Noname manuscript No.
(will be inserted by the editor)

An adaptive load balancing algorithm for cluster-based web systems

S. Kontogiannis · S. Valsamidis · P. Efraimidis · A. Karakos

S. Kontogiannis

14-4, Ektenepol, Xanthi, Greece

Tel.: +30-25410-79963

E-mail: skontog@ee.duth.gr

S. Valsamidis

16, Ag. Loukas, Kavala, Greece

Tel.: +30-2510-462370

E-mail: svalsam@ee.duth.gr

P. Efraimidis

12, Vas. Sofias, Xanthi, Greece

Tel.: +30-25410-79756

E-mail: pefraimi@ee.duth.gr

A. Karakos

12, Vas. Sofias, Xanthi, Greece

Tel.: +30-25410-79755

E-mail: karakos@ee.duth.gr

Abstract We present an adaptive load balancing algorithm for cluster-based web systems that uses dynamic weights. The new balancing policy is based on two criteria; "*HTTP process time*" and "*network delay*". The former describes web servers ability to process a forthcoming request, while the latter estimates network conditions. Periodic calculation of the two criteria is web server unaware and independent of the web server's architecture. We compare our implementation with known "*blind selection*" balancing algorithms used on web-clusters, such as: *Round Robin (RR)*,*Weighted Round Robin (WRR)* and state full ones such as *Least Connections (LC)*. We show that the previously mentioned algorithms can be outperformed by an adaptive mechanism. We confirm that the combination of the two criteria used, increase responsiveness of our algorithm towards network conditions and web server load. While existing algorithms balancing decisions depend only on web server computational load or network connections, our algorithm can distinguish between network congestion and load instances. As a result this increases the performance of the whole balancing system.

Keywords Load balancing · HTTP · Distributed web systems · Web clusters

1 Introduction

The growth of web based applications and usage of services over HTTP, lead to the building more efficient web servers and implementing different scheduling policies on requests. For example, to improve performance of the market leading Apache web server, Apache group created an hybrid multi-process

multi-threaded "*worker*" model to replace the old "*prefork*" one, that handled HTTP requests [17]. Furthermore, Zeus web server [32] uses a small number of single-threaded I/O processes each one capable of handling many simultaneous connections, like Apache *"worker"* model does.

Nowadays HTTP traffic reflects on a variety of different user applications, causing HTTP services to operate at different manners accordingly. This differentiation increases the need of web services classification performed not at the endpoint web server, but at intermediate routers [2, 9]. We take this assumption one step further; common types of HTTP traffic are:

Normal HTTP Traffic: This type of traffic is provided either by (a) static or (b) by dynamic content information, requested by clients. Dynamic content data may over-utilize web server resources than static content data do, but both categories (a) and (b) present similar characteristics from the network's point of view.

Non Congestive traffic: This type of traffic includes flows that use small packets in terms of size, whose priority is predetermined by a static"*NCQ threshold*" rate [16]. Those HTTP flows must experience minimum network delay and high priority precedence. From this type of traffic, as not mentioned by [16], TCP acknowledgments that do not belong to the aforementioned HTTP flows need to be excluded, so as not to drain the NCQ "*Less impact*" mechanism.

Maximum Throughput Traffic: Maximum throughput HTTP traffic flows contain packet sizes near network MTU and usually these flows operate

in bursts. Typical examples of this category of HTTP traffic are: *HTTP downloads*, P2P applications traffic over HTTP and huge up-link *HTTP POSTS*.

Multimedia Traffic: Multimedia Traffic [9] is HTTP or non-HTTP streaming video and audio traffic, instantiated by HTTP requests. Non-HTTP multimedia traffic is mostly carried out by UDP flows. Such kind of traffic must be taken into consideration as it can degrade web servers performance due to its persistent nature.

Secure HTTP Traffic: This type of HTTP traffic is provided by the SSL-TLS protocols through secure web services such as web commerce. This type of traffic makes intensive use of web servers CPU resources [9].

In order to further increase web server performance, distributed web architectures consisting of web server farms were introduced as a unified structure, and load balancing architectures were deployed. Distributed web architectures are classified to: *Distributed web systems*, where a number of web servers that sustain different types of web content interact with clients without the use of centralised switching. *Virtual web clusters*, where a number of web servers exchange periodically a virtual IP address that the clients use to connect to. Moreover, this is also achieved without the use of centralised switching point, and *cluster-based* web systems (or *web clusters*), where a centralized point of connection exists, called web switch [6]. We will focus on *Cluster-based* web system's architecture.

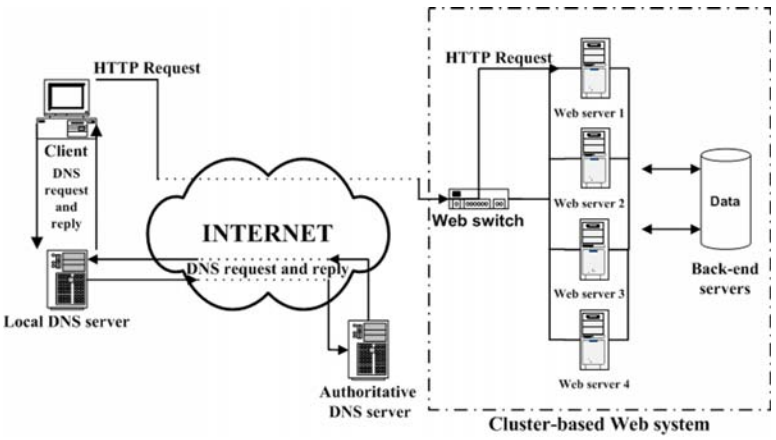

Fig. 1 The architecture of a distributed cluster-based web system.

2 Cluster-Based web systems

A *cluster-based* web system is comprised of web servers joined together as a single unity. These servers are interconnected and oppose a single system image for the outside world. A cluster-based web system is advertised with a single site name and a virtual IP address (VIP). Fig. 1, depicts the architectural structure of a cluster-based web system. The front end node of such a system is the web switch. The web switch receives all in-bound packets that clients send to the VIP address and routes them to the web server nodes accordingly. A cluster-based web system has the following structural parts; the *balancing process*, that selects the best suited target servers to respond to requests and the *routing mechanism*, that redirects clients to appropriate target servers.

Load balancing routing mechanisms of a cluster-based web system can be divided into three categories: *client-based*, *DNS-based* and *cluster-based* [7]. *Client-based* load balancing requires the client software to be aware of web

servers attributes; for example static web servers geographical locations maintained by regular client software updates. *DNS-based* load balancing is a predecessor of client load balancing and the balancing process is coordinated by DNS servers [24]. The aforementioned routing mechanisms can be used with other distributed architectures as well. *Cluster-based* routing processes are divided into application layer (layer 7) routing processes and network layer (layer 2/3) routing processes. Layer 7 balancers are proxy servers or application gateways. They redirect requests at application layer, causing requests to traverse the entire protocol stack. Layer 7 routing processes may also use the application layer for balancing decisions. Layer 2/3 routing processes redirect packets through web switches at physical or network layer, based on predefined algorithms, which are unaware of session or application layer attributes. The two basic sub-categories of cluster-based routing processes characterise web clusters architecture and differentiate on the use of content aware (layer-7) or non content aware switching (layer 2/3). There are also hybrid implementations that offer both solutions as option [4].

Content-aware routing processes examine HTTP requests at application layer. They can support sophisticated routing based on HTTP content. Their major advantage is the use of content-aware algorithms for their balancing decisions. This provides them with the ability to differentiate among types of HTTP traffic mentioned before. On the other hand, layer-7 routing policies introduce severe processing overhead at the web switch to the extent of degrading web cluster performance [25]. Moreover, there is a tendency to reduce

the impact of overhead by applying those processes at transport layer. Content aware processes that route flows on both directions, are: (1) caching clusters and gateways [26, 31], (2) TCP splicing [1, 15, 22] and for one way routing processes: (1) TCP connection hop [23] and (2) TCP hand-off [21, 29].

The routing techniques commonly used in a non content aware web switch are the following: (1) IP network address translation [27, 28], (2) Packet tunnelling, (3) Packet forwarding (also referred to as "MAC address translation") and (4) Distributed Packet Rewriting (DPR). DPR lets individual web servers to route connections to less loaded ones without the use of front end web switches and it is not used for cluster-based web systems [3]. In this paper we focus on non content aware routing policies, implemented on the web switch of a cluster based web system.

3 Load Balancing Algorithms

The load balancing algorithms of a non content aware web switch, coordinate the routing process of the clustered web system. These algorithms are dealing with a highly dynamic case of client requests, hence the objective of a load balancing system is not perfect cluster work-balance but user efficiency and fair use of resources. There are four major types of load balancing algorithms used at a web switch: ***Stateless non adaptive***, ***State full non adaptive***, ***Stateless adaptive*** and ***State full adaptive***. As ***state full-stateless*** algorithms we characterise those algorithms that keep track or not of client connection requests. As ***adaptive-non adaptive*** algorithms we characterise

those algorithms that investigate web servers status or not, through appropriate metrics.

Stateless non adaptive algorithms do not consider any kind of system state information. Typical examples of such algorithms are *Random* and *Round-Robin*. Although the *Round-Robin DNS* scheduler [24] is based on a similar principle, per host DNS caching usually leads the system to a definite state of load imbalance. In contrast, *Round Robin* implementation in virtual server [33] is more superior to *DNS Round-Robin* due to its fine per connection scheduling. Both *Random* and *Round-Robin* policies can be easily extended to treat web servers of heterogeneous capacities [6, 11]. For example if C_i is an indication of the server capacity, a relative server capacity can be defined as:

$$R_i = \frac{C_i}{max\{C_1, ..C_i, ..C_k\}} \ , \ 0 \leq R_i \leq 1 \tag{1}$$

As far as *Round-Robin* policy is concerned, different probabilities can be assigned to heterogeneous capacities. As for *Random* policy, a random generated number p, where $0 \leq p \leq 1$ can be compared with the relative server capacity in order to circulate to another server. That is: $S_{i+1} \leftarrow \ if \ \ p \leq R_i$, else: S_{i+2}.

State full non adaptive algorithms keep track of client requests. The classic *Weighted Round Robin* (*WRR*) can also be used as a stateless balancing algorithm on web servers with different processing capacities. Each server is assigned a static integer weight which corresponds to an index of its own capacity: $W_i = \frac{C_i}{min\{C\}}$, where $min\{C\}$ is the minimum servers capacity. *WRR* serves connection to server i, up to its weighted capacity W_i in a first-come first serve manner. So, the maximum period of the *Round-Robin* cycle

equals to: $\sum_{i=1}^{n} W_i$. *List based WRR (LWRR)* is an extension of the classical *WRR* algorithm. *LWRR* introduces fairness, by multiplexing aggregated client requests to the cluster's web servers. Moreover, a fair *LWRR* implementation was proposed by [33] and implemented by [20, 34].

The *Least Connections - Weighted Least Connections (LW-WLC)* algorithm directs network connections to the web server with the least number of established connections, while the *Weighted Least Connections* algorithm (*WLC*), used in many commercial systems [10], assigns a new connection to the web server k, that: $\frac{C_k}{W_k} = \min\{\frac{C_i}{W_i}\}$, where $i = 1, .., n$.

The *Shortest Expected Delay* (*SED*) scheduling algorithm [30] is similar to the *WLC* algorithm and assigns client connections to the web server with the shortest expected delay. That is: $E_i = \frac{C_i + 1}{U_i}$, where C_i is the number of connections and U_i is a fixed service rate weight in terms of maximum throughput that the web servers can handle. A modification of *SED* algorithm, the *Never Queue* scheduling algorithm [30] balances HTTP flows using the following two criteria: (a) if there is an idle server, meaning: $C_i < \min\{U_i\}$ then assign connections to the idle server or (b) if there is no idle server, use the *Shortest Expected Delay* schema.

Destination hashing locality based scheduling algorithm [8], assigns network connections based on a statically assigned hash table of destination IP addresses. This algorithm uses flow state information in order to prevent flow service delivery. That is, if S_i is overloaded meaning: $C_i > 2W_i$, then the connections from that point and forward are dropped. Similarly, *Source hashing*

locality based scheduling algorithms assigns client requests from a statically assigned hash table of source IP addresses, maintained at the web switch.

Stateless Adaptive balancing algorithms take into account server state conditions but do not keep track of connection state information. These algorithms use monitor agents running on either the web switch or web servers. The information retrieved from agent look ups is taken into account in order to determine metric values and therefore weights used for balancing decisions. Some commonly used metrics are: "*cpu load*", "*memory usage*", "*disk usage*", "*current process number*" and "*ICMP request-reply*" time. Web servers metric values are usually stored by network monitor services like *SNMP*, that run on the web switch.

The *Fastest Response Time* algorithm, assigns new connections to the web server that responds faster. The basic criteria used is "*ICMP request-reply*" time or "*HTTP response time*". That is the amount of time needed for small size web server objects to be downloaded by the web switch agent.

State full Adaptive use both adaptive algorithms metrics and client flows state information, in terms of: Number of connections or ratio of connections that a server has received to the average connections received at a specific time interval $t_2 - t_1$: $R_i = \frac{|C_{i2} - C_{i1}|}{\frac{1}{n}\sum_{i=1}^{n} C_i}$ [20,34], source or destination IP address (locality based [8]).

The *Least Loaded* algorithm uses an adaptive *WRR* schema with weights that depend on a *SNMP* advertised aggregation load metric, or other SNMP parsed attributes [5]. Other implementations use synchronised UDP connec-

tions between web servers and the web switch where load messages arrive at a predefined rate [2]. Load messages-metrics are an aggregation of several server attributes into one load value. Calculation of such metrics uses the following formula: $AGG_{load_i} = \sum_{i=1}^{n} K_i \cdot Server\_metric_i$ and $W_i = W_{initial_i} \cdot (1 - \frac{AGG_{load_i}}{\sum i = 1^n AGG_{load_i}})$. Furthermore, another formula used at the linux kernel load balancer to calculate web server weights from the aggregate load metric, is the following: $W_i = W_{initial_i} \cdot \sqrt[3]{1 - AGG_{load_i}}$.

4 The ALBL Algorithm

We implement a stateless adaptive load balancing algorithm, called *Anonymous Load BaLancer* (*ALBL*), which estimates both network conditions and web servers load in order to make balancing decisions. Preliminaries of this work was presented at [13].

In particular, two basic criteria-metrics are used to dynamically adjust web server weights; These metrics are: "*HTTP response time*" and "*network delay*". Metric calculation is performed by agents that run at the web switch and those agents are unaware of web sever conditions. Based on those metrics, weight calculation occurs. The whole process is repeated periodically. Initial weight values and metric calculation period are given by the administrator, as a primer estimation of the system's balancing point and system's responsiveness to web server conditions. Then a *WRR* process periodically occurs among the web servers, until the metrics of *HTTP response time* and *network delay* for each server are updated and adaptive weight calculation of web server weights

for the following period is complete. Web pages with real-time graphs of web servers metric values per time interval are also kept at the web switch, for visualisation purposes. *WRR* algorithm runs on each period using a random process that shuffles web server selection in order to distribute resources fairly among client requests. Assume a cluster of three web servers S_1, S_2, S_3. The normalised web server weights 0.3, 0.5 and 0.2, in one round of the balancing period. We also assume that these weights remain the same all balancing periods. In this case, the web-switch *WRR* selection process will randomly allocate each one of the web servers, for example: S_1, S_2, S_3 for the first period, S_3, S_1, S_2 for the second and so forth. This random selection process is used in order to prevent synchronised HTTP requests from receiving best or worst service so as to further improve fairness of the balancing system towards client requests and avoid synchronisation effects.

In depth, the balancing algorithm is as follows; the load balancer sends an HTTP request for a predefined object, to each web server and waits for a reply. The application estimates the time that the request was sent and the time that the reply was received. The time difference between request and reply, corresponds to the *HTTP response time* metric. If the server does not respond within a fixed timeout interval, then the application marks it as being down and excludes it from receiving further requests during that period. The *HTTP response time* metric is equal to the sum of network propagation delay, network queuing delay and web server processing delay:

$$HTTP_{resp} = Q_{Prop} + Q_n + Q_{Proc} \tag{2}$$

where *Propagation delay* Q_{Prop}, is the time required for a signal to travel from one point to another and is assumed equal for all web servers. *Queuing delay* Q_n in a packet-switched network, is the sum of all the delays encountered by a packet from the time of its insertion into the network until the delivery to its destination *Processing delay* Q_{Proc} of a web server, is the time needed for a process to perform a request and construct an appropriate reply message. Processing delay depends on web server CPU load index.

In order to estimate network delay, a *network delay* metric is used as follows; a minimum size TCP "SYN" packet is constructed, with source the IP address of the web switch and destination the IP address of each one of the web servers. An appropriate timeout is set for "SYN" packet transmission. The timeout value is set to 25ms, equal to web-switch *OS* clock granularity and small enough so as to recognise as excessive delays requests that timeout. The timeout value can be altered according to the network topology and conditions of the system. If timeout occurs then that web server is removed from the balancing process until the next agent look up period. The agent responsible for the *network delay* metric calculation sends the packet and awaits for an acknowledgment. Then it calculates the response time between TCP "SYN" request and "SYN-ACK" reply. This response time is an index of the network delay between the web server and intermediate routers; an approximation of the propagation and queuing delay of the network path between the web-switch and the web server:

$$N_{delay} = Q_{Prop} + Q_n \tag{3}$$

We assume constant propagation delay for all web server links, therefore we subtract Q_{Prop} from N_{delay} metric (Equation 3). We also assume equal rate in terms of throughput for *HTTP response time* and *network delay* agents per web server. Then the time difference between *HTTP response time* bandwidth delay product and *network delay time* bandwidth delay product equals to the processing delay of the web server, that is given from the following equation;

$$HTTP_{resp} - kN_{delay} = R_H dT_H + T_{pr} - k(R_n dt_n) \rightarrow T_{pr} \tag{4}$$

where R_H is the average throughput of the HTTP flow used for calculating *HTTP response time* metric, T_H is the time needed for known size object to be downloaded by the aforementioned HTTP flow, T_{pr} is the web server's processing time for the HTTP request, $R_n t_n$ is the bandwidth delay product of the *network delay* metric ($R_H = R_n$) and k equals to the ratio of *HTTP response* time agent object size to *ndelay* agent object size: $\frac{Obj_H}{obj_n}$ in bytes.

In order to discriminate among web servers CPU load periods and congestion periods, a *threshold value* is used by the algorithm as follows; if at least one web server has *HTTP response time* less than the fixed threshold value, the algorithm concludes that there is no network congestion and the weights for each web server are calculated using the product of *HTTP response time* with *network delay* metric, as in Equation 5. If all web servers have *HTTP response time* metric value greater than the *threshold value*, then network congestion is taken into account and the weights are calculated from Equation 6 using the product of process time of each server, calculated from Equation 4, with the network delay metric. The reason for using the product instead of

the sum is to increase responsiveness of the load balancing system towards network conditions. Moreover, to further increase responsiveness towards network conditions, processing delay estimation and *network delay* metric values are used instead of the *HTTP response time* metric value.

$$\mathbf{Weight}_i = \frac{\frac{\mathbf{1}}{\mathbf{http\_resp}_i \cdot \mathbf{ndelay}_i}}{\sum_{i=1}^{\mathbf{n}} \frac{\mathbf{1}}{\mathbf{http\_resp}_i}} \tag{5}$$

In addition, if none of the servers are up for more than three sequential time periods, then the administrator is notified.

Since *network delay* metric values are smaller than *HTTP response time* metric values, Equation 5 does not provide adequate responsiveness towards network conditions, but can fairly estimate load conditions. In Equation 6 the processing delay time for each web server is extracted from the *HTTP response time* metric. Then the product of processing delay time with the *network delay* metric, is used for weight calculation. The weight value from Equation 6 is more responsive to network delay variations and can spot congestion incidents, because processing time is closer to *network delay* metric than *HTTP response time* metric is. That indicates that Equation 6 is appropriate for conditions where the web server has no computational load and network delays are of importance.

$$\mathbf{Weight}_i = \frac{\frac{\mathbf{1}}{\mathbf{ndelay}_i \cdot (\mathbf{http\_resp}_i - \mathbf{ndelay}_i)}}{\sum_{i=1}^{\mathbf{n}} \frac{\mathbf{1}}{\mathbf{delay}_i \cdot (\mathbf{http\_resp}_i - \mathbf{ndelay}_i)}} \tag{6}$$

After the weight calculation, the server that will receive requests is randomly selected per time period. Finally, a linux kernel module uses the cal-

culated weights in order to redirect using NAT and a *WRR* scheduler, the forthcoming requests to the appropriate web server.

5 Experimental Evaluation

In our experimental scenarios, we used a web cluster with web servers of equivalent processing power connected to a 100Mbit web switch, as depicted in Fig. 2. The web servers use Apache software, version 2.2 and the default "prefork model" as processing model for HTTP requests. We performed our cluster tests with Apachebench, a set of Perl scripts designed to support the automation of benchmarking with *httperf* tool [18].

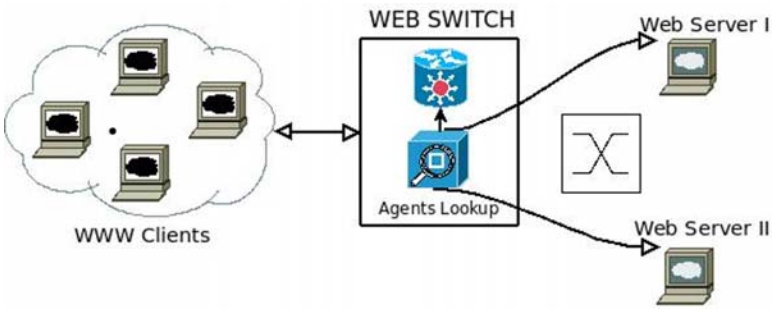

Fig. 2 Experimental scenario architecture.

The web-switch is a Pentium 4 computer with a CPU that operates at 2GHz and 512Mb of memory. The operating system that runs on the web-switch is a Linux operating system. Our *ALBL* algorithm uses Linux netfilter [19] for the routing process and a set of Perl scripts for the collection of metric

values. Communication between agents and kernel routing process is done through the Linux proc file-system.

Each experimental scenario consists of clients that send HTTP requests to the web servers. These generated requests are evenly distributed among nodes of the client cloud, aiming the web switch. The submission rate of the requests varies for every experiment accordingly. For every scenario, a fix set of request rates per second is executed until a determined number of requests is reached and for each rate performance metrics of the web cluster system are measured.

The performance metrics that are used are the following:

- *Throughput*, in terms of response rate.
- *HTTP response time*, is defined as the time from the initial HTTP request being sent until the complete HTTP response is received (in ms). In addition, in our scenarios, *HTTP response time* is expressed as an average value over all client HTTP requests per request rate.
- *Error Rate*, is the percentage of the requests that were not serviced or delayed service more than 10 seconds. *Error rate* is an important performance metric of the balancing algorithm. More specifically, as errors increase HTTP requests that failed service increase. This is an indication of sub-optimal selections of the balancing algorithm that lead HTTP requests to timeout.

In our experimental scenarios we investigate the following balancing algorithms: Stateless non adaptive *Round Robin* (*RR*) and state full non adaptive *Least Connections* (*LC*), implemented at the linux kernel [20]. We compare

those results with measurements from our stateless adaptive *ALBL* implementation in three separate scenarios:

I. The web servers have no initial load and the clients retrieve an object from the cluster. This scenario resembles the case of normal network conditions and lack of web servers load.

II. The web servers have no initial load, but the network link of one of the web servers is congested. This is an attempt to test the aforementioned balancing algorithms under utilized network resources.

III. The first web server's CPU is loaded, as an attempt to test in what extent load balancing algorithms will follow CPU utilization, when adequate network resources exist.

5.1 Scenario I

In this scenario HTTP client requests retrieve 1Kb object from the web cluster. This operation is initially performed at a rate of 20 HTTP requests per second until 2000 HTTP requests per second (actually 1970 requests/sec), with a step of 50 requests per second. The algorithms that are put to test are: *RR*, *LC* and our *ALBL* implementation. The purpose of this scenario is to compare the performance of the stateless *RR* and of the state full *LC* balancing algorithms with the performance of the *ALBL* balancing algorithm.

The results of this scenario in terms of throughput (Fig. 3), show that under normal conditions all algorithms perform almost equivalently. This was something expected for this scenario since none of the balancing mechanisms

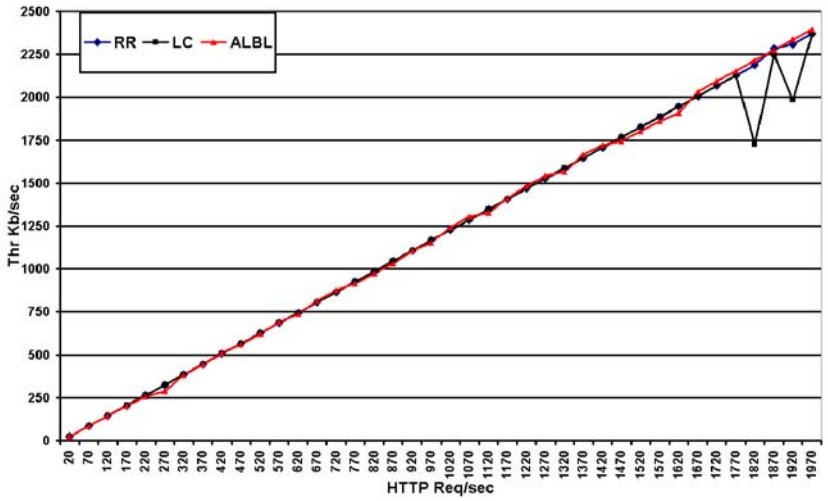

Fig. 3 Scenario I. Throughput Kb/sec over clients Req/sec

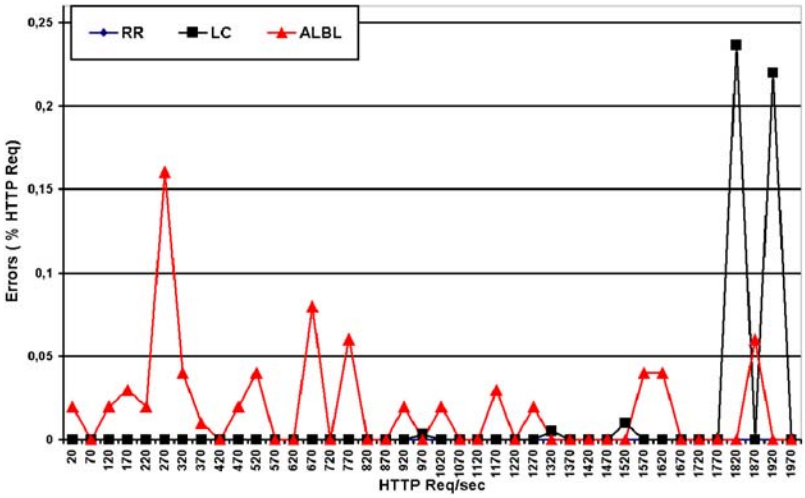

Fig. 4 Scenario I. % Error connections from total number of connections over clients Req/sec

of *LC* or *ALBL* are actually used. This is the case where a "blind mechanism" such as *RR* outperforms "intelligent mechanisms" (*LC, ALBL*), that from

their part contribute only to consumption of web switch CPU and network resources. That conclusion can be verified from the small number of errors in Fig. 4 of the *LC* and *ALBL* algorithms. The drawbacks of web switch resource consumption become visible when clients request rate increases. The *LC* algorithm presents less throughput (Fig. 3) than *RR* and more errors in terms of error rate, while *RR* errors are essentially zero (Fig. 4). Moreover the *ALBL* algorithm that consumes network resources is more prone to errors than the *LC* algorithm.

The conclusion of scenario I is that simple algorithms like *RR* are sufficient as long as web servers are of the same processing power and have equivalent network resources at their disposal. Furthermore, HTTP requests of small HTTP response content in bytes, can be efficiently balanced over a web cluster of uniformly distributed servers, with the use of a simple balancing algorithm like *RR*.

5.2 Scenario II

In this scenario HTTP client requests retrieve 10Kb object from the web cluster. This operation is initially performed at rate of 10 HTTP requests per second until 120 HTTP requests per second, with a step of 10 requests per second. For each specific rate value the experiment lasts 360 seconds and the total number of connections increases as the rate increases.

The algorithms that were put to test are: *RR*, *LC* and our *ALBL* implementation. The purpose of this scenario is to compare state full, stateless

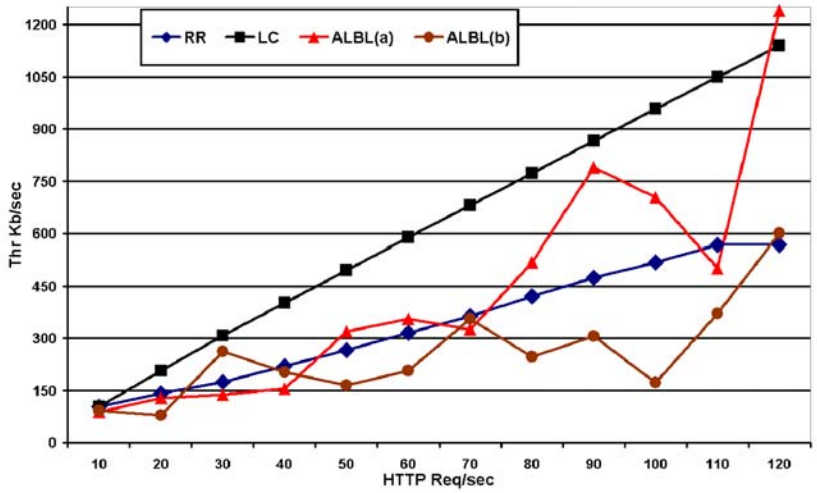

Fig. 5 Scenario II. Throughput Kb/sec over clients Req/sec

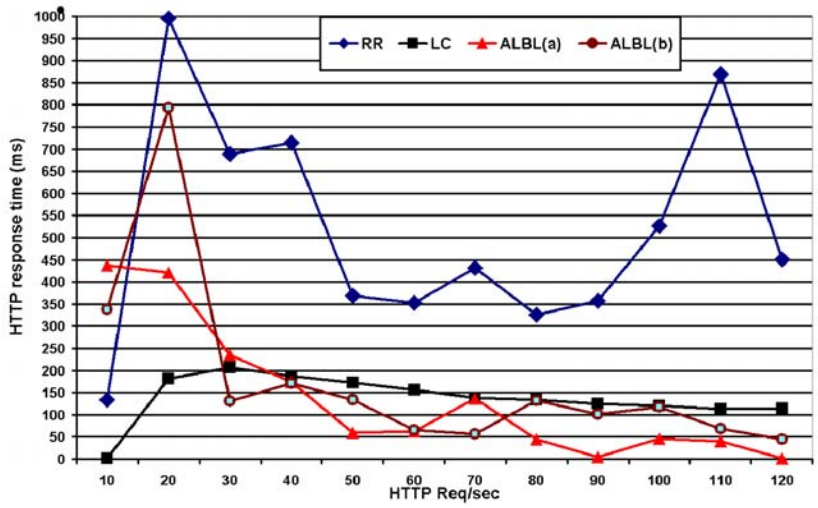

Fig. 6 Scenario II. Average HTTP response time over clients Req/sec

and adaptive balancing mechanisms under limited network resources. In order to achieve congested network conditions, we shape the traffic that passes from the first web server using a Token Bucket Filter, the *TBF* queuing disci-

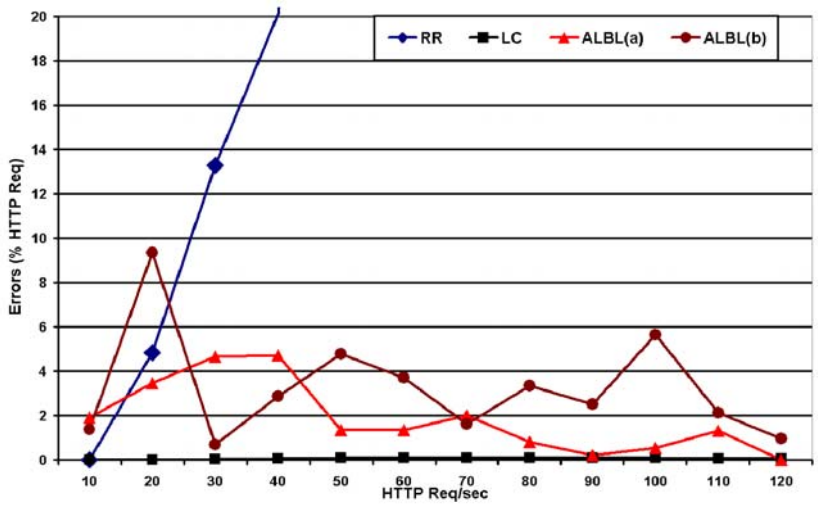

Fig. 7 Scenario II. % Error connections from total number of connections over clients Req/sec

pline [12, 14]. This queuing discipline limits the bandwidth of that web server to a rate of 0.5 Mbit/s with a queue size expressed in 10ms latency and a burst size of 2Kbytes. The other web server is connected to a 100Mbit network. We also decreased client timeouts value to 5 seconds. In order to optimise the performance of the web servers we activated Apache's caching ability. This way we assure that the bottleneck will be the network.

We test the *ALBL* algorithm with two different threshold values: (a) a small *threshold* value that causes *network delay* metric to be of importance on the weight calculation (use of Equation 6) and (b) a high *threshold* value that causes the *HTTP response time* to be the key metric (from Equation 5).

The congestion conditions on this experiment cause HTTP response rate to fluctuate, while HTTP request are submitted with a constant rate. This is

due to packet drops and excessive packet delays on the congested web server. The *LC* algorithm in this scenario, achieves the best results in terms of performance. That is minimum errors, minimum response time and maximum throughput. *ALBL(a)* algorithm performs close to that of the *LC* algorithm performance in terms of HTTP response time and some times even outperforms *LC* in terms of throughput. From Fig. 5, 6 and 7 it is obvious that the adaptive mechanism of *ALBL(a)*, which takes into account the *network delay* metric, performs better than *ALBL(b)*.

There is a potential drawback in *LC* when a large number of inactive HTTP requests exists, the *LC* algorithm interprets all pending requests active and inactive as potential load. This behaviour leads to non-optional balancing decisions. The TCP's TIME_WAIT status of inactivity is usually large enough (1-2 minutes) until a connection terminates. In this time interval thousands of new connections may request service. Furthermore, problematic behaviour of the *LC* algorithm leaves plenty of space for *ALBL* and generally adaptive algorithms to outperform it, even if they are more resource consuming than the *LC* algorithm is [1].

Even though *ALBL(a)* performs generally well, we noticed during the experiment that some unexpected errors occur. In certain cases network conditions change more frequent than the *ALBL* metric update period. Taking balancing decisions based on erroneous information caused by fast congestion

[1] This is a known drawback of the *LC* algorithm and the implementation uses a trick that mitigates the consequences of the drawback. This trick is to multiply the active connections with a fixed factor in order to separate them from the inactive ones

incidents is the main reason for the fluctuation of *ALBL(a)* in Fig. 5. HTTP errors of the *ALBL* algorithm (Fig. 7), apart from web switch load increase, are caused also due to the algorithm's design to probe the network at time intervals. Some congestion incidents in this scenario are fast enough and the *ALBL* algorithm cannot follow them. However if we modify probing intervals to be very small, near web switch operating system clock granularity, then the web switch CPU will become overloaded no matter if congestion incidents occur or not. Furthermore, the above problem of the *ALBL* algorithm can be mitigated by making the measurement period adaptive during congestion intervals. We consider this a future work.

From Fig. 5, *RR* achieves low throughput. Fig. 6 shows fluctuation of HTTP response time while Fig. 7 shows a increase of error rate up to 78%. Sporadic improvements of *RR* algorithm HTTP requests response time (Fig. 6) is because HTTP response time measurement is more or less calculated by a small number of client connections, since error rate increases exponentially and HTTP connection timeouts are not used at the average HTTP response time calculation (Fig. 7).

5.3 Scenario III

In this scenario the CPU of one of the web servers is overloaded, while HTTP clients retrieve 1Kb object from the web cluster. This operation is initially performed at a rate of 10 HTTP requests per second, with a step of 10 requests per second, until 120 HTTP requests per second.

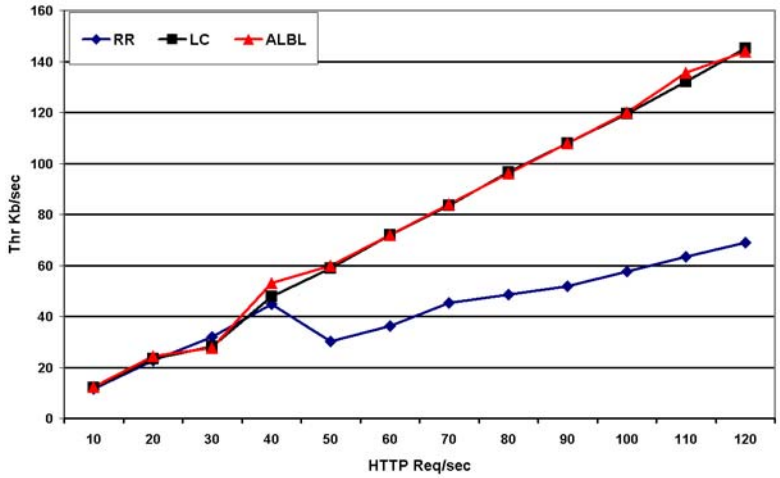

Fig. 8 Scenario III. Throughput Kb/sec over clients Req/sec

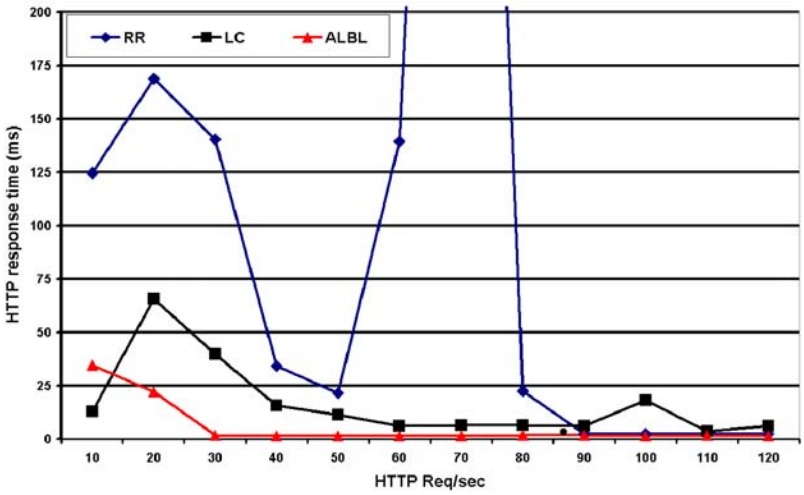

Fig. 9 Scenario III. Average HTTP response time over clients Req/sec

The purpose of this scenario is to compare the performance of the stateless *RR* and the state full *LC* balancing algorithms with the *ALBL* algorithm, under severe unbalancing conditions due to CPU overloading of one of the web

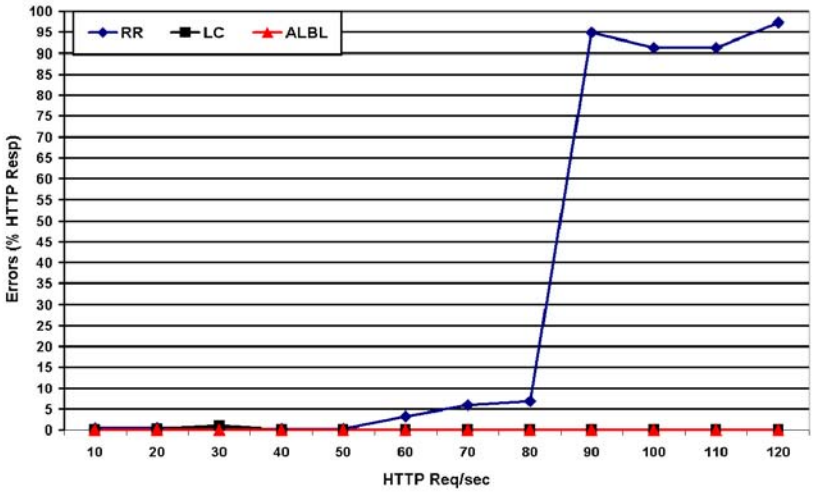

Fig. 10 Scenario III. % Error connections from total number of connections over clients Req/sec

servers, while adequate network resources exist. To achieve CPU overloading of the first web server, a CPU intensive script is used at the web server, while an agent instantiated from the web switch retrieves that script at constant time intervals. *ALBL* implementation uses a high *threshold*, which takes into account only the *HTTP response time* metric.

As expected, *LC* and *ALBL* algorithms outperform *RR* in terms of throughput (Fig. 8), but now we can view more clearly the advantages of an adaptive mechanism such as *ALBL*. In Fig. 9, *ALBL* algorithm selection causes HTTP flows to maintain better average HTTP response time than *LC* algorithm, while (Fig. 10), errors are kept close to zero.

The *RR* algorithm initially increases its response time, but as connections increase, its response time decreases rapidly due to errors (and timeouts) rapid

increase. From rate 90 and above all requests timeout (Fig. 10). This large number of errors causes HTTP response time to decrease rapidly, since HTTP timeouts are not used for the calculation of the average HTTP response time. The LC behaves similarly as to scenario II. Furthermore, the $ALBL$ algorithm successfully spots load conditions, due to their longer time intervals and thus manages to perform better that the LC algorithm.

6 Conclusions

In this paper, we present the $ALBL$ balancing algorithm for cluster-based web systems, which takes into account web servers CPU load as well as network conditions for the balancing process. We compare our algorithm with known stateless and state full algorithms. We prove from a number of experimental scenarios that $ALBL$ matches or even overcomes the performance of those algorithms. In particular $ALBL$ balances efficiently HTTP traffic, on unbalanced conditions that dynamically change: (a) Due to utilized network conditions and (b) due to web servers limited computational resources, while adequate network resources exist. We also pinpoint the significance of adaptive balancing strategies over state full ones, as well as state full ones over stateless ones. We also leave for further investigation an adaptive mechanism of the threshold value that will improve the discrimination between congestion incidents and web server load conditions in order for the metric update period to adapt accordingly.

There is a trade-off between stateless and state full, adaptive algorithms. That is the complexity of state full and moreover adaptive algorithms towards simplicity and lack of network or CPU provisions of stateless ones. Complexity affects balancing, causing system's performance to degrade while simplicity may operate in favour of imbalanced conditions. Since the web switch is the weakest point of a web cluster, in order to reduce resource consumption of adaptive algorithms, there is a tendency for next generation sophisticated balancing algorithms to be implemented on separate Network Management Systems (NMS). Those systems inform the web switch for network or load changes, while the web switch is responsible only for the routing process.

References

1. Adhya, S. K., and Ganguly, S. Asymmetric tcp splice: A kernel mechanism to increase the flexibility of tcp splice, master thesis. `http://www.cse.iitk.ac.in/research/mtech1999/9911134.ps.gz`, 2001.
2. Andreolini, M., Casalicchio, E., Colajanni, M., and Mambeli, M. A cluster-based web system providing differentiated and guaranteed services. *Cluster Computing vol. 7-1* (2004), 7–19.
3. Aversa, L., and Bestavros, A. Load balancing a cluster of web servers using distributed packet rewriting. In *In Proc. of IEEE International Perfofmance, Computing and Communications conference* (2000), pp. 24–29.
4. Balasubramanian, J., Schmidt, D. C., Dowdy, L., and Othman, O. Evaluating the performance of middleware load balancing strategies. In *In Proc. of IEEE International Conference of Enterprise Distributed Object Computing* (2004), pp. 135–146.
5. Batheja, J., and Parashar, M. A framework for adaptive cluster computing using javaspaces. *Cluster Computing vol. 6-3* (2003), 201–213.

6. Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. S. The state of the art in locally distributed web-server systems. *ACM Computing Surveys vol. 34-2* (2002), 263–311.

7. Cardellini, V., Colajanni, M., and Yu, P. S. Dynamic load balancing on web server systems. *IEEE Internet Computing vol. 3-3* (1999), 28–39.

8. Cardellini, V., Colajanni, M., and Yu, P. S. Geographic load balancing for scalable distributed web systems. In *In Proc. of 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2000), pp. 20–28.

9. Casalicchio, E., and Colajanni, M. A client-aware dispatching algorithm for web clusters providing multiple services. *WWW ACM* (2001), 535–544.

10. Cisco Inc. Cisco Distributed Director. `http://www.cisco.com/warp/public/cc/pd/cxsr/dd/`, 2004.

11. Colajanni, M., Yu, P. S., and Dias, M. D. Analysis of task assignment policies in scalable distributed web-server systems. *IEEE Trans. on Parallel Distributed Systems vol. 9-6* (1998), 585–597.

12. Hubert, B., Maxwell, G., Mook, R. V., Oosterhout, M. V., Schroeder, P., and Spaans, J. Linux advanced routing and traffic control howto. `http://www.tldp.org/HOWTO/Adv-Routing-HOWTO`, 2002.

13. Karakos, A., Patsas, D., Bornea, A., and Kontogiannis, S. Balancing HTTP traffic using dynamically updated weights, an implementation aproach. In *In Proc. of 10th Panhellenic Conference on Informatics* (2005), pp. 873–878.

14. Maga, E., Izkue, E., and Villadangos, J. Review of traffic scheduler features on general purpose platforms. In *In Proc. of SIGCOMM LA '01: Workshop on Data communication in Latin America and the Caribbean* (2001), pp. 50–79.

15. Maltz, D., and Bhagwat, P. Application layer proxy performance using tcp splice. RFC 21139, 1998.

16. Mamatas, L., and Tsaoussidis, V. A new approach to service differentiation: Non-congestive queueing. In *In Proc. of International Workshop on Convergence of Heterogeneous Wireless Networks* (2005), pp. 78–83.

17. Menasce, D. A. Web server software architectures. *IEEE Internet Computing vol. 7-6* (2003), 78–81.

18. Mosberger, D., and Jin, T. A tool for measuring web server performance. In *In Proc. of ACM Workshop on Internet Server Performance* (1998), pp. 59–67.

19. Netfilter Team. Linux netfilter project. `http://www.netfilter.org`.

20. O'Rourke, P., and Keefe, M. Performance Evaluation of Linux Virtual Server. In *In Proc. of 15th System Administration Conference, LISA* (2001), pp. 79–92.

21. Pai, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. Locality-aware request distribution in cluster-based network servers. *ACM SIGPLAN Notices vol. 33-11* (1998), 205–216.

22. Purkayastha, S. K., and Ganguly, S. Symmetric tcp splice: A kernel mechanism for high performance relaying. Tech. rep., Dept. of C.S. Indian Institute of Technology, 2001.

23. RESONATE Team. Tcp Connection Hop. *White Paper* (April 2001).

24. Schemers, R. J. ldnamed: A load balancing name server in perl. In *In Proc. of 9th USENIX conference on System administration* (1995), pp. 203–210.

25. Song, J., Abegnoli, E. L., and Dias, M. D. Design alternatives for scalable web server accelerators. In *In Proc. of IEEE International Symposium on Performance Analysis of Systems and Software* (2000), pp. 184–192.

26. SQUID Development team. SQUID web proxy cache. `http://www.squid-cache.org`.

27. Srisuresh, P., and Egevang, K. Traditional IP Network Address Translation. RFC 3022, 2001.

28. Srisuresh, P., and Gan, D. S. Load sharing using IP Network Address Translation. RFC 2391, 1999.

29. Wang, L. Design and implementation of TCPHA - Draft release. `http://dragon.linux-vs.org/dragonfly/`, 2006.

30. Weinrib, A., and Shenker, S. Greed is not enough: Adaptive load sharing in large heterogeneous systems. In *In Proc. of IEEE INFOCOM'88* (1988), pp. 986–994.

31. Wessels, D., and Claffy, K. Internet Cache Protocol (ICP) version 2. RFC 2186, 1997.

32. Zeus Development Team. Zeus web server. `http://www.zeus.com.uk`, 2002.

33. Zhang, W. Linux server clusters for scalable network services. In *In Proc. of Ottawa Linux Symposium* (2000), pp. 437–456.

34. Zhang, W. Build highly-scalable and highly-available network services at low cost. *Linux Magazine* (November 2003).


Mr. Sotirios Kontogiannis is a PhD candidate student at Dept of Electrical and Computer Eng., Democritus University of Thrace, Xanthi, Greece. He received a five-year Eng. diploma and MSc in Software Eng. from Department of Electrical and Computer Eng., Democritus University of Thrace. His research interests are in the areas of Distributed systems, computer networks, middlware applications and computer network modelling and performance evaluation. His e-mail is: skontog@ee.duth.gr and web page: `http://utopia.duth.gr/~skontog`.


Mr. Stavros Valsamidis is a PhD candidate student at Dept of Electrical and Computer Eng., Democritus University of Thrace, Xanthi, Greece. He received a five-year Electrical Eng. diploma from Department of Electrical Eng., University of Thessaloniki, Greece and MSc in Computer Science from University of London, UK. He is an Applications Prof. in the Dept. of Information Technology, TEI of Kavalas, Greece. His research interests are in the areas of Distributed systems, computer networks, database architectures, data analysis and data mining. His e-mail is:
svalsam@ee.duth.gr.


Dr. Pavlos S. Efraimidis holds a permanent position of a Lecturer at the Democritus University of

Thrace (Greece). He graduated from the Dept. of Computer Engineering and Informatics of the University of Patras (Greece) in 1995 and obtained a PhD in Informatics in 2000 from the University of Patras under the supervision of Paul Spirakis. His research interests are in the field of algorithms. He is a member of ACM and IEEE. His e-mail is: pefraimi@ee.duth.gr.


Dr. Alexandros Karakos received the Degree of Mathematician from the Department of Mathematics (Aristoteles University of Thessaloniki, Greece) in 1972 and the Maitrise d' Informatique from the University PIERRE ET MA RIE CURIE - PARIS at 1977. The Ph.D. received from University PIERRE ET MA RIE in 1978.He is Associate Professor at the Department of Electrical and Computer Engineering, Democritus University of Thrace, Greece. His research interests are in the areas of Distributed systems, data analysis and programming languages. His e-mail is: karakos@ee.duth.gr and web page: `http://utopia.duth.gr/~karakos`.