Prototype of High Performance Scalable Advertising Server with Local Memory Storage and Centralised Processing

Jakub Marszalkowski

Prototype of high performance scalable advertising server with local memory storage and centralised processing

Jakub Marszałkowski

Institute of Computing Science, Poznań University of Technology,
Piotrowo 2, 60-965 Poznań, Poland
jakub.marszalkowski@cs.put.poznan.pl

Abstract. Advertising servers play an important role in the entire contrary e-business. In this paper an approach for a high performance ad server is proposed. A prototype of a new architecture is presented: system achieving scalability by multiplication of ad servers with separate local memory storage and single managing server for entire cluster of those, providing data processing and ads preparation.
The paper includes benchmarks of alternate technologies possible at the design stage, results of the stress tests of the prototype, as well as the data from its performance with real web traffic. Limitations of the proposed solution are discussed. Although the prototype is created on a specific platform, all technologies used are widely available or have replacements on other platforms, granting generality to the proposed solution.

Keywords: internet advertising, ad server, performance

This research was partially supported by a grant of National Science Centre NN519643340.

1 Adserver

Advertising server or *ad server* is a system specialised in serving advertisements. It operates by receiving HTTP request and sending appropriate file to client terminal [8]. Ad server can provide many additional functions, e.g. use of business rules for ad serving, tracking of internet user and/or web pages and targeting ads with this data, income optimisation based on earlier results, dynamic creation of ad units, calculation of statistics of users and/or pages, and many more. Moreover, providing ads as quickly and foolproof as possible is of greatest importance. Advertisements loading too long will not be noticed by user, and cannot sell [3][7].

There are numerous research papers with concern in ad servers subsystems providing optimisation or targeting. Most up to date surveys of these are offered by [6] and [2]. This paper will be dedicated to the matter of ad server performance. Proposed solution is designed to work in a single HTTP request model:

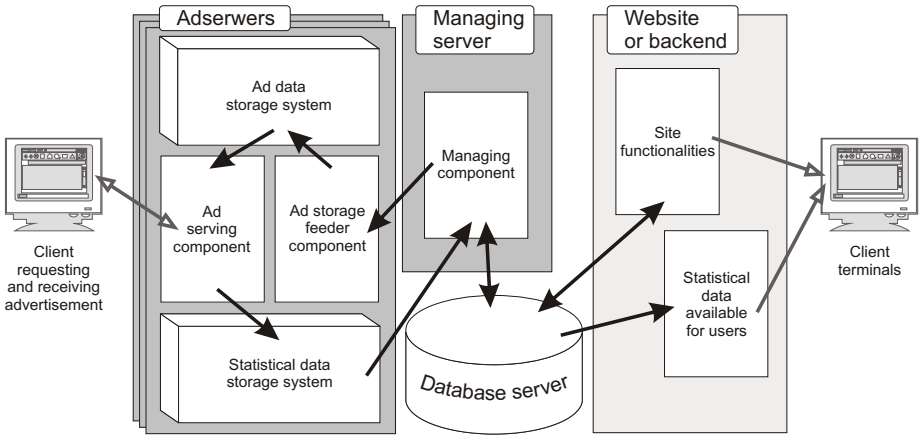

Fig. 1. Proposed ad sever architecture scheme.

client receives from ad server a single file with personalised content. This can be a dynamically created graphic file, like a banner containing some of the page usage statistics, as offered by numerous rankings and services providing statistics measurement [10]. Alternatively it can be JavaScript code, with some personalised variables included, like applets showing how many users read or liked an article. Graphical look of the last one can be either HTML/CSS generated or raster image embedded in JS with base64-encoding [11]. If an additional image file has to be loaded by JS code, it will be assumed, that it comes from a separate static content server and is easily cached by browser. This takes it out of scope of this paper.

Efficient techniques used for serving static files, without server side scripts and Database Management System (*DMBS*), cannot be used for ad serving. These disallow inclusion of page usage statistics or user specific variables into ad code, as well as gathering statistical data. For ad serving on each request a script must be run that will communicate with some data storage twice: to read personalised ad data and to write data of this event for statistics gathering. Efficient method of solving this problem is proposed in this paper.

For the prototype realisation, most popular open source platform LAMP [9] was selected, although ideas presented can be used on any platform. All tests were performed on computer with Intel Pentium Dual E2180 2.00GHz processor and 2GB RAM operating under FreeBSD v8.1 64 bit, that also serves as hardware for the prototype. As HTTP server Apache v1.3.42 with PHP v5.2.14 was used. DMBS used was MySQL v5.1.49.

2 System architecture

The proposed architecture is presented in Fig. 1. Its part of greatest importance is the *ad serving component*, the only one that communicates with the client supposed to receive an ad. This component is using two local storage systems as described earlier: *ads storage* and *stats storage*. The *managing component* is working on the statistical data from ad servers and other data from the main database. It generates the ad codes for serving, and sends them to the *feeder component* that places them in the ad servers local storage. Aside of this part, there can be a website, making use of the gathered statistical data, or being a back-end for ad server, allowing to set ad campaign parameters.

Such a structure offers good scalability, where more network traffic can be covered by introducing an additional cluster of several ad servers with single managing server. For this, a load balancer that directs users to the specific ad servers according to IP addresses, will be also necessary. "Servers" in this architecture can be separate machines, virtual machines or cores of a processor and in the low end scale even processes multitasking within single core. In the last, managing component can contain the feeder within itself and will be separated from ad servers rather by manner of time, for example working for a second every minute.

2.1 Ad serving component

Ad serving component is separated from the main database. For best performance both of its storage systems should be RAM operating. In case of building clusters of ad servers, machines running them do not need hard drives at all. As both storage systems differ in requirements, it is possible that separate technologies should be used to serve them. Ad serving component performance is achieved by simplicity, its algorithm, almost without simplifications, could be like this:

1. From ad storage fetch ad code appropriate for client or web page, as received in HTTP request.
2. Send code of graphic file or script with proper HTTP headers.
3. Write data of this event to the stats storage.

The most important refinement here is that ad serving component assumes that appropriate ad code always is in ad storage. Memory serves as storage, not as cache. Common solutions work oppositely. Facebook code [13] checks if data is in L1 cache, then in L2 cache and then even reads it from the database, at the end writing both caches. This handles many general purposes well, but not necessarily ad serving. With proposed build there are no cache misses and no pessimistic scenarios. The ad serving component never does anything except simplest possible fetching ad code and serving it. Obviously, the feeder component must be run properly to ensure that the ad code truly is in the ad storage.

2.2 Centralised processing

For the best performance, the ad serving component does not operate neither on the ad code, nor on the event data gathered for statistic. Both of these parts of the process are performed by the managing component. At best, for many events and a number of ad servers at one time. Exact functions of the managing component will vary according to requirements of website or advertisement optimisation and targeting used. The presented version is designed for website where prototype is presently tested, i.e, a ranking of websites that is generating banners with usage statistics:

1. Init, connect to DBMS.
2. Read stats storage. Erase it.
3. Do sanitization on entire input data read from stats storage.
4. Aggregate data grouping it by user visits.
5. Check in database which visits are unique.
6. Create lists of visits to insert or update.
7. Create list of member websites for stats update.
8. Read list of all member sites with their stats and scores. For each:
   (a) Calculate new statistical data and ranking position.
   (b) Generate new ad codes where required.
9. Update the database.
10. Send all the new ad codes to appropriate feeder components.

On each step from 3. to 9. the data processing is moved from the ad server to the managing server. This allows performance gain both on such an ad server and globally. Performance gains come from aggregation of event data before processing, especially before any DBMS operations. For example with creation of a key from IP adress and web page identifier, visits can be effectively summarised under such keys. The gain there will reflect the ratio of views per visit on the operated websites.

If no DBMS is used as stats storage, writing there input data without sanitisation, can also provide a huge performance gain. Data from all events can be sanitised as one string in one operation in step 3. Otherwise for each ad served at least the IP address of a client and the referrer address should be sanitised. It was measured that such single operations was taking 40-120 μs, while only ca. 7 000 μs when done globally for 20 000 ads displayed. Another boost can be achieved, by limiting bottleneck in form of numerous operations competing for access to database: like at least two or three queries with each ad served. Presented managing component will perform four queries per run, manipulating all the necessary rows at once. Sometimes programming tricks have to be used for that. No SQL standard known to author, allows to update many rows each with different data in one query. Usually there will be need to update hundreds or thousands of rows containing visits or websites statistics. Walk-around is achieved with insertion of all update data with appropriate keys to temporary MEMORY table and then doing single update from this table.

If the ad server is using graphical banners, process of generating them takes time, especially compression of used graphical format. Thus, it should be performed as rarely as possible. Additional marker of actualisation time should be assigned for each member website according to its average popularity. There is no need to generate new banners every minute for a website that has 48 visitors a day.

2.3 Local memory storage systems

Choice of technologies for the storage systems is highly dependant on their requirements. Ad storage requires a possibility of reading the ad code quickly. Size of the ad code can differ. For graphical banners this can achieve even 100kb each, and this has to be stored for several thousands of websites. The client related data for JS file will be rather limited to hundreds of bytes, but has to be stored for tens of thousands of users. On the other hand, stats storage requires the ability to quickly add current event data to the data already stored. Sizes of stats data can vary again, for example: the IP address, website identifier and the referrer address take together 280 bytes. This data will be stored for up to hundreds of thousands of events before they get processed. Still, contemporary computers have enough RAM to provide space for all this storage.

The realisation of both reading of ad code from ad storage, and putting the event data in the stats storage require achieving a lock on the container, as there are other processes that might be writing it. For stats storage numerous separate containers can be used instead of a single one, to reduce access conflicts. Efficient method of selecting container can be for example taking last digit of IP address.

Following technologies allowing for RAM storage in PHP were identified during research: System V shared memory support (*shm*) [12], Alternative PHP Cache (*APC*) [12], Memcached [5], MySQL MEMORY Storage Engine [14], MySQL Cluster [14] and tmpfs [16]. All with exception of APC can be used in numerous other programming languages.

First two are shared memory systems available for PHP. Shm allows creation of data segments of given size. To read a variable from a segment a connection by appropriate identifier must be achieved. Numeric variable identification used is important difficulty. Parameters like the website names would need to be hashed to read the stored data. Shm has no locking mechanism, usually System V semaphores (*sem*) would be used for this. APC is an opcode cache for PHP, preventing compilation of scripts on each request. It also offers shared memory system working simpler than the shm. There are no segments, variables are accessible by `string` identifiers, and it has built-in collection of locking mechanisms. From the last the fastest: spin Locks [15] was taken into account. For both technologies there is no possibility to append anything to data already stored in RAM. It would require reading of entire variable and then rewriting it.

Memcached is a distributed memory caching system working in client-server model. It was widely tested and proven that locally working Memcached is slower than APC [17] [4]. Also Facebook uses APC as faster L1 cache and Memcached

Table 1. Time measurements (μs) from tests of technologies for data storage.

measurement	avg	std	measurement	avg	std
Database connection	505.0	22.6	**For stats storage:**		
			tmpfs+flock	79	11
			tmpfs+sem	82	11
For ad storage:			APC+sem	642	214
APC	7.4	0.8	APC+flock	671	205
shm+flock	58.6	41.4	shm+sem	2353	427
tmpfs+flock	81.2	7.3	shm+flock	2389	226

as slower L2 cache [13]. This project does not need Memcached distributed functionality, scalability is introduced by its own architecture. On this basis, finally Memcached was excluded from speed comparison.

MySQL's MEMORY Storage Engine option creates db tables entirely in RAM (with few limitations). MySQL Cluster is distributed database system largely using memory storage. For both of them DBMS connection is required to access to data storage. Preliminary measurements for DBMS connection acquisition showed times unacceptable in comparison to entire data access for alternative methods (cf. Tab. 1). Thus, both technologies were excluded from more detailed tests. Please note that MEMORY tables are still used by managing component to fasten DBMS operations.

Tmpfs is a temporary file storage, working like a mounted file system, but stored entirely in RAM, namely a RAMDisc. Variable storage is difficult here, as it primarily stores files. It allows opening files in append mode and adding data at the end of the file, without reading entire contents. Tmpfs has no locking mechanism, usually file locking (*flock*) would be used for this.

For every time measurement PHP function `microtime()` returning unix timestamp with microsecond resolution [12] was used. Despite most obvious connections of technologies and locking mechanisms, shm can be used with flock, and tmpfs can be used with sem. Although semaphores are of very limited number (only 10 in FreeBSD) and thus cannot be used for locking access to ad storage containing thousands of variables. Both locking mechanisms can be used for stats storage. For flock, a handler of a file located in tmpfs was used to achieve maximum performance. Tests for ad storage were performed with real graphic banner of size of 5557 bytes. Tests for stats storage were performed with appending 280 bytes to 500kb of data already gathered. Additional tests, not included here, confirm that these choices had no matter, and choice of fastest technologies is doubtless. Results achieved are presented in Table 1.

APC's read time was absolutely unrivalled, on average it was more than twice as fast as only achieving a lock with semaphores. Things look completely different for stats storage. APC and shm had to read entire variable, large one for their standards and then rewrite it. APC also required external locking: reading and then writing such a variable cannot be considered an atomic operation. Both their times prove to be unacceptable. Tmpfs with flock unexpectedly achieved

slightly better results than with sem, although flock is almost two times slower than sem. This can be explained by considering operation of obtaining file handle, flock requires it, but same is tmpfs file opening operation, and there come savings. Also for tmpfs time of performing append will be completely unrelated to capacity of data already gathered. That would increase robustness. Choices are obvious: APC should be used for ad storage while tmpfs with flock locking for stats storage.

Last matter to discuss here are limitations of volatile memory. APC is lost with Apache restart. This causes no real issue, just upon Apache start-up feeder component has to be run. Tmpfs is lost on system down, and there partial statistical data can be lost. Power failures can be easily covered by backup power solutions, seconds are needed to process entire data and store it in DBMS. For the same reason at worst statistics from dozens of seconds would be lost.

2.4 Uncommon HTTP server and DBMS configuration

Ad server, even a typical one, works differently from standard www server. It is serving web traffic of different characteristics with different files. In normal www serving, after first request for HTML file, next ones for graphic files and other includes come in a short time. To maintain such traffic standards of persistent connections were created in HTTP 1.1. In case of ad server subsequent query comes only when client loads another web page connected to this ad server. That would be at delayed or will not happen at all. First cases of stress testing performed on prototype, until default setting allowing persistent connections was disabled, lead to crash of Apache in a dozen of seconds. For the same reasons as described abovem, within the same load Adserver is operating much more unique clients, requiring more connections to get one file. Thus Apache had to be recompiled to limit of 2048 processes.

Size of files sent by ad servers can be often well known, or at least distribution of these sizes. On prototype server most used size was small graphical banners all below 5600 bytes. According to this sent buffer can be set to a size, that will fit entire such files at once. Also for small graphical files and even smaller JS scripts even size of HTTP headers sent by server can matter. Using shortest forms of server signatures can give gain of even 200 bytes. For smallest file used by prototype, being 2400 bytes large, this makes 8% difference.

In presented architecture DBMS will work only with managing component. Even if there is a website connected to it, it can have separated DBMS. Queries performed by managing component are limited, and quite specific, i.e., are very large both in number of affected rows as well as in sheer length, due to very spacious lists of parameters.

The queries will never repeat, so query cache can never score a hit. Thus it should be either turned off, or have strictly limited memory, and the queries should me marked as noncacheable. On the other hand to efficiently operate on a database with millions of visitors, indexes are necessary. And these work best if keys are read from memory cache. Also performing sorting and merging operations with such a long queries rather in RAM, requires increasing of some

Table 2. Prototype performance results.

Platform capabilities ('Hello world')	1919 req/s	100%
The prototype	1761 req/s	92%
Hard Disc read (no write)	1739 req/s	91%
DBMS connection (no read/write)	1406 req/s	73%

buffers. Finally in the prototype, settings for key buffer, both of the sort buffers and the temporary table memory limit, were all greatly increased. Exact numbers will depend on the network traffic served, as well as hardware/platform and can be found with profiling tools.

All presented tweaks were made accordingly on Apache and MySQL. However, as numbers and names my be software specific, the concepts are more general. Corresponding competitive software packages usually have similar solutions, and will require similar adjustments.

3 Prototype performance

Ready built prototype was tested with stress tests, being most accurate ones. For it `ab` [1] (also know as ApacheBench) was used. Remote terminals emulation was run from second computer connected to the prototype with local network. Tests were performed with 20 000 requests for a 5619 bytes banner. The concurrency was changed to find maximum performance. Results are presented in Tab. 2.

The achieved performance is very good for the economy hardware used, but it is difficult to analyse such a result or compare it to anything. For this reasons, reference tests were performed. Firstly a 'hello world' like script was build as measure of platform maximum capabilities. It does not perform any operations, especially no reads, and only outputs same amount of data, that it has hard-coded inside. This shows that the prototype achieves 92% of platform highest possible performance. Next is a script that reads same banner from disc and outputs it. The read is actually satisfied from cache in memory. Still it achieves performance below the prototype, even without storing data in any form of stats storage. Finally script that only connects to DBMS, and then outputs hard-coded banner. It does not perform any queries to avoid a dispute on storage types or table structure. Regardless, there is clearly visible performance loss that would only increase with any additional operations.

The centralised processing is causing some delay in actuality of statistical data or user data on the ad server. There is a classic trade-off. The delay can be decreased at the cost of performance. This would demand more managing servers per ad server, allowing processing data more often. As this needs to be discussed with some numbers, please note that these will vary with regards of numerous aspects mentioned before. The claims are made on assumption that managing server has same computational capabilities as ad servers and these serve ads on maximum (peek) performance. Data from both experimental and real settings of the prototype are used.

To process statistical data every second, on average, one managing server will be needed per four ad servers. One second delay should be acceptable for enforcing virtually all of the business rules for ads display or optimisation algorithms. Oppositely, for websites offering rankings and statistics any delay measured in seconds is still way better of what is usually offered there now. Thus, one managing server can work even with as much as fifty ad servers and delay would be kept way below 100 seconds. Also strength of the architecture is that any peeks can be partially covered by increase of delay, for every use that can accept it.

The prototype is under real life testing for several months now. It provides ad serving for a website that ranks other websites, and offers them banners with statistical data. For this purpose, on mentioned earlier hardware, ad server was set together with managing server, as well as database server and website server. In this limited architecture it was serving up to 10 millions banners a day, never showing any performance problems. The managing component is processing every 120 seconds and 99.76%, of its run times are below two seconds. While running, managing component reduces ad server performance to ca. 900 req/s. Data from these tests, shows that average load is ca. half of peak load. This allows to calculate that the prototype can serve safely even 70 millions banners a day.

4 Summary

Presented benchmark data shows that the prototype is efficient and probably approaching the hardware/platform limits. Currently it is working on 10% of its possible workload, while author has no possibilities to stress it with more internet traffic than 10 millions banners a day.

There are not many limitations of presented architecture, and most of these were already discussed. The technologies used for the prototype are not such a limitation - of them operating system, HTTP server and database can be switched for almost any other. Only for PHP programming language there are used elements and technologies that would require finding a replacement, although this should not be difficult. The concept and the solution are quite general in their nature.

The last difficulty that should be mentioned is running ad optimisation algorithms in this ad server architecture. Putting heavy computations inside of an ad server component would reduce its performance. It can be reduced to the point where any remaining gains would be anyway covered by the computation time. This however creates interesting research area for ad optimisation algorithms that could be run on managing server and would work on aggregated data. By performing calculations for many users at same time, or using some caching techniques for partial or final results huge performance gains can occur.

References

1. Apache HTTP Server Documentation Project. Manual page: ab. [on-line] `http://httpd.apache.org/docs/1.3/programs/ab.html`.

2. V. Boskamp, A. Knoops, F. Frasincar, and A. Gabor. Maximizing revenue with allocation of multiple advertisements on a web banner. *Computers & Operations Research*, 2011.
3. Graham Charlton. Eight second rule for e-commerce websites now halved. [on-line] `http://econsultancy.com/uk/blog/500-eight-second-rule-for-e-commerce-websites-now-halved`, 2006.
4. Artur Ejsmont. Comparing apc and memcache as local php content cache. [on-line] `http://artur.ejsmont.org/blog/content/comparing-apc-and-memcache-as-local-php-content-cache`, 2010.
5. B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.
6. S.D. Hunter III. Pricing banner advertisements in a social network of political weblogs. *Journal of Information Technology Theory and Application (JITTA)*, 12(2):2, 2011.
7. R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *Computer*, 40(9):103–105, 2007.
8. M. Langheinrich, A. Nakamura, N. Abe, T. Kamba, and Y. Koseki. Unintrusive customization techniques for web advertising. *Computer Networks*, 31(11):1259–1272, 1999.
9. G. Lawton. Lamp lights enterprise development efforts. *Computer*, 38(9):18–20, 2005.
10. Jakub Marszałkowski. The importance of advertising exchange for marketing browser games. *Homo Ludens*, 3(1):103 – 116, 2011.
11. Daniel McLaren. Embedding base64 image data into a webpage. [on-line] `http://danielmclaren.com/node/90`, 2008.
12. Antony Dovgal Nuno Lopes Hannes Magnusson Georg Richter Damien Seguy Jakub Vrana Mehdi Achour, Friedhelm Betz. Php manual. [on-line] `http://www.php.net/manual/en/index.php`, 2010.
13. Lucas Nealan. Facebook performance caching. [on-line] `http://sizzo.org/wp/wp-content/uploads/2007/11/facebook_performance_caching-dc.pdf`, 2007.
14. Oracle and/or its affiliates. Mysql 5.1 reference manual. [on-line] `http://dev.mysql.com/doc/refman/5.1/en/index.html`, 2010.
15. Brian M. Shire. apc@facebook. [on-line] `http://www.scribd.com/doc/3288293/`, 2007.
16. Peter Snyder. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 EUUG Conference*, pages 241–248, 1990.
17. Peter Zaitsev. Cache performance comparison. [on-line] `http://www.mysqlperformanceblog.com/2006/08/09/cache-performance-comparison/`, 2006.

Online sources accessed on 30 March 2012.