developing and customization as a part of Knowledge management system.

The system is implemented using Apache 1.3 web server, PHP 4.3 programming language and PostgreSQL 7.3 database server. Its main characteristic is the concept of "portlets" - modules (or "parts of a web page – portal") that behave as a distinct and independent subsystems. The modules control their HTML [5] representation using Smarty templates – so the programming logic is separated from the presentation. This modular approach allows for much freedom in adding features to the CMS system and also allows for applying various micro-optimizations that exploit the isolation between the modules.

A web application can roughly be described as consisting of a front-end, the program logic layer (also called business logic, just "logic", or "the web application"), and the database end (or back end).

The system serves completely dynamic HTML content, generated from database data via PHP code. Images on the web page are static files served from the filesystem. Database interface code is implemented as a separate layer, allowing for many optimisations in data access operations. Business logic and presentation logic are separated by the use of templating system (Smarty Templates).

In production, the system is usually deployed with PHP acceleration and object caching software (eAccelerator), but which was disabled in the test setup.

2. Why Distribute a System?

The FERweb system started as a research project done by faculty staff and undergraduate students, but has evolved in a self-supporting and viable project, and thus surpassed its original goals. In particular, increased attention had to be given to how the system performs in situations with large number of concurrent clients.

This work focuses on providing three benefits to the system:

- increased performance,
- increased system reliability,
- added new functionality.

Of these, the first two are of the greatest importance when deploying to a corporate environment with high expectations for concurrent access and where availability is of critical importance.

3. On system performance quantification

For the purpose of this article, performance of a web system will be defined with two values: the number of completed transactions per second that

are served to the clients, and the number of concurrent connections to the system that can be active at the same time (e.g. are in the process of being served). Since this is a web application, performance is measured using the "siege" program [6]. This program takes a list of web addresses (URLs), and fetches each page in the list with desired concurrency level and duration of a test.

Performance measurements were conducted under the following protocol:

- The "siege" program was used to perform the requests, from a computer in the same switched Ethernet network with enough computing resources to sustain the request load
- The list of URLs to fetch was created to contain all the dynamic pages contained in the system. The static elements were present in the list only when specially noted for the test
- All of the already present system optimizations (such as the SQL cache) were active, unless otherwise noted
- The tests were made 10 times in a row, with pauses between them until all systems involved were recovered from the load and settled to "100% idle" state
- The concurrency level of the requests was varied and reported with the results
- The average transactions/sec and concurrency scores are reported

3.1. Initial performance

The reference performance measurements were taken on the following server setup:

- Intel Pentium4, 3GHz (hyperthreading disabled), 1GB RAM memory, single 10kRPM SCSI hard drive
- FreeBSD 5.2.1 operating System
- PostgreSQL 7.3.4 RDBMS
- Apache 1.3 web server
- PHP 4.3.5 programming language as Apache module

The software setup was in the usual mode of deployment, with all the components running on the same system. The purpose of this benchmark is to introduce the performance capabilities of the system and serve as comparison with later benchmark. This benchmark was conducted with the URL list including static graphic files.

The "transactions/sec" and concurrency" data is reported by the "siege" program and represent the number of completed client requests per second and the number of simultaneous active connections to the server during the test, respectively. CPU load is (very roughly) the average number of processes simultaneously running on the server in the last minute of the test (the first of the three standard Unix "average load" numbers) and is present here as a guideline of server load only (CPU load is often not the limiting factor of server performance).
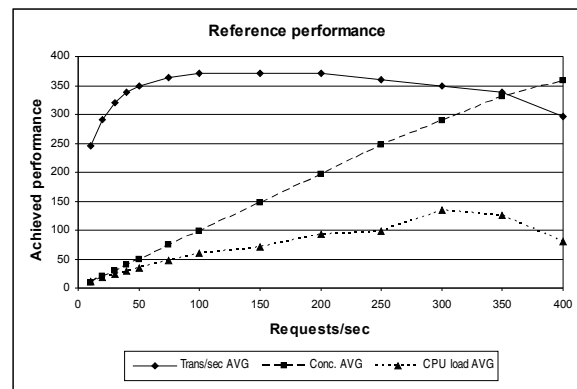

Figure 1. Reference performance

On this and subsequent graphs of the same type, the X axis represents number of concurrent requests attempted by the benchmark program, and the Y axis is used to show completed transactions/sec, achieved connection concurrency and CPU load. Measurement points are linearly interpolated for better visualisation. This graph is representative for this type of measurements, and shows that to achieve optimal performance a substantial number of concurrent requests is required, and that the performance curve has a characteristic maximum that depends on the saturation parameters of the whole system.

Graphical representation of the data reveals that the optimal performance for this test is with a concurrency level of 150-200 transactions/sec. After that number comes a decline in performance, up to the point where not all requests can be satisfied within the timeout time, in which case they are aborted.

To gain on conciseness and to shorten the total time it takes to finish the tests, further tests were made for concurrency levels of 50, 100, 200, 300, 400 and, where applicable, 500 and 600 simultaneous connections to the server.

3.2. On System Reliability

Measuring system reliability is a much more complex task than measuring performance, and the complexity grows with the size and complexity of the system itself, as special attention must be given to each of the components that make such a system. This work will not attempt to measure reliability, but will rather describe the expected effects particular components have on the system reliability.

4. Distributing the front-end

The front end encapsulates everything on the virtual data-path from the web client (browser) to the HTTP request handler, and every other network resource that is used to make the contact to the actual core of the system ("business logic").
Various parts include:

- The network infrastructure (network access devices, packet switching and routing equipment [1]).
- The Domain Name Service (DNS) used to look up the common name-IP address mapping of the web site
- The HyperText Transfer Protocol (HTTP) request handler [2]

In the traditional, standard architecture of web application services, the DNS service is handled by a separate (often distant) computer system which is completely unrelated to the web application. In such architecture, a single computer (server) often hosts all other components of a web application system: the web server, the business-logic code, and the database server. By taking charge of the DNS service, and separating the HTTP request handler from the other components, performance can be highly increased.

4.1. DNS Load Balancing

This is a method of distributing HTTP requests to multiple web servers. The DNS server is configured to respond to name-to-address queries with one IP address from a predefined list, either in round-robin, or a server load-aware fashion. Thus, the task of responding to HTTP requests is distributed between multiple web servers. To make this work, the instances of web application distributed on the servers must have some way of sharing internal state. This method offers great performance increase: the number of concurrently served requests per second almost

linearly increases as more servers are added to the setup.

To support this setup, the web application needs to be able to:

- Share user session data between the instances of the web application
- Share database backend between the instances of the web application

The first goal is easily solved by specifying that the session data should be placed on a network-shared file system (e.g. NFS), or by using a specialized session server daemon, such as "memcached". In such a setup, multiple web application instances are accessing the same session information (since session data is of comparatively small size, a single session server usually suffices).
Even though no actual implementation of the DNS load balancing has been made (due to the lack of proper equipment), the aforementioned components of such a solution were built during the course of this work.
DNS load balancing is one common way of increasing reliability by redundancy of web servers. Even though the simple implementation introduces a single point of failure in the form of the DNS server, more complex solutions distribute the DNS server itself. Session data servers are essentially a form of database, and appropriate solutions are available and discussed later.

4.2. Request Proxy

Most of web traffic is generated by "average" users with Internet access in their homes. Such access is nowadays still mostly implemented with 56kbit/s modems or ISDN adapters. On traditional web system architecture, the path this average home user's HTTP request takes can be represented by the following diagram:
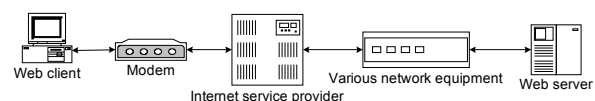
Figure 2. Network path of a HTTP request

Such low-bandwidth connections carry a often-overlooked and potentially disastrous paradox: while it is true that a network server ideally should be able to serve slow connections better than fast connections because requests can

be served completely while many others are still arriving, the real-life situation can often be quite the opposite.

Slow connections cause "slow requests" - while the web application program can serve an average request in 0.05s, the fact that the request comes from a 56kbit/s line means that (for an average page size of 38K) it takes almost 5.5s until the page is transmitted to the client. During that time, the instance (a fork or a thread) of the web server process that is doing the serving is doing nothing but waiting for the notification about the end of transfer. Such instances are only consuming valuable system resources: kernel structures (file descriptors, sockets, etc.), memory and database connections. Resulting resource starvation can cause the server to become unresponsive although the number of connected clients is fairly low.

The most efficient solution is to setup a buffering proxy server on a fast network in front of the actual web server. The purpose of this proxy is to act as a mediator between slow clients and the web servers, buffering the data coming from the server before forwarding it to the clients. Web server sees only fast requests with responses that are quickly transmitted away. Such a proxy server can be implemented on two layers:

- as a generic TCP proxy (on the transport layer), with a comparatively simple task of routing TCP connections and data, or
- as a specialized HTTP proxy (on the application layer), which can analyze and cache the data across the requests (so that a request is served without accessing the actual web server).

Since FERweb system is an explicitly dynamic system, with each request served from the database, HTTP caching is of limited usefulness, so only the effects of a generic TCP proxy were further explored.

For the purpose of this work, a simple proxy server program has been made. It was written in Python language and uses threading and synchronization objects to handle high amounts of network traffic. It separates the web server and the remote network client by providing a memory buffer which backlogs data coming from the web server process. The buffer accepts data at the speed the web server generates it, and streams it to the client at the rate it can accept data. The connection to the web server process is closed as soon the data is generated, so the process can

serve other requests while the proxy is sending the data in its own pace.

4.2.1. Benchmark setup and results

To simulate slow connections, network traffic shaping was employed, using "ipfw" (IP firewall) and "dummynet" (traffic limiter) from the FreeBSD operating system. Traffic shaper parameters were: 56kbit/s bit rate, 75ms latency, 5 queue slots and 1024 buckets.

Three computer systems were employed in the test: the web server, an intermediate machine with traffic shaping enabled, and a client computer on which the benchmark program (siege) was run (also without traffic shaping).

The available physical memory on the server was capped to 512MB in order to better observe resource starvation. The list of test URLs contained only dynamic pages, without static content (as these take comparatively insignificant resources to transfer and would skew the measurements), so these results are not comparable with the ones from the initial performance benchmark.

The difference is quite dramatic, and best seen when graphed:
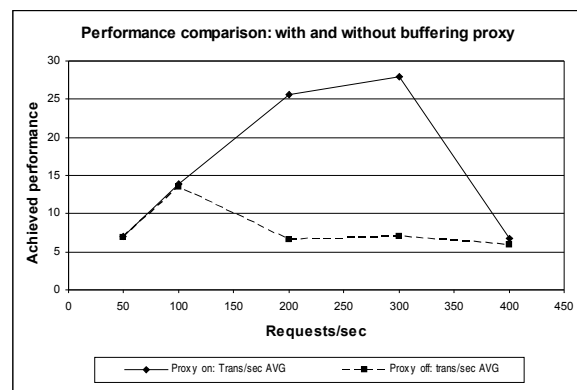
**Figure 3. Performance with and without a buffering network proxy**

Without a buffering proxy to offload sending data to the slow clients, the "knee" of the performance curve is situated at 100 completed transactions/sec. Observations made on the web server using system monitoring utilities have shown the system was severely in shortage of free memory, and excessive memory swapping was present. Adding buffering proxy moved the knee to about 30 transactions/sec which resulted in peak performance that is about 400% better than in the first case. This clearly indicates that adding a buffering proxy on the same machine as the web

server yields better utilization of the available hardware even if it means less free memory for the web server. With memory dedicated to the buffers, less web server instances are spawned which has the ultimate result of lowering the memory requirements.

4.3. Separating Static from Dynamic Content in the CMS

Previous results have opened possibilities for further research. Analysis of the requests to the server showed that, although requests for dynamic content make for only 4% in numbers, they take up to 500% more in process time.

In the light of these facts a new setup was made, similar to the original setup, but with a separate, lightweight web server configured for serving static images. This was implemented by setting up "Tiny Turbo Throttling HTTP Daemon v1.25" (THTTPD) server [7] on the same server, but on a separate network port, and modifying the CMS application to support it. New results show a significant increase in performance achieved by such separation.
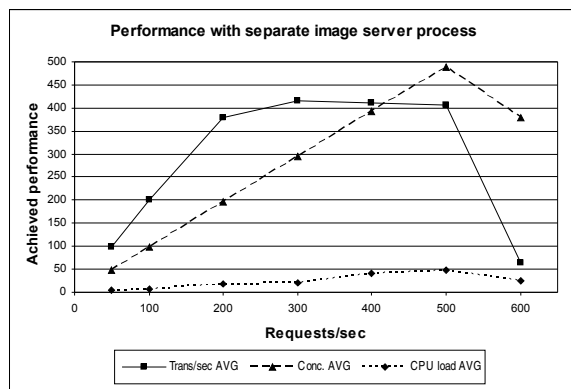

Figure 4. Performance achieved with separate web server for static content

Since these measurements were taken in the same conditions as the reference performance measurements (meaning with mixed static and dynamic content), the graph in Figure 4 is comparable to the initial performance graph in Figure 1. Though the performance in low requests/sec region is slightly worse, the knee of the performance curve has moved from 200 requests/sec to about 400 requests/sec (indicating increased scalability), and the peak number of served requests has increased by about 11%.

5. Distributing the Database

The most common method of speeding up database processing is distributing the database to multiple servers. Web servers can access the databases either through a special load balancing/distributing system or directly (with load balancing information kept on each server) as shown in Figure 5:
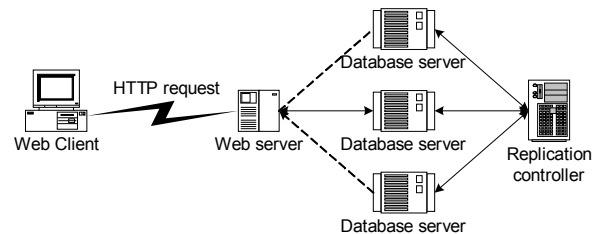
Figure 5. A web application utilising multiple database servers

As is the case with distributing the web servers via DNS load balancing, distributing the database yields both performance increase and improved reliability by introducing redundancy.

The database (PostgreSQL 7.3) was distributed using the PgCluster program package [8], which offers synchronous multi-master replication. PgCluster project at the time was still in development stage and not ready for production usage, but the available features were enough to setup a replication system and conduct measurements. The scheme in which the web servers directly access the database backends (as described in the above diagram) was employed, which required modifications to the database access code of the system (web servers pick one of configured database replicas as the default for the duration of the web transaction in a random fashion; each replica has an associated "weight" value that affects its chance of being chosen, so the faster servers can achieve higher utilization).

Since the focus here is on the dynamic content generation, benchmarks were conducted with a list of dynamic URLs only.

Three setups were considered. The first setup included the database on the same system as the web server, and with database-based optimizations (such as caching of the SQL queries) turned off. Without the standard database optimizations, and with the database server consuming the resources from the web server, the system was clearly under performing.

The second setup is similar to the first one, except that the database server application was moved to a separate server.

Finally, in the third setup, the database was replicated on two separate servers, while the web server runs on the original one.
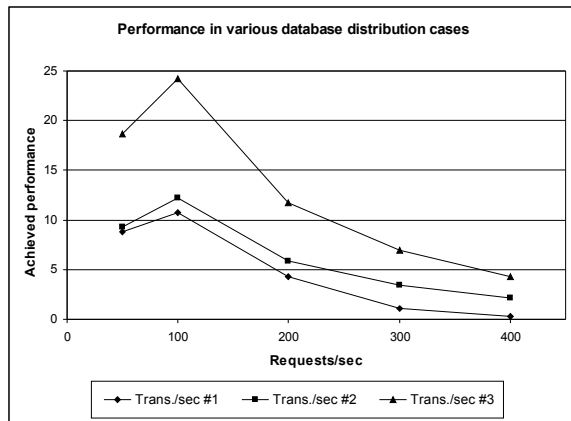
Figure 6. Performance of web application utilising multiple database servers

Results show that performance of a web application can be improved by distributing the database load almost linearly, which is why it is a popular and well understood [4] method nowadays. The effect of moving the database to a separate server had favourable influence on the performance, but the advantage is not large, especially while the number of clients is relatively low. Introducing another database server almost doubles the performance.

This method is also popular because it impacts the reliability favourably. Since the data of any application is usually much more important than the application code, in some cases database distribution is used as a sort of "live backup". In case of failure of one of the servers in a replication cluster, the data is still safe and viable on other servers, and hot-synchronization is usually much more convenient than restoration from a full backup archive.

6. Conclusion

This work resulted in numerous enhancements to the existing FERweb system. Several benchmarks were conducted to demonstrate and explore effects of distributing various parts of the web application to different computer systems. The efforts in improving the overall performance of the system have been very successful. It was shown that various strategies by themselves can have a huge performance impact while minimizing the impact on existing application code – in every case discussed, the changes to the application were minimal, and then only to the basic low-level components.

The web application system was extended to support several additional facilities:

- A separate image server system. The benefits should be at least those measured in this work, which are doubled scalability (with respect to achieved connection concurrency) and about 11% increase in peak transactions/sec.
- A load balancing DNS server and multiple web server systems, which should yield in linear increase of most aspects of performance. A session server is needed for sharing user state data between web server systems.
- Buffering proxy servers in front of actual systems serving dynamic web content help performance up to 400% in situations with many slow clients.
- Multiple database servers containing live replicas of the CMS data.

The performance improvements gained from modifications are a necessity for deployment of any web system in heavy-duty environments.

7. Acknowledgements

8. References

[1] Andrew S. Tanenbaum: "Computer Networks", Third Edition, Prentice-Hall 1996.

[2] RFC Document #2616: "Hypertext Transfer Protocol - HTTP/1.1"

[3] The World Wide Web Consortium (W3C) : "Extensible Markup Language", http://www.w3.org/XML/

[4] Bettina Kemme: Database Replication for Clusters of Workstations, Swiss Federal Institute of Technology, http://citeseer.ist.psu.edu/kemme00database.html

[5] The World Wide Web Consortium (W3C) : HTML 4.01 specification, http://www.w3.org/TR/html4

[6] Siege benchmark program: http://www.joedog.org/siege/

[7] Tiny Turbo Throttling HTTPD server: http://www.acme.com/software/thttpd/

[8] PgCluster: http://pgfoundry.org/projects/pgcluster/

[9] FERweb system description: http://www.fer.hr/webtech