Distributing a Web-based Content Management System - "FERweb"

Ivan Voras, Sonja Miličić

University of Zagreb,
Faculty of electrical engineering and computing,
Unska 3, 10000 Zagreb, Croatia
e-mail: ivoras.voras@fer.hr, sonja.milicic@fer.hr

Abstract

This work explores various ways of making the FERweb CMS system (a web-based Content Management System of the Faculty of electrical engineering and computing at the University of Zagreb) a "distributed system". Here, the notion of a distributed system is taken quite literally, and stands for executing as many components of the system as possible on as many different computer systems. The only limits here imposed are that of the practical value of such distribution – its end result must be either an increase of total performance delivered to the user of the system, an increase of overall system reliability, or providing distinct new functionality that was previously not possible, and that the efforts must be based on existing technologies used in the project already, with minimal impact on the existing code. Where appropriate and where available resources permit, measurements are taken and reported.

1 Table of Contents

2 The FERweb System

The "FERweb system" is a colloquial name for a custom-made web CMS (*Content Management System*), developed at the University of Zagreb, Faculty of Electrical engineering and computing. The system is a result of over three years of development and it has successfully been deployed as the Faculty's web site (which includes 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 as the programming language, and POSTGRESQL 7.3 database server. Its main characteristic is the concept of "portlets" - modules (or "parts of a web page") that behave as a distinct and independent subsystems. The modules control their HTML [HTML4] 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 end (also called business logic, just "logic", or "the web application"), and the database end (or back end).

3 Why Distribute a System?

This work focuses on three aspects of benefits for a system: increased performance, increased system reliability, and added new functionality that could not have been achieved otherwise. Of these reasons, the first two are of the most importance when deploying to a corporate environment with high expectations for concurrent access and where availability is of critical importance. The FERweb system started as a research project done by undergraduate students and faculty staff to satisfy their academic responsibilities, but has evolved in a self-supporting and viable project, and has surpassed its original goals. In particular, increasing attention has to be given to how the system performs in situations with many concurrent clients and demands for fault-tolerance. Also, as the system complexity grows, it must be able to exchange various information with other systems, both similar and dissimilar in their nature and purpose [Pavlinusic99].

As the FERweb system is currently designed to run in its entirety on a single system, the first step in accomplishing these goals is adding support for various parts of the system to be distributed on multiple computer systems. The distribution efforts will target the system on multiple levels – from basic low-level protocol handling, to higher level business logic.

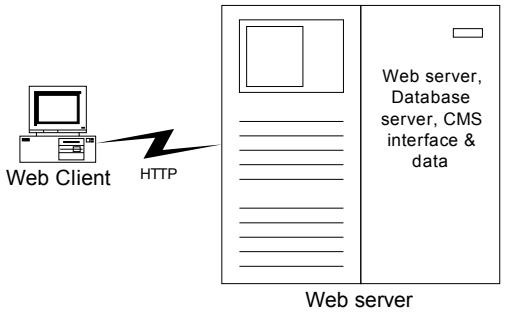

Figure 1. A simple web application setup, with all components residing on the web server system

3.1 Measuring Performance

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). It is measured using the SIEGE program. The 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 CPU power and resources to sustain the request load
- The list of URLs to fetch is created to contain all the (dynamic) pages contained in the system
- The lists of URLs can contain the static graphics elements (images) if so noted for each particular 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
- Each test took 5 minutes in length
- The concurrency level of the requests is varied and reported with the results
- The average transactions/sec and concurrency scores and their respective standard deviations are reported

3.1.1 Initial Performance

The reference performance measurements are taken on the following setup:

- Intel Pentium 4 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

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. The benchmark, as is the other benchmarks, was conducted with the SIEGE program under the standard protocol, from a computer on the same LAN without speed hindrance (100Mb/s), and including static graphic files.

Simultaneous connections	Transactions/sec.		Concurrency		CPU load avg.
	Average	Std. dev.	Average	Std. dev.	
10.00	244.84	8.83	9.77	0.22	11.42
20.00	290.84	6.03	19.73	0.21	17.51
30.00	319.45	4.21	29.60	0.31	24.34
40.00	337.65	3.49	29.51	0.27	28.69
50.00	348.64	3.92	49.07	0.74	34.77
75.00	362.99	5.36	73.68	0.40	48.10
100.00	370.34	9.92	97.46	0.61	59.61
150.00	370.83	3.20	147.73	1.28	70.09

Simultaneous connections	Transactions/sec.		Concurrency		CPU load avg.
	Average	Std. dev.	Average	Std. dev.	
200.00	371.66	8.56	195.83	4.23	91.97
250.00	360.50	9.01	246.37	1.46	97.93
300.00 (*)	349.24	10.35	289.49	7.74	134.28
350.00 (*)	339.09	18.43	331.23	2.10	125.68
400.00 (*)	295.94	43.49	358.17	26.72	80.17

Table 1. Initial performance benchmark results

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, 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 (it will be shown that the CPU load is often not the limiting factor of server performance).

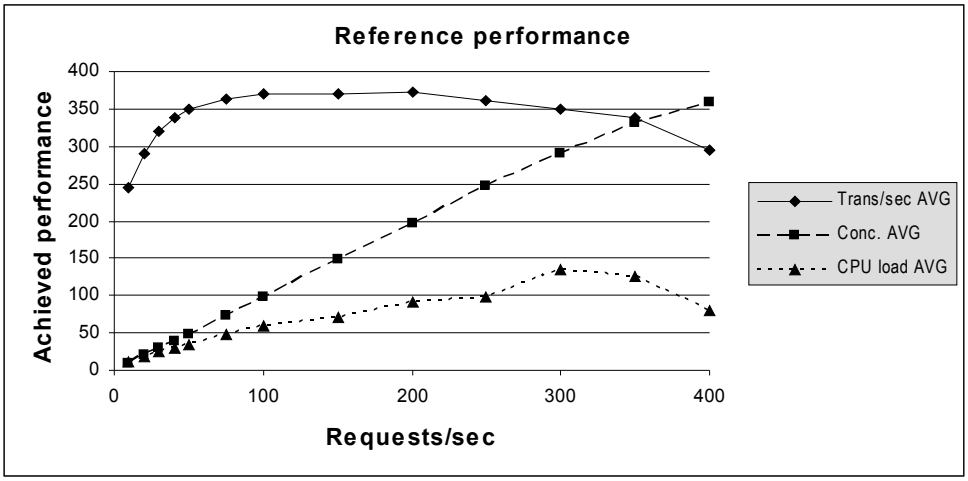

Figure 2. Reference performance graph

On this graph, 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 visual impact. 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. Those cases are marked with the asterisk sign in the table above. This result also presents an upper boundary for server performance – in no case will *this* setup perform better than measured here.

Note that, in this case, the transactions made have mixed requests for static images as well as for dynamic content.

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

3.2 About 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 the 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 (the business logic code).

The various parts include:

- The network infrastructure (network access devices, packet switching and routing equipment [Tan96]).
- 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 [RFC2616]

In the traditional and 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 an 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 responsibility to respond to HTTP requests is distributed between multiple web servers. To make this work, the instances of web application that are 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 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 filesystem (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):

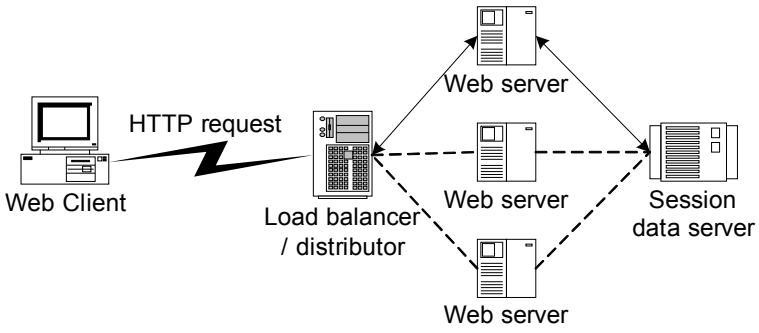

Figure 3. Load balancing/distribution using a session data server

The second goal, sharing database backends, is discussed in the appropriate section of this article ("Distributing the Database").

Even though no actual implementation of the DNS load balancing has been attempted (due to the lack of proper equipment), the aforementioned components of such a solution are build 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 a database, and appropriate solutions are available and discussed at a later time.

4.2 Web Request Proxy

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

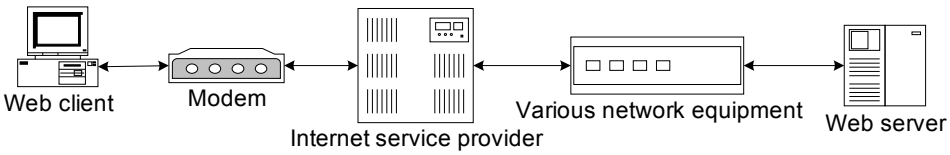

Figure 4. Network path of a HTTP request

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 in arrival, the real-life situation can often be quite the reverse.

Slow connections yield "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 that is doing the serving is doing nothing but waiting for the network notification about the end of transfer. Such instance is only consuming valuable system resources: kernel structures (file descriptors, sockets, etc.), memory (used for processing, data and network buffers) and database connections. In such case the resource starvation can cause the server to become unresponsive although the number of connected clients is fairly low.

The solution is to setup a buffering proxy server on the local network in front of the actual web server. The purpose of such a server 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. Since the

proxy server is on the same (fast) network as the web server, the 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, caching is not normally useful, so only the effects of a generic TCP proxy will be further explored.

4.2.1 Implementation of a TCP Proxy Server

For the purpose of this work, a simple proxy server program has been made. It was written in Python language and uses multithreading and synchronization objects to handle high amounts of network traffic. In it, each proxied connection instance can be represented by the simplified diagram:

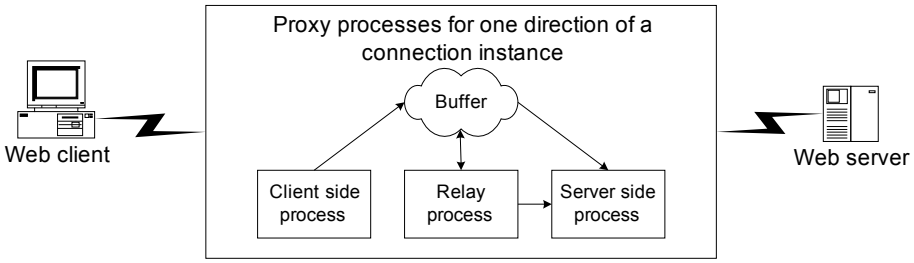

Figure 5. Buffering network proxy (the diagram shows one side of communication)

The processes execute simultaneously: the server-side process collects data from the server into the buffer, the client side process takes the data from the buffer (if available) and sends it to the client, and the relay process controls and synchronizes the flow of the data and takes care of connection setup, synchronization and shutdown (as well as making everything bidirectional).

A more complex form of a TCP proxy can be used in a similar way as the aforementioned DNS load balancing system, by distributing the requests to several redundant web servers.

4.2.2 Benchmark Setup and Results

To simulate slow connections, network traffic shaping is employed. Using IPFW (IP firewall) and DUMMYNET (traffic limiter) from FreeBSD, the following setup was created:

Bandwidth /connection	Latency	Queue slots	Buckets
56kbit/s	75ms/direction	5	1024

Table 2. Slow connections' simulation – traffic shaping parameters

Three computer systems were employed in the test: the web server, which has not been altered in any way, an intermediate machine, with traffic shaping setup as described above to simulate a slow ISP, and a "client" computer on which the benchmark program (siege) was run (without traffic shaping).

Because this benchmark is trying to measure resource starvation on the web server, its available physical memory was capped (in the FreeBSD loader.conf via `kern.physmem` directive) to 512MB. The list of test URLs contained only dynamic pages, without static images (as these take comparatively insignificant resources to transfer and would skew the measurements), so the results are not comparable with the ones from the previous benchmark.

Without a buffering proxy, with data passed through the intermediate system directly to the web server, benchmark results are:

Simultaneous connections	Transactions/sec.		Concurrency		CPU load avg.
	Average	Std. dev.	Average	Std. dev.	
50	6.90	0.25	48.53	0.34	4.72
100	13.53	0.27	96.98	1.15	12.67
200	6.57	0.45	166.54	8.16	29.76
300	6.99	1.79	234.77	23.24	20.01
400	5.86	1.69	320.83	14.34	16.93

Table 3. Performance of web server system with requests from slow clients

After introducing the proxy, the results are:

Simultaneous connections	Transactions/sec.		Concurrency		CPU load avg.
	Average	Std. dev.	Average	Std. dev.	
50	7.00	0.08	48.93	0.15	2.69
100	13.84	0.15	98.08	0.21	3.98
200	25.62	0.35	194.50	0.46	15.86
300	27.95	1.32	290.45	1.90	57.48
400	6.76	1.72	267.43	52.96	24.18

Table 4. Performance of web server system with requests from slow clients and with TCP buffering proxy

The difference is quite dramatic, and best seen when graphed:

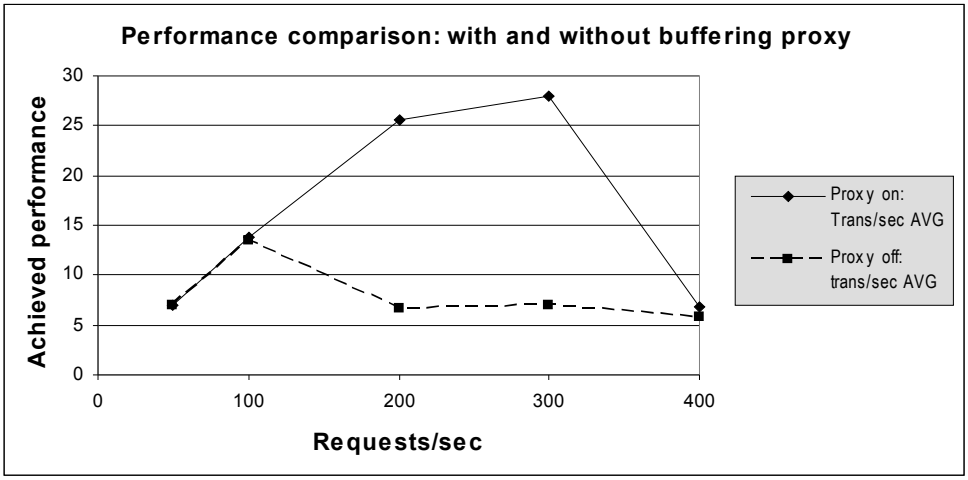

Figure 6. Influence of a TCP buffering proxy on web server performance with slow clients

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. Adding the buffering proxy moves the knee to about 300 transactions/sec with peak performance that is about 400% better than in the previous case. Also to be noted is the behavior of the "CPU load" series: clearly, with slow clients, the server is not CPU-bound, but rather other resources are in demand. Observations made on the web server using system utilities show that the system was severely in shortage of free memory, and excessive disk-swapping was present.

4.3 Separating Static Content from CMS Dynamic Content

Previous results offer possibilities for further investigation. Since it was shown that web server instances consume significant amount of resources, a question arises if that effect can be alleviated. Analysis of the types of requests presented to the server show that although requests for dynamic content make for only 4% in numbers, they require up to 500% more time to process.

4.3.1 A separate server process for static image files

In the light of these facts, a new setup was made: separate, lightweight web server process will be used for serving static images, and dynamic content will remain served by the original web server (both running on the same computer system). This was implemented by setting up "TINY TURBO THROTTLING HTTP DAEMON v1.25" (THTTPD) server on a different network port than the usual (APACHE) server, and changing the image URLs in the CMS application accordingly. New results show a significant increase in performance achieved by such separation:

Simultaneous connections	Transactions/sec.		Concurrency		CPU load avg.
	Average	Std. dev.	Average	Std. dev.	
50	99.06	2.54	48.98	0.26	3.44
100	201.19	1.29	98.18	0.13	7.43
200	378.64	5.84	195.78	0.33	17.62
300	415.67	4.05	293.60	1.24	21.20
400	411.25	4.11	392.03	0.52	40.19
500	405.36	5.41	488.53	1.17	48.10
600	63.36	6.71	379.44	41.98	109.64

Table 5. Performance of web server achieved using separate web server process for static images

These measurements were taken in the same conditions as the reference performance measurements and are directly comparable to them. Not only has the knee of the transactions/sec performance curve moved from 200 requests/sec to about 400 requests/sec (indicating increased scalability), but the peak number of served requests has increased by about 11%.

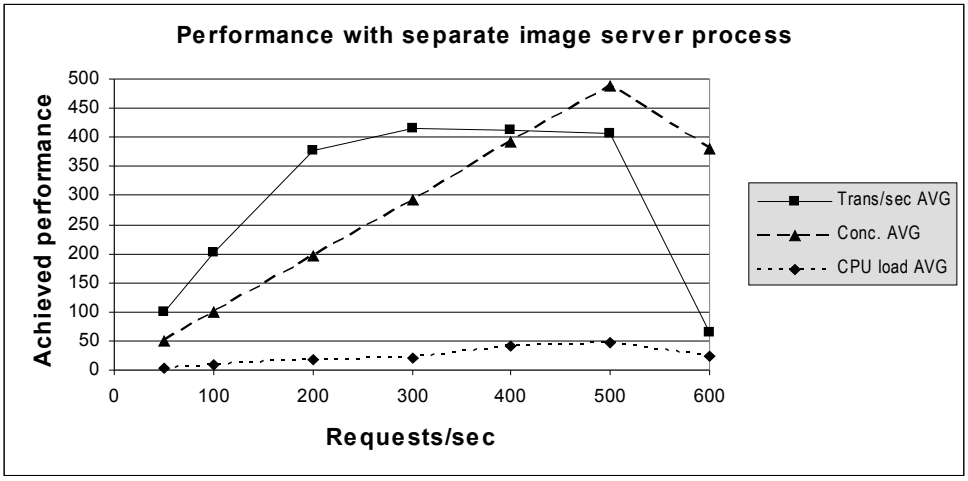

Figure 7. Performance with separate image server process

A further investigation can be conducted about the feasibility of adding a specialized computer system for serving only the static images. Such setup could be implemented by using a network

shared filesystem to share the static data, or a synchronization mechanism such as the RSYNC program.

5 Distributing the Business Logic

Here, the goal is not so much to increase the performance of the web system, but rather to add new functionality to the system, by shifting the workload of executing modules and generating HTML representation off the main web server and into auxiliary or specialized servers. The new possibilities that are opening with such an architecture are very exciting [Turner03]:

- Establishing a separate server for e.g. collecting and administrating news articles, with a separate database, and have it generate HTML representation of the module in-site, instead of transferring the data to the web server
- Replicating the database and having some modules execute on separate computer system although they are accessing the same database, to gain improve the speed of transactions
- Delegating a function of the web system to a separate system, such as user authentication and authorization
- Monitoring a device connected to a remote system by having it generate data in HTML form and presenting as a FERweb module.

Implementation of such possibilities required modifications to the core layout (page generator) and module subsystems of the FERweb system. In particular, the database schema which describes the available modules was extended with information about the whereabouts of the remote module (IP address and request information) and the type of the module.

5.1 Extending the Layout System

As a consequence of its highly modular design, the FERweb system supports using (and extending) multiple application programming interfaces (APIs) for use with its modules (also called portlets). A relevant subset of the module information schema used in the system is:

- `id` – Unique module id. Modules are referenced by their id number
- `name` – Module name
- `description` – A description of the module
- `type` – Type of API the module uses
- `api_data` – Data relevant to the API subsystem for the particular modules

An example of module information records is:

id	name	description	type	api_data
10	mod_news	News module	0	-
52	mod_directory	Faculty staff directory (yellow pages)	1	-
103	mod_rpc_tini	TINI interface module	2	• url=http://10.0.1.32/xmlrpc • method=FERweb.rpc.tini • mode=tiny

Table 6. Examples of module information present in the system

The main output generator subsystem, the "layout loop" treats modules as black-box entities, and can interface with them in various ways (the type field serves as the input to a switchboard-like functionality). Relaying on that support, it was straightforward to add another interface for modules residing on different system(s) - "remote" modules.

With this newest addition, there are three interfaces with which the modules can be realized:

- the Procedural API, in which modules are implemented as PHP functions
- the Object-oriented API, in which modules are implemented as PHP classes
- the Remote modules' API, in which modules are implemented as remote XML-RPC functions

The process of handling and generating data can be represented as such:

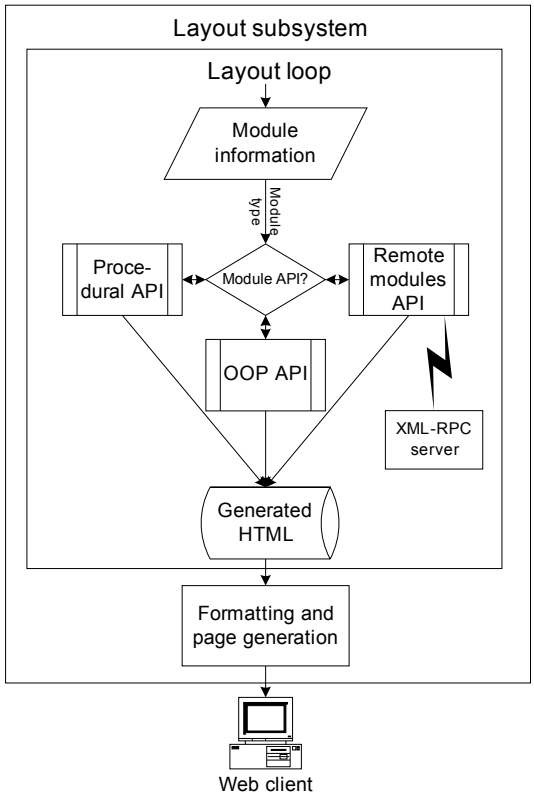

Figure 8. How the layout subsystem interacts with various module interfaces

The remote modules API is implemented using XML-RPC [Winer99] mechanism for network communication. The XML-RPC is a light-weight RPC (Remote Procedure Call) protocol which is in turn based on communicating with XML [XML] formed requests and replies over the HTTP protocol. In this case, a remote FERweb module is implemented as a XML-RPC function residing on a remote server.

The system passes important state information (like the environment of the layout system and security information) to the remote module in the XML-RPC parameters (since such information can be quite large in size, a special "tiny" mode is implemented, designed for e.g. embedded systems, in which only the most basic information is passed). The remote procedure's prototype is declared as:

```
XML-RPC function (struct $system_state, struct $module_info)
  returns struct { "title", "html",... } | struct { "error" }
```

The `system_state` parameter is described in the paragraph above, and the `module_info` parameter contains module information as described at the start of the chapter.

The function returns either the data necessary to display an HTML or graphical representation of a module, or an error identifier (The details of specification and implementation can be obtained from the authors the FERweb development team).

Defined like this, the module specification becomes language-independent. XML-RPC libraries exist for all major languages, modules are not confined to be written in PHP anymore. New modules can be created with e.g. C/C++, Java, Python etc.

Similar works in this field, of which Resource Description Framework ([RDF], defined by the W3C) is most popular, approach the problem on a different level, specifying a syntax (in XML) for sharing abstract data, and is in the process of finding its audience. In contrast, it is the authors' opinion that the method described in this work, which is more suited for todays CMS implementations, which are still based on proprietary technologies and interfaces.

The system introduces the possibility of cooperation between various CMS systems, achieving the goals of interoperability, and/or enhanced performance. Multiple CMS systems can share data and code in a way that is transparent to the end user.

A future work might explore different methods of distribution and/or remote call implementation, such as CORBA (Common Object Request Broker Architecture) or SOAP (Simple Object Access Protocol).

5.2 An Implementation of a FERweb Remote Module on an Embedded System

One of the major development efforts of today is in the field of Pervasive computing – integrating various appliances into Internet-published services. The effort is still young and there are many obstacles [Kolberg03]. The remote modules system described in this work can help with its simplicity.

To demonstrate versatility and power of such approach, a FERweb module was implemented to run on the Dallas Semiconductor's TINI (TINY INTERNET INTERFACE) [TINI], an embedded platform based around a DS80C390 microcontroller (Intel 8051-compatible), and running a version of Java VM supporting a subset of JAVA1.1 [Java] platform. The TINI board is directly connectible to a 10BaseT Ethernet and can act both as a TCP/IP client and server. The board in question is also connected to a number of sensors and controllers: three temperature sensors measuring temperature at two places in an AC-DC computer power supply and the room temperature, and switches for the cooling fan and a light bulb.

Using the provided SDK tools, which include most of the JAVA1.1 platform API and extensions for accessing the board internals, and a compact XML parser package (NANOXML/LITE, designed for use in embedded environments), an XML-RPC server application was made that serves a FERweb module. The module displays the state of various sensors and provides a way to toggle the light switch, thus demonstrating a two-way communication with the module.

The module returns sufficient data to be displayed by the layout system as any other module:

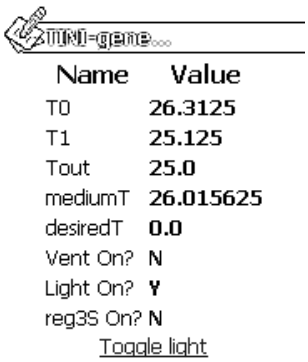

Figure 9. FERweb module displayed in a user-accessible page (screenshot)

Important fact here is that all of the module processing is done on the TINI board, including processing the inputs (e.g. the click on the link displayed above), and generating the HTML output.

As the TINI board is not a high-performance system (the microcontroller runs at 40MHz, with 512KB total SRAM memory, and the sensor readings are fairly slow), usage of some kind of caching mechanism is recommended in case of intensive use.

6 Distributing the Database

Empirical evidence (monitoring the server status in real-time by e.g. the "TOP" utility) shows that, consistently most of the system load is being made by the database server. As such, it is a primary candidate for speedup. Multiple web servers access multiple database server either through a special load balancing/distributing system, or directly (with load balancing information kept on each server) as shown here:

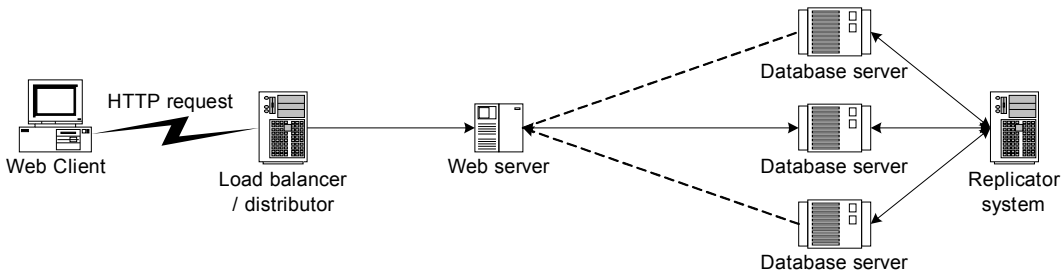

Figure 10. Distributed database

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

The database (POSTGRESQL 7.3) in this work is distributed using the PGCLUSTER program package, which aims to offer synchronous multi-master replication. The project is 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 (fortunately, the modular design helped a great deal, as the database code is isolated and centralized). 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.

The first benchmark is showing the initial performance, with the database on the same system as the web server, and with database-based optimizations (caching of the SQL queries) turned off.

Simultaneous connections	Transactions/sec.		Concurrency		CPU load avg.
	Average	Std. dev.	Average	Std. dev.	
50	8.76	1.39	49.69	0.19	7.36
100	10.76	1.27	97.99	0.44	8.97
200	4.25	0.91	191.83	0.98	2.60
300	1.12	0.23	250.19	5.31	1.41
400	0.31	0.17	317.97	27.25	0.53

Table 7. Initial performance of the web system (oriented towards heavy database usage)

The performance achieved was quite low. Without the standard database optimizations, and with the database server consuming the resources from the web server, the system is clearly under performing.

More interesting results are with the database offloaded to a separate computer system (to explore the effects of relieving the web server of running the database), the next set of results is:

Simultaneous connections	Transactions/sec.		Concurrency		CPU load avg.
	Average	Std. dev.	Average	Std. dev.	
50	9.31	1.02	49.71	0.15	18.31
100	12.17	0.67	99.19	0.27	19.78
200	5.87	0.59	193.12	0.97	6.69
300	3.34	1.07	271.12	4.35	4.11
400	2.15	0.32	368.89	18.02	1.13

Table 8. Performance of the web system with database moved to a separate and dedicated system

The results shows that by offloading database computations to a separate computer system, the web server was free to do more work. When compared to the previous results, it is clear that running the database server on the same machine as the web server server can throttle the overall performance. The third benchmark was made with a second database replica added on a system of identical characteristics:

Simultaneous connections	Transactions/sec.		Concurrency		CPU load avg.
	Average	Std. dev.	Average	Std. dev.	
50	18.64	0.63	49.60	0.16	18.31
100	24.23	0.59	98.30	0.38	23.54
200	11.76	0.49	191.14	1.09	1.41
300	6.91	0.23	255.09	3.36	1.36
400	4.31	0.18	329.59	20.85	0.67

Table 9. Performance of the web system with database replicated on two separate and dedicated systems

A graphical representation gives insight into the achieved performance:

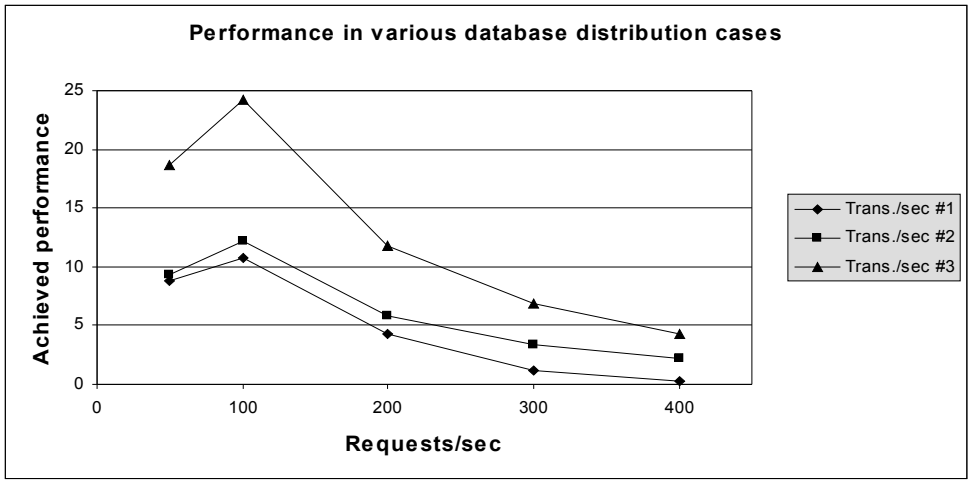

Figure 11. Comparison of influence of various cases of database distribution on the web application performance

Results show that performance of a web application can indeed be improved by distributing the database load almost linearly, which is why it is a popular and well understood [Kemme00] method nowadays.

This method is also popular because it impacts the reliability favorably. 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.

7 Conclusion

This work resulted in numerous enhancements to the existing FERweb system:

- Support for running multiple application instances behind a DNS load balancer and/or a buffering TCP proxy
- Support for accessing static images from a separate web server application
- Support for "remote modules", executing on arbitrary remote servers, and interfaced to by the XML-RPC protocol
- Support for accessing multiple backend databases in a load-balancing fashion

Several measurements were conducted to demonstrate and explore effects of distributing various parts of the web application to different computer system. 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.

In summary, a diagram of an example system employing most of the approaches discussed in this work follows:

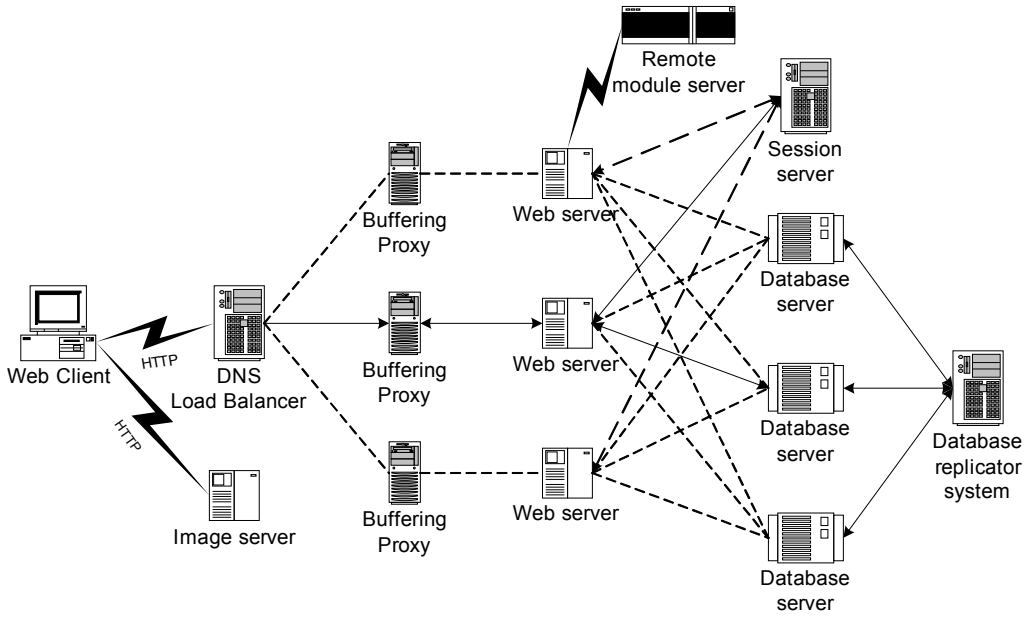

Figure 12. Distributed web based Content Management System

There are significant additions between this diagram and the one in Figure 1:

- Separate image server system. The benefits should be at least those measured in this work: double scalability (in respect to achieved connection concurrency) and about 11% increase in peak transactions/sec.
- 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 put in front of actual systems serving dynamic web content improves performance up to 400% in situations with many slow clients.
- Remote modules offer significant new functionality, ranging from interoperability with remote content management systems, to communication with embedded systems.
- Multiple database servers contain live replicas of CMS data, to be shared between web server systems.

The remote modules' feature offers unique features for extending system modularity across disparate environments, and interoperability with many kinds of equipment. The performance improvements gained from modifications are a necessity for deployment of any web system in high-demand environments.

8 Software Used in This Work

Software used in this work is Open-sourced, and as such is freely available off the Internet. Here is a list (and descriptions) of the software, with URLs of their respective homepages.

- APACHE 1.3 - A popular, modular and extensible Open source web server. Homepage: http://httpd.apache.org/
- PHP 4.3 – A popular web Open source application language. Homepage: http://www.php.net/
- POSTGRESQL 7.3 – A powerful open source relational database. Homepage: http://www.postgresql.org/
- SIEGE 2.59 – A web server benchmark program. Homepage: http://joedog.org/siege/

- MEMCACHED – A distributed object caching system. Homepage: http://www.danga.com/memcached/
- THTTPD – Tiny/turbo/throttling HTTP server. Homepage: http://www.acme.com/software/thttpd/
- TINI SDK – TINI System Development Kit. Homepage: http://www.ibutton.com/TINI/index.html
- NANOXML/LITE – a compact XML parser designed for embedded environment. Homepage: http://nanoxml.cyberelf.be/
- PgCluster – The multi-master and synchronous replication system for PostgreSQL. Homepage: http://www.csra.co.jp/~mitani/jpug/pgcluster/en/index.html

9 References

- [Tan96] – Andrew S. Tanenbaum: "Computer Networks", Third Edition, Prentice-Hall 1996.
- [RFC2616] – RFC Document #2616: "Hypertext Transfer Protocol - HTTP/1.1"
- [TINI] – Dallas Semiconductor : "TINI Board", http://www.ibutton.com/TINI/hardware/index.html
- [Java] – James Gosling, Frank Yellin, The Java Team: "The Java(tm) Application Programming Interface, Volume 1: Core Packages", Addison-Wesley, 1997.
- [XML] – The World Wide Web Consortium (W3C) : "Extensible Markup Language", http://www.w3.org/XML/
- [Turner03] – Mark Turner, David Budgen, Perl Brereton: "Turning Software into a Service", IEEE Computer, October 2003
- [Kolberg03] – Mario Kolberg, Evan H. Magill, Michael Wilson: "Compatibility Issues between Services Supporting Network Appliances", IEEE Communications, November 2003
- [Kemme00] – Bettina Kemme: Database Replication for Clusters of Workstations, Swiss Federal Institute of Technology, http://citeseer.ist.psu.edu/kemme00database.html
- [Pavlinusic99] – Dobrica Pavlinusic, Alan Lovrenic: "Integration of Databases and World Wide Web based on Open-Source Technologies", presented at IIS '99 at Varazdin, Croatia
- [Winer99] – Dave Winer, 1999: "XML-RPC Specification", http://www.xmlrpc.com/spec
- [RDF] – The World Wide Web Consortium (W3C) : Resource Description Framework, http://www.w3.org/RDF/
- [HTML4] – The World Wide Web Consortium (W3C) : HTML 4.01 specification, http://www.w3.org/TR/html4/
- [SOAP] – The World Wide Web Consortium (W3C) : Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/soap/

10 Acknowledgments

We would like to thank our friends and colleagues that have supported us all along and helped us create this work, especially our menthor Mario Žagar, Ph.D, full professor, and Kristijan Zimmer, B.Sc.