PEGASUS: Competitive load balancing using inetd

George Oikonomou[1] Vassilios Karakoidas[2]
Theodoros Apostolopoulos[1]

[1]Department of Informatics
Athens University of Economics and Business
{g.oikonomou,tca}@aueb.gr

[2]Department of Management Science and Technology
Athens University of Economics and Business
bkarak@aueb.gr

1 Introduction

As it is proven by practice, load balancing techniques are the only tool that network service provider have, in order to support and handle scalable network load. This paper presents PEGASUS, a novel framework that provides load balancing in network services transparently, using a competitive scheduling algorithm. PEGASUS is designed upon the INETD superserver, thus providing a easy to configure, yet efficient infrastructure. The paper is structured as follows. In section 2 we provide an overview of the current frameworks and popular implementations both open source and commercial. Then we describe our system architecture and the basic axioms that it is designed. The paper concludes with a benchmarking experiment of our prototype implementation, in comparison with other popular load balancing frameworks.

2 Related Work

In the past few years, many load balancing techniques were developed to address the problem of distributing incoming service requests in a fair fashion to server clusters. In this section we will describe the most typical solutions used in production environments.

2.1 Round Robin DNS

One the most widely used load balancing techniques is the Round Robin DNS [4]. The nodes offering content are mapped to the Name Server using the same domain name. The name server responds with a different IP address for each DNS request referring to the same name. This happens in a simple round robin manner. This scheme provides high flexibility and full transparency to the end user. On the other hand, RRDNS is problematic when dealing with node failures. When a node fails, the name server will not cease to direct clients to it, so without being reconfigured and thus a percentage of requests will reach a server that is down. Furthermore, due to the caching nature of DNS resolvers (clients) and the hierarchical approach of DNS, the RRDNS system can lead to load imbalance. This can be avoided by the correct selection of the Time To Live (TTL) value of DNS records. In practice, the correct choice of TTL is not trivial. A low value will result in many requests to the Name Server, therefore making it the bottleneck. In contrast, a high value will fail to resolve the load imbalance issue.

2.2 The Linux Virtual Server Project

The Linux Virtual Server (LVS) [11, 9] project aims to help create a scalable, highly available and robust virtual server. The architecture is based on a set of nodes that actually offering services and a load balancer node. The load balancer accepts all incoming requests from users. Based on a variety of load balancing schemes, it selects a server that will actually handle the request and forwards the request.

The LVS architecture is fully transparent from the end user perspective. No modifications are required on the client side of the applications and users interact with the cluster in the same manner as if it were comprised of a single server. A heartbeat scheme is implemented in order to detect daemon failures and provide for transparent reconfiguration of the system.

The LVS load balancer uses a variety of scheduling algorithms to select the node that will serve the requests like Round-Robin Scheduling, Weighted Round-Robin Scheduling, Least-Connection Scheduling among others [11]. A detailed analysis of each of those scheduling algorithms is out of scope of this paper.

The LVS uses three load balancing techniques. These are:

1. LVS via NAT (LVS/NAT)
2. LVS via IP Tunneling (LVS/TUN)
3. LVS via Direct Routing (LVS/DR)

In order for these techniques to be supported, a modified TCP/IP stack is required. Furthermore, these techniques are all based on the assumption that all real servers offer identical services, having identical content. The latter is achieved either by replicating the content on the local storage of the real servers or by giving them access to a shared or distributed file system. Finally, the LVS load balancer and the real servers need specific configuration in order to become part of the system. The biggest advantage of LVS is that it operates purely on kernel space, making the overhead the smallest possible. It's biggest disadvantage is that it requires Linux operating system to execute.

2.3 mod_backhand

The mod_backhand project [8] is an attempt to provide a load balancing mechanism, that features request redirection for the HTTP protocol. It is implemented as an APACHE web server module and it is written in C++. Its architecture is quite simple, the balancer consists of a classic APACHE server, where the mod_backhand module is initialized. The information for the actual servers is gathered on the mod_backhand information manager, where the statistics are processed by the scheduler. The main benefit of mod_backhand is that it provides redirections on HTTP sessions, preserving client information about access and authentication. The main drawback of mod_backhand is that is a balancing mechanism only for HTTP. Mod_backhand is available for numerous operating systems, such as Linux, Microsoft Windows, FreeBSD and Solaris.

2.4 Other Approaches

A few other approaches have been proposed as a solution to the load balancing problem. Among them, those we consider to be most significant are NETDISPATCHER [1], REVERSE PROXY [6], SWEB [2] and EDDIE [7]. Another significant solution is the Cisco Local Director [5], which is a commercial, hardware based solution. All these approaches have their pros and cons,

3 Competitive Loab Balancing

Our proposal is the PEGASUS architecture, an approach that provides a simple, yet efficient, service-transparent competitive load balancing scheme.

3.1 Architecture

The PEGASUS system architecture is illustrated in Figure 1. The system is comprised of two distinct entities, typical for load balancing frameworks.

Balancer The balancer is responsible for the request redirection to the service providers. Each balancer has a FIFO queue that holds tickets. Tickets are service availability notifications that the service providers sent, in order to notify the balancer about their presence, and their service capacity. The ticket queue is maintained by a daemon, named PEGASUS redirector. In our prototype the balancer software is built using the INETD superserver, so that all requests are tunneled through the balancers.

Service Provider Service providers do the actual work. The overhead that out architecture requires is a daemon, the PEGASUS monitor, that observes the servers' workload and send tickets to a number of balancers. At this point we have to note that each service provider can have one or more balancers, and divide its service capacity among them. Each service provider has a limited capacity, known also to the balancer. Consequently a specified number of requests can be redirected to each Service Provider, in order to ensure that we will not have bottlenecks. The capacity is updated with each ticket update of the Service Provider, as we will see later on.

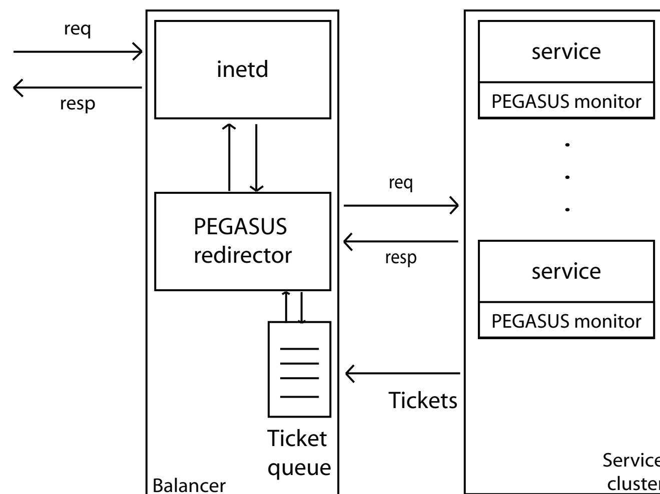


Figure 1: The PEGASUS Architecture

The scheduling that our system proposes is competitive. Most implementations rely on the existence of a delegator that holds all the information about the network capacity and thus performing the scheduling accordingly. In our case, we base our algorithm on two basic principles:

1. Each service provider wants to maximise its activity, thus it monitors the workload itself and demands more requests from the balancers in a competitive way to the other providers.

2. The balancer receives tickets from the service provider in an ad-hoc manner and stores them in a FIFO queue. On request, the balancer pops the first ticket form the queue and redirects the data to the appropriate service provider.

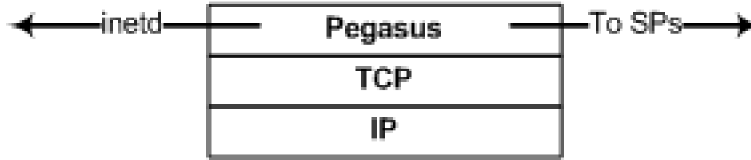
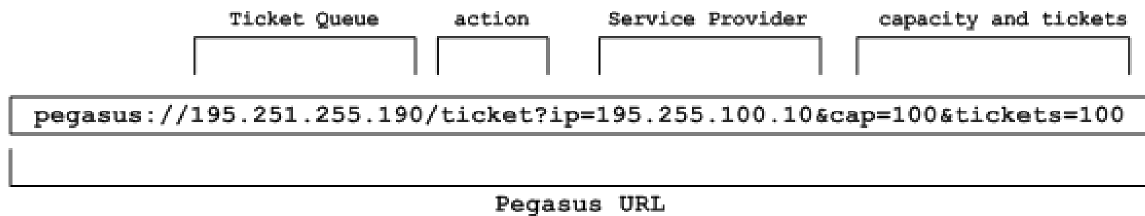


Figure 2: The PEGASUS Protocol Stack

The pegasus protocol stack is illustrated on Figure 2. The PEGASUS protocol is in the application layer and uses TCP for transport. The communication between the components is done through requests. There are three available messages, *(1) a ticket registration (TICKET_SP)*, a *(2) next available service provider (NEXT_SP)* and *(3) Service Balancer failure (FAILURE_SP)*. The requests are expressed through specified URLs [3].

For example the TICKET_SP command, requests from the redirector to add in its ticket queue, 100 tickets with maximum capacity 100 for the Service Provider with IP address 195.255.100.10. The request also states the service provider with its ip address, in this case 195.251.255.190.



The NEXT_SP command is listed below.

```
pegasus://195.251.255.190/next
```

The above command requests the next available Service Provider in the load balancing cluster. The response to this command is the IP address of the Service Provider or the message "error", that indicates that this balancer cannot server the request. Table 1 summarizes the available protocol commands.

Message	Description
TICKET_SP	Advertise availability by registration of tickets
NEXT_SP	Get next available Service Provider
FAILURE_SP	Informs the ticket queue that a Service Provider is unavailable

Table 1: PEGASUS protocol commands

Figure 3 depicts the internal workings of the PEGASUS load balancing system. Each request is received by the *inetd* superdaemon. Afterwards, it is redirected to the balancer, which communicates with the ticket queue using the protocol message *NEXT_SP* in order to get the next available Service Provider. Then the request is redirected to the Service provider that finally accept and address it.

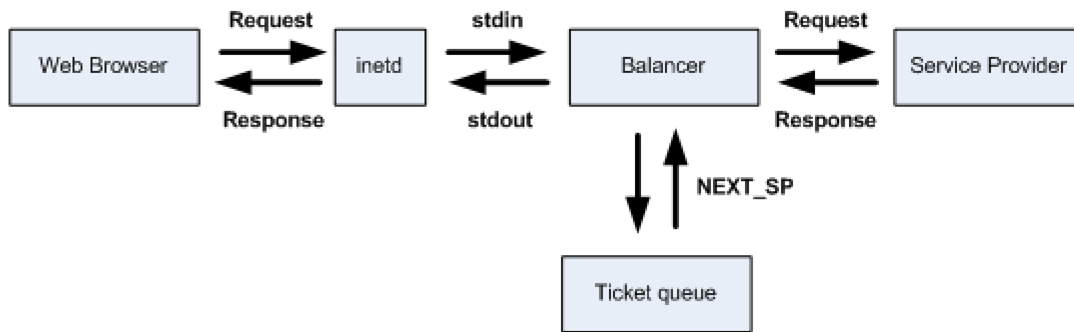


Figure 3: Internal PEGASUS Request Handling

3.2 Basic Characteristics

Our proposed architecture is the result of studying the advantages and drawbacks of previous open and commercial attempts. We have tried to provide a solution to various issues, while at the same time keeping the architecture as simple as possible.

Our implementation offers competitive performance and functional characteristics in the following aspects.

3.2.1 Multiple Points of Entry

By the term 'Multiple points of entry' we are referring to the fact that, in contrast to most available solutions, there is no single node that accepts all incoming requests. On the contrary, the number of nodes that accept incoming requests is left to the discretion of the system administrator, and the decision can be based upon performance or high availability considerations. It can be as many as all the nodes that provide content or as few as one.

3.2.2 Scheduling

The scheduling algorithms used are implemented in library files and communicate with the core of the system via a well defined application programming interface (API). Therefore, it is easy to add new scheduling algorithms plug-ins. The algorithm to be used can be defined in the PEGASUS configuration file. In our prototype we implemented a round robin algorithm using the ticket queue. For each request the available tickets for the selected Service Provider are decreased by one. If the balancer tries to redirect a request to a Service Provider that is unavailable, then the ticket queue is notified for the failure and all tickets for this Service Provider are discarded.

3.2.3 Service Transparency

The system has been designed to be service transparent in a sense that the pool of nodes may offer a variety of services with only a simple restriction. It will not handle persistent connection applications correctly. We have assumed that each connection is independent of all others and may be assigned to a different node. This is not true in the case of some well known applications, such as FTP and HTTP over SSL. The aforementioned applications are based on a number of different connections with the same end point. Furthermore, similar issues arise in the case of UDP-based services that 'remember' the state of the datagram exchange and are based on that to function correctly. Such an example is the well known Trivial FTP service. In our design each UDP request will be forwarded, in the typical scenario, to a different node. Therefore a TFTP transfer should fail.

We have refrained from considering service-specific solutions to the issue of connection affinity, such as those proposed and used by LVS [10]. We are planning to design a generic scheme that will automatically detect such cases and handle them correctly, without any modification to the configuration. For example, we are considering to detect UDP 'streams'. Each UDP datagram that is a potential part of such a stream will be forwarded to the same back-end node bypassing a scheduling decision. This is rather similar to the way that NAT handles UDP datagram exchange.

3.2.4 Sparse Distribution of Nodes

Our proposed architecture is designed in a way that will allow for a sparse distribution of nodes, in network topology terms. This is in contrast to NAT-based solutions which generally require all server nodes to reside on the same network, in a sense that they must share tinetdhe same gateway to the internet.

3.2.5 Ad-hoc

The PEGASUS system should allow a sparse physical network topology. The balancer nodes should not be statically aware of their service providers. Each content provider should be able to establish its presence dynamically and withdraw without notification and other configuration change. It is PEGASUS' system main goal, that each node will load-balance itself and notify a number of balancers about its availability. In our prototype we gather information about each node availability using the *getloadavg* function call of *stdlib.h*. More specifically we process CPU load at fixed time intervals.

Furthermore, many available load balancing solutions require that all nodes offer an identical set of services and have access to the same content. This is not necessarily the case with PEGASUS. The same infrastructure can be used to perform balancing for multiple distinct services at the same time. Therefore not all services will be necessarily offered by all nodes. Rather, a service will be provided only by a subset of content providers.

3.2.6 Simple Installation and Configuration

Our proposed architecture is based on well known user space tools. The prototype implementation is using INETD as the basis application. Using INETD we provide a simple scheme for service categorization, using TCP/IP network ports. The rest of the system, is written in C using BSD socket API. These decisions where made in order to make the system portable and with minimal overhead. The prototype is not using any DBMS or other storage method. Therefore, installing the suite of tools is a trivial task. No modification to the kernel is required. Downloading the binaries and performing trivial modifications to text configuration files are all the tasks needed in order to get the system up and running. Each participating service provider requires also to run a daemon that will notify the load balancer about their availability.

3.3 Comparison

In order to compare our prototype system with other load balancing frameworks, we should define a set a functional and non-functional characteristics. These are:

Participation policy: The policy in which the Service Providers are participating into the load balancing cluster. For example, in PEGASUS, each service provider registers and de-registers itself according to its availability. On the other hand, in order to configure mod_backhand, the configuration files of the apache web server must be modified for each service provider.

	PEGASUS	LVS	mod_backhand	RRDNS
Participation policy	ad-hoc	fixed	fixed	fixed
Network Topology	sparse	sparse	dense	sparse
OS Dependence	no	Linux only	no	no
Services	TCP, UDP based	TCP, UDP based	HTTP	any
Scheduling	round robin	varies	varies	round robin
Redirection Overhead	small	minimal	small	none
Installation Complexity	minimal	complex	simple	simple

Table 2: Comparison of Popular Load Balancing Frameworks

Network Topology: Most frameworks are dependent of the IP configuration of the load balancing cluster. The categorization *sparse* denotes a cluster that includes Service Providers alienated with each other. Dense indicates a cluster of Service Providers in the same or neighboring subnets. Therefore, a sparse topology means that service providers can be geographically spread over a wide region.

OS Dependence: Dependencies on one or many operating systems. For example, LVS works only on Linux based machines.

Services: The type of services that can be balanced. For example, TCP denotes all services that use it as transmission protocol. HTTP refers only to Hyper Text Transfer Protocol. It is all an issue of the layer that the balancer operates on. mod_backhand is essentially an application (layer 7) redirection mechanism, whereas PEGASUS operates on layer 4 and lvs, depending on the configuration, can perform request redirections on layer 4 or lower.

Scheduling: The scheduling algorithms that a load balancing system uses to distribute the request among the Service Providers.

Redirection Overhead: The actual overhead of request redirection. It is categorized in *small*, *minimal* and *none*.

Installation Complexity: The complexity of the installation procedure. Each installation dependency increases the complexity. For example, LVS requires kernel recompilation on both Balancers and Service Providers and it is labeled complex under this characteristic. Aside from that, depending on the choice of configuration (lvs/nat, lvs/tun etc), it may require manually tweaking the kernel routing table and setting virtual, 'spoofed' IP adresses on the node interfaces.

Table 2 summarizes the main characteristics of some popular load balancing frameworks.

As depicted, all frameworks have their advantages and disadvantages. Generally LVS is a very good solution which covers all major protocols, but it is available only on Linux boxes. mod_backhand its one of the most well implemented and featured frameworks, but it supports only HTTP. We compared PEGASUS with these frameworks having in mind the current implementation and not the final distribution. This is the reason that the scheduling characteristic has only implemented the round robin algorithm, instead of a variety that will be available when it reaches maturity.

3.4 Prototype Implementation

PEGASUS is implemented purely in C. It is compiled under Microsoft Windows and various UNIX variants using GCC. To work properly on Microsoft Windows PEGASUS uses the cygwin ex-

ecution environment. The PEGASUS distribution includes three executables, a *ticket queue*, a *pegasus_inetd* redirection and a *monitor module*. The first two are running on the Balancer and the monitor module on the Service Provider. The prototype was compiled with 4.0.2 version of GCC and the request were redirected on an apache 1.3.33 web server. The initial testing of the prototype was promising, but it is still on alpha version and no conclusive benchmarks could be conducted.

4 Conclusions and Further Work

In the near future we are planning to conduct extensive benchmarks. Furthermore, we are considering to test PEGASUS on a large-scale, real-world installation. This will, most likely include many Server Provider nodes in a sparse topology. We will implement four distinct benchmarking scenarios:

Single Service Provider: In this scenario a single node will act as the Service Provider.

Multiple Service Providers with mod_backhand: Many Service Providers are used. Apache and mod_backhand will be used to balance requests.

Multiple Service Providers with LVS: Many Service Providers are used. Apache and LVS will be used to provide load balancing.

PEGASUS: The same Service Providers will be used but requests will be balanced with PEGASUS.

For our benchmarks we will use the infrastructures of the Computer & Communication Systems Laboratory (CCSLAB) and the Information Systems Technology Laboratory (ISTLAB) of the Athens University of Economics and Business. All nodes will be Intel-based, single-CPU PCs.

Security is also an issue worth considering. With the current design of PEGASUS, the message exchange is unencrypted and the nodes that communicate do not authenticate the originator of the message. Imagine a scenario where a malicious node pretends to be a Service Provider, issuing ticket requests to a Ticket Server. The fact that the latter trusts that ticket requests are issued by valid hosts would result is a number of serve requests getting redirected to the malicious host. This could eventually lead to possible Denial of Service attacks. In order to counter this, we are planning to use methods that will help authenticate the nodes exchanging load information messages. Those are most likely to be combined with encrypting the content of the messages. A typical approach to the issue could be the usage of Secure Shell (SSH) tunnels or the use of an SSL/TLS based solution. However, this has not been decided at the time of writing.

As stated previously, the goal we had when designing the prototype application was not to out-perform existing open or commercial solutions. Instead, we wished to offer proof of concept regarding the various advantages PEGASUS has to offer. However, high performance is the quintessence of Load Balancing schemes. Therefore, in the near future, once the prototype is stable, we will start fine-tuning the code in order to optimise performance.

References

[1] Germ an Goldszmidt and Guerney Hunt. NetDispatcher: A TCP Connection Router. Technical report, 1997.

[2] D. Andresen, T. Yang, V. Holmedahl, and O. Ibarra. Sweb: Towards a scalable world wide web server on multicomputers. In *Proc. of 10th IEEE International Symp. on Parallel Processing (IPPS'96)*, pages pp. 850–856, April 1996.

[3] Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (uri): Generic syntax. Technical report, Network Information Center, January 2005. RFC-3986.

[4] T. Brisco. DNS Support for Load Balancing. *RFC 1794*, April 1995.

[5] Inc Cisco Systems. World Wide Web, January 2005. Available online at http://www.cisco.com/univercd/cc/td/doc/product/iaabu/localdir/ldv42/index.htm.

[6] Ralf S. Engelschall. Load balancing your web site. practical approaches for distributing http traffic. Website, May 1998. Available online at http://www.webtechniques.com/archives/1998/05/engelschall/.

[7] Michael Rumsewicz. Web servers for commercial environments: The imperatives and the solution. Website, March 1999. Available online at http://eddie.sourceforge.net/txt/WP_1.0.html.

[8] Theo Schlossnagle. The backhand project: load-balancing and monitoring apache web clusters. ApacheCon 2000, 2000.

[9] Wensong Zhang. Linux virtual server for scalable network services. Ottawa Linux Symposium 2000, July 2000.

[10] Wensong Zhang. Persistence handling in LVS. Website, 2004. Available online at http://www.linuxvirtualserver.org/docs/persistence.html.

[11] Wensong Zhang and Wenzhuo Zhang. Linux virtual server clusters: Build highly-scalable and highly-available network services at low cost. *Linux Magazine*, November 2003.