IV. Alkalmazott Informatika Konferencia 2005, University of Kaposvár, Hungary, Acta Agraria Kaposváriensis (2005) Vol 10 No 1, 195-207.

INTELLIGENT DYNAMIC LOAD BALANCER FOR JBOSS APPLICATION SERVER

Péter Mileff[1], Károly Nehéz[2]

[1]PhD *student,* [2]PhD,

Department of Information Engineering, University of Miskolc

Abstract

The growth of Internet services during the past few years has increased the demand for scalable distributed computing systems. Ecommerce systems concurrently serve many clients that transmit a large, number of requests. An increasingly popular and cost effective technique to improve server performance is load balancing, where hardware and/or software mechanisms decide which server will execute each client request. Load balancing mechanisms distribute client workload equally among server nodes to improve overall system responsiveness. Load balancers have emerged as a powerful new technology to solve this.

This paper focuses on a new generation of adaptive/intelligent dynamic load balancing technique, which based on the J2EE technology and can be practical in J2EE application servers. The paper discusses in detail both the theoretical model of the load balancing and its practical realization. The effectiveness of the new balancing method will be demonstrated through exact measurement results compared with former traditional non-adaptive methods.

IV. Alkalmazott Informatika Konferencia 2005, University of Kaposvár, Hungary, Acta Agraria Kaposváriensis (2005) Vol 10 No 1, 195-207.

Keywords: **Distributed systems, Adaptive Load Balancing, J2EE Application server, JBoss**

1. Introduction

As the number of concurrent requests received by a standalone server increases, the application exceeds the estimated respond time when the work load is too much on a server machine. At this time, there are two options to solve this problem: using faster machines or using multiple machines. The first solution is expensive and limited by the speed of a standalone machine. Second choice is more straightforward: deploy the same application on several machines and redirect client requests to those machines. The system is transparent from outside, which means, client applications perceive a standalone very-fast server with one accessible IP address. (see Figure 1) To achieve the performance and transparency, load balancing algorithms must be utilized.

Load balancing can improve the system performance by providing better utilization of all resources in the whole system consisting of computers connected by local area networks. The objective of load balancing is to reduce the mean response time of requests by distributing the workload.

1.1 Theoretical possibilities of realizing load balancing on OSI Layers

The OSI model was developed as a framework for developing protocols and applications that could interact seamlessly. The OSI model consists of seven layers and is referred to as the 7-Layer Networking Model [2]. Each layer represents a separate abstraction layer and interacts only with its adjoining layers. Load balancing mechanism can be realized on the Layer 3 - 7. OSI

levels 3 and 4 can be supported balancing mechanisms via network router devices. On layers 5 and 7, 'URL Load Balancing' can be achieved. A lively example of 'URL Load Balancing' can be the following: the URL may be static (such as *http://www.xxx.net/home)* or may be a cookie embedded into a user session. An example of URL load balancing is directing traffic to *http://www.xxx.net/documents* through one group of servers, while sending *http://www.xxx.net/images* to another group. URL load balancing can also set persistence based on the "cookie" negotiated between the client and the server.

1.2 Network-based load balancing

This type of load balancing is provided by network router devices and domain name servers (DNS) that service a cluster of host machines. For example, when a client resolves a hostname, the DNS can assign a different IP address to each request dynamically based on current load conditions. The client then contacts the designated server. Next time a different server could be selected for its next DNS resolution. Routers can also be used to bind a TCP flow to any back-end server based on the current load conditions and then use that binding for the duration of the flow. High volume Web sites often use network-based load balancing at the *network* layer (layer 3) and *transport* layer (layer 4). Layer 3 and 4 load balancing (referred to as "switching" [1]), use the IP address/hostname and port, respectively, to determine where to forward packets. Load balancing at these layers is limited, however, by the fact that they do not take into account the content of client requests. Higher-layer mechanisms – such as the so-called layer 5 switching described above – perform load balancing in accordance with the content of requests, such as pathname information within a URL.

1.3 Operating System - based load balancing

This type of load balancing is provided by distributed operating systems via *clustering*, *load sharing*, or *process migration* mechanisms. For instance Microsoft provides a new clustering possibility: Microsoft Cluster Server (MSCS) This special Microsoft software provides services such as failure detection, recovery, and the ability to manage the servers as a single system.] Clustering is a cost effective way to achieve high-availability and high-performance by combining many commodity computers to improve overall system processing power. Processes can then be distributed transparently among computers in the cluster. Clusters generally employ load sharing and process migration. Balancing load across processors – or more generally across network nodes – can be achieved via *process migration* mechanisms, where the state of a process is transferred between nodes. Transferring process state requires significant platform infrastructure support to handle platform differences between nodes. It may also limit applicability to programming languages based on virtual machines, such as Java.

1.4 Middleware-based load balancing

This type of load balancing is performed in middleware products, often on a per-session or per-request basis. For example, layer 5 switching has become a popular technique to determine which Web server should receive a client request for a particular URL. This strategy also allows the detection of "hot spots," *i.e.*, frequently accessed URLs, so that additional resources can be allocated to handle the large number of requests for such URLs.

Middleware-based load balancing can be used in conjunction with the specialized network-based and OS-based load balancing mechanisms outlined above. It can also be applied on top of

consumer level (COTS) networks and operating systems, which helps reduce cost. In addition, middleware-based load balancing can provide semantically rich customization possibilities to perform load balancing based on a wide range of application-specific load balancing conditions, such as run-time I/O vs. CPU overhead conditions.

Figure 1.

2. The practical approach of balancing problems

After we have surveyed the theoretical bases of the balancing in a few words we direct our attention to more practical scope of the problem.

A dynamic load balancing can be either *preemptive* or *non-preemptive*. A non-preemptive mechanism transfers only jobs that have just arrived, while a preemptive mechanism transfers jobs at any time, even when the jobs are in execution. Because preemptive mechanism are more costly than non-preemptive one and most of the benefit that can potentially be achieved through dynamic load balancing can be achieved using non-preemptive transfer only, non-preemptive transfers are usually used. Various proposed dynamic balancing methods are based on several policies. Three important ones among them are the transfer policy, the location policy and the selection policy, which decide when, where and what jobs should be transferred respectively. Much work [2][4] has been published on the design of transfer and location policy but very few on the selection policy.

Balancing policy: When designing a load balancing service it is important to select an appropriate algorithm that decides which server node will process each incoming request. For example, applications where all requests generate nearly identical amounts of load can use a simple Round-Robin algorithm, while applications where load generated by each request cannot be predicted in advance may require more advanced algorithms. In general, load balancing policies can be classified into the following categories:

- *Non-adaptive* – A load balancer can use non-adaptive policies, such as a simple Round-Robin algorithm or a randomized algorithm, to select which node will handle a particular request.
- *Adaptive* – A load balancer can use adaptive policies that utilize run-time information, such as CPU and disk I/O utilization, network loading.

2.1 Problem of real-time load balancing

The requests over the network arriving from clients and start a process in memory. Each process runs separated from one another and rivals in gaining resources. The objective of the balancers is to distribute these processes among the individual servers, that response time of processes will be minimal. Because the characteristic of the running tasks can be very various, so it is essential to use an adaptive load balancing algorithm, which try to distribute the tasks in an intelligent way using as it is called *load information*. This is a very difficult objective, beacause the balancer must accommodate the given job. When we could know in advance what type of task will be arrive, the scheduling algorithm could easily choose the most suitable server for the

task, but the type of the tasks knows in general only the client. So the traditional algorithms like Round-Robin or Random access can be usable only with a certain type of tasks.

Leland and Ott [4] analysed 9.5 million UNIX processes and found that there are three type of processes: CPU intensive processes use great amount of CPU cycles but do a little I/O operations; I/O intensive processes do a great deal of I/O but use a little CPU cycles; canonical processes do a little I/O and use a little CPU cycles. The amount of processes using great amount of CPU cycles and doing a great deal of I/O is extremely small.

Cabrera[5] analysed 122 thousand processes running on VAX11/785 and found that mean lifetime of processes is 400 ms, the lifetime of 78% of processes is shorter than one second, 97% of processes terminate within 8 seconds. The author concluded that only long live jobs should be candidates for load balancing due to the overhead costs involved. In a loosely coupled distributed system based on network message passing, a job running longer is often more suitable to transfer than a shorter job since the overhead of transferring a short job may override the benefit.

But not all long running jobs are suitable to transfer. Interactive jobs which constantly need I/O through keyboards and screens, and I/O intensive jobs which heavily access the local file system, will run better locally even though the local CPU load is very heavy[3]. In a world, only long life and CPU intensive jobs are worthwhile to transfer for remote execution. The question is: how does a scheduler know whether a job is long running and CPU intensive before executing the job? So, a selection policy based on predicting behaviour of a job including it's lifetime and type is needed to choose which job is suitable for transfer. Little work has been published on this area. The difficult task is to decide the suitability of transferring a job non-preemptively, i.e., predicating the execution time and resource requirements of the job before it is actually executed.

Leland and Ott[5] found that the residual CPU time needed by a process is linearly related to the amount of CPU time already received by the process (age). The authors developed an assignment scheme which distributes processes based on their age. But an estimate of CPU requirement is unavailable prior to a process's execution.

3. Concept of an Intelligent Load Balancer

To create an efficient Load Balancer is a very difficult objective. There are of course many theoretical load balancing solution methods, but many times the practical modell doesn't make these implementation and efficiency possible. To find the suitable and optimal method for balancing, it is essential to have the most deep knowledge level of the specific system.

Before we go into the details of the Load Balancer, let us examine first the theoretical modell, which is show in *Figure 2*:

Figure 2.

The theoretical functionality of the balancer is the following: Standalone clients initiate requests over the network through HTTP protocol or RMI to the JBoss cluster. The JBoss cluster can be a complex of homogeneous or inhomogeneous computer on which the JBoss application server runs in cluster mode. Of course, more clients can initiate a request at the same time to the cluster, so the cluster must fulfil more than one request parallel. The incoming requests are received and directed to the compliant node of the cluster by the intelligent load balancer. So it's objective is to choose the most ideal node in term of execution based on the collected *load information* by the Dispatcher. To elect the ideal node is not an easy matter. The main objective

of the balancer is to realize a more effective task-division, which response time can be more better than the former algorithm. In additional we concentrate the detailed elaboration of the practical realization.

3.1 Components of the Load Balancer

The architecture of our Balancer essentially can be divided into three individual components: the *Statistics Service*, the *Dispatcher*, and the *Scheduler* as well. The individual units are in close communication with one another, none of them can operate without the others. At present the connection of the units works on the concept of the *Remote Method Invocation(RMI)*, but the following objective is to change the entire comminucation or part of that to the new TreeCache method of JBoss. Utilizing TreeCache, response time may be shorter.

3.1.1 Statistics Service

We can consider from the description above, that the Statistics Service is responsible for the load information. Naturally this unit must run on each node. When a new node come into the cluster, then the Statistics Service start immediately on it. It attempts to find the Dispatcher and provide data to it. *Figure 3* shows the architecture of the Statistics Service and the Dispather:

Figure 3.

Figure 3 shows that Statistics Service is consisted of three parts: *CPU* -, *I/O Statistics* and *Fuzzy Engine*. The functionality arise from those name: CPU Statistics services the CPU usage

and I/O Statistics the I/O usage of the specific node. The CPU Statistics and the Fuzzy Logic represent collectively an MBean*(Managed Bean)* unit, however the I/O Statistics is an another separate MBean unit. In the JBoss system each MBean indicate services. The sufficient node-information are essential to the compliant operating of the balancer. In fact, Java classes are running in a virtual machine on each host, therefore it does not make it possible to query the load information directly from the operating system. For this reason we had to evolve individual methods and had to utilize operating system specific resources. Nevertheless these resources are operating system dependent.

The current version of the balancer works on MS Windows Systems, but further objective is to create Linux/Unix version too. Since the Java 1.5 appeared , it become possible to measure the CPU average usage with the Java Management Extension technology, using the built in OperatingSystemMXBean class. It has a function named *getProcessCpuTime()*, which can query the CPU time of the specific JVM*(Java Virtual Machine)* in nanosecond, from which the average CPU usage can be computed. The CPU usage can be query direct from the operating system, but in this case the efficiency of the balancer can degrade to a great extent. The reason for this is that, the MS Windows operating system updates the data of the *Performance Monitor* every 1000 millisecond, on account of which the schedule of the short task becomes impossible. The JBoss system can work with 50 ms sample time, but in this instance the data acquisition is fulfilled in every 100 ms.

The acquiring the I/O information is already much harder task by far. Now Java helps us neither so much as was in case of CPU usage. To get the required infromation we need operating system level methods, to which the C/C++ programming language ensures the suitable environment. The solution was realized by the technology as called JNI*(Java Native Interfaces)*,

which makes merging the C/C++ and the Java programming language possible. So the survey data of the I/O are realized by native invocation. However the operating system is again a limiting factor, because the data are only updated in every 1000 millisecond. If the client requests are not so frequent, this limit is enough in practice.

Before we change to the consideration of the Fuzzy Engine, it is necessary to make a mention of a relevant feature of the statistics collector MBeans. All the nodes send information to the Fuzzy Engine, when the avarage usage of these are smaller than 100%. This is the most essential condition of the operating of the balancer, what we will detail in the discussion of the Dispatcher.

The Fuzzy Engine is responsible for the part of the adaptivity of the balancer. It gathers the information sended by I/O and CPU services and deducts a *fuzzy* value between 0 and 1 supported by a preset *Fuzzy Engine*. This fuzzy value will be sent to the Dispatcher, that stores it in a hashtable. Current version of Balancer use three fuzzy linguistic variables: one for I/O and an other for CPU utilization and the third one indicates the service capability of a server node. First two variables are considered input variables and third one as output variable. Both input variables are divided into three membership functions, therefore output server capability must be divided into six membership functions. Further aim is to fine the shape of membership functions using a fuzzy-neuro engine. In Figure 4, all membership functions of fuzzy variables can be seen.

Figure 4.

3.1.2 The Dispatcher

The Dispatcher is the second most important part of the Load Balancer. It is also realized by MBean. It's objective is to store the status information sended by each node in a hashtable structure. *Figure 3* shows the architecture of the Dispatcher.

The sent forwarded information consist of two parts: a fuzzy engine value and the IP address of the specific node. The IP address is essential to identify the nodes. The infomormation gets to a hashtable bucket in a vector with the time of arrival together. As *Figure 3* shows, the key of the hashtable is the IP address, because it is individual.

By the discussion of the Statistics Service we have mentioned, that there is a condition, whereas a node only send the information to the Dispatcher, when its load is fewer than 100%. In Dispatcher this effects, that the belonging stored information of the hashtable bucket will not be updated. The *time stamp* of the data is therefore essential, because relying upon these findings will the balancer make a decision to wich information are timely, and which not.

The Dispatcher can be find on only one node in the cluster. It makes no difference on which, but the best thing to do is that, it is started on the fastes node. The connection between the Statistics Services and the Dispatcher is dynamic, that is each node in startup finds and stores the address of the node on which the Dispather runs.

3.1.3 The Balancer

After preparation of the data the work of the balancer is no more so difficult. However we have to pay attention at the optimal implementation, because the least mistake can also cause big response time decrease. The balancer is a java class implemented a *CustomLoadBalancePolicy* interface, which is functionally part of the JBoss base interfaces.

Its theoretical operating is the following: The balancer make a decision on the bases of the status information collected from server nodes. It considers those information valid, which arrived within 150 ms. Those nodes, which are highly loaded, they don't send any information to the Dispatcher, so naturally the balancer doesn't give them a new task. The balancer will choose the node with the best fuzzy engine value. However in case of a big loaded cluster can often occure so, that all of the nodes are loaded fully and none of them makes a sign. Nevertheless in this case the balancer have to choose one of them, but the question is which one.

Many solution methods have sprung up, however by reason of the tests it appeared, that such method needed, which can efficiently distribute the works in case of big loaded nodes. The first solution is the random distribution. It can be good, or can be very bad because of the random distribution. For instance if the random balancer gives the work to such node, which is slower than the others, and of course also loaded on 100%. It proved a little better that method which gives the work to that node, which average non-response time is the least, if every node are out of time constraint.

A very important element of the balancer is the following: in the current version of the balancer a node can get a work twice one after the other only, if its CPU usage doesn't correspond to the stored value at the giving out of the previous work and also this value is so more little, than the value of all the nodes. This condition came into the balancer therefore, because when almost more clients all at ones give their requests parallel, then without this condition the same node receive the request of more clients, because the requests are so close to one another, that the data of the balancer couldn't update so quickly.

3.2 Test and results

The testing process has been carried out on a JBoss cluster consisting seven homogeneous PC-s. Each machine had Pentium III 733 MHz CPU with 256 MByte RAM. Machines were connected via 100Mbps Ethernet network. Utilized operation system was Windows 2000 SP5. Application server version was JBoss 3.2.5 'WonderLand'.

Simulating client requests was carried out with a generic professional simulation environment: Apache JMeter [8]. During testing process server machines where slowed-down randomly with a special Loader-MBean emulating I/O or CPU load. Loader-MBean is used for emulating other clients requests and other applications that are parallel launching on the server nodes.

We have started the simulations with a client, then we increased the number of clients to seven. In the course of all simulation we have tested all algorithms three times, then we represented these average results on the *Figure 5*. The diagram shows properly, that in every case the results of the Round-Robin fell short of the results of the Intelligent Balancer.

Figure 5.

If we examine the results we can see that, the value of the Throughput increases with the increasing number of the clients, although it is not in direct ratio. The more clients initiate request to the cluster, the more clients share the CPU. Exactly that is why it does no good to more increase the number of the client - like number of the nodes - in the course of the measurement, becase at such times the scheduling lose its importance.

Of course, it depends on the type of the task scheduling requisted task, that they in what extent require the resources. In the course of seven homogeneous nodes optimal distribution is, if all of them get one. Certainly, we assume that the request of the clients arrive in near time.

Because the artificial loads run in random time on the nodes, therefore certain corresponding with the number of nodes or more the nodes become full.

At that time every node are maximum load. Whereas at such time the scheduling is impossible, therefore the best solution is that, if we distribute the taks optimal among the nodes till then, while the scheduling will be become possible. The Balancer does it in two ways:
with random node-choosing and with average response time. One node could not get two tasks one after another.

The Figure 5 shows both results of the algorithm and with increasing of the number of clients - which means that the more task get into the system - better and better approach the theoretical maximum of the respose time of the Round-Robin and the Intelligent Balancer.

In the event of inhomogeneous nodes certainly we can reach much better response time, but of course it depends on the inhomogenity of the nodes.

The following table summarizes, how much speed increase can be achieved utilizing new Balancer compared with applying Round-Robin algorithm. Results highly depend on the type of tasks: a task to what extend claims the capcity of a node. In our test environment, execution time of a task is 500 ms on a non-loaded server node. Client requests follow each other within 500ms time intervall and plus minus 200ms uniform random time. Aim of random intervall is to simulate realistic non-predicted client requests. Based on the test results, it is clear that our intelligent balancer algorithm has better performance than Round-Robin algorithm.

Table 1

During tests intelligent load balancer and only Round-Robin algorithm was compared because Round-Robin algorithm is definetly better that other classic methods like: First Available and Random balancer algorithms. Thus our aim was to outstrip this traditional non-adaptive method.

4. Conclusion

An intelligent fuzzy-based Load Balancer Application and its test results have been presented in this paper. Continuing work will focus on further developing and implementing more flexible XML based configuration possibilities and redesign communication between server nodes and the dispatched session bean utilizing the new JBoss TreeCache introduced by the latest JBoss version 4.0.

5. Acknowledgements

The research and development summarized in this paper has been carried out by the Production Information Engineering and Research Team (PIERT) established at the Department of Information Engineering and supported by the Hungarian Academy of Sciences. The financial support of the research by the aforementioned source is gratefully acknowledged.

6. References

[1] J. Lindfors, M. Fleury, The JBoss Group: **JMX:** ***Managing J2EE with Java Management Extensions***. SAMS Publishing Inc., 2002.

[2] J. Basney and M. Livny, **"Deploying a High Throughput Computing Cluster,"** ***High Performance Cluster Computing***, vol. 1, May 1999.

[3] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, **"The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware"**, in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[4] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann: ***Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.*** Wiley, 2000.

[5] L.M.Cabrera, ***"The influence of workload on load balancing strategies"***, in Proc. Summer USENIX Conf., pp. 446-458, June 1986.

[6] W.Leland and T.Ott, ***"Load balancing heuristics and process behavior"***, in Proc. ACMSIGMETRICS Conf. Measurement and Modeling of Computer Syst., May 1986.

[7] Jack Shirazi: ***Java Performance Tuning, Second Edition***, O'Relly, 2003.

[8] ***JMeter Generic Simulation Environment***, http://jakarta.apache.org/jmeter, 2005. (Apache Jakarta JMeter)

[9] ***JBoss – Leading J2EE Open Source Application Server***, www.jboss.org, 2005

[10] ***Fuzzy Logic Systems***: http://www.seattlerobotics.org/encoder/mar98/fuz/flindex.html, 2005

[11] Chandra Kopparapu: ***Load Balancing Servers, Firewalls, and Caches***, Wiley, 2002

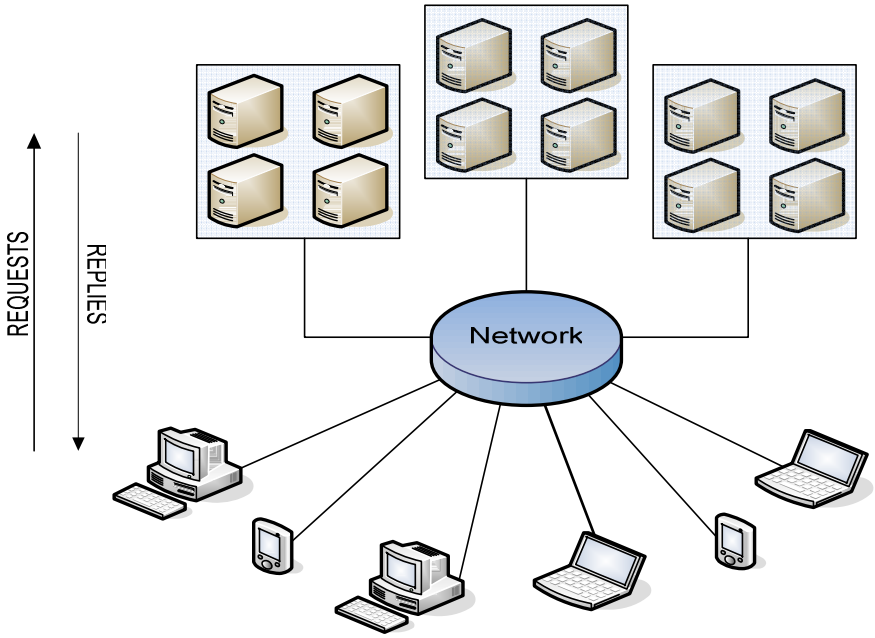

Figure 1.
Horizontal load balancing

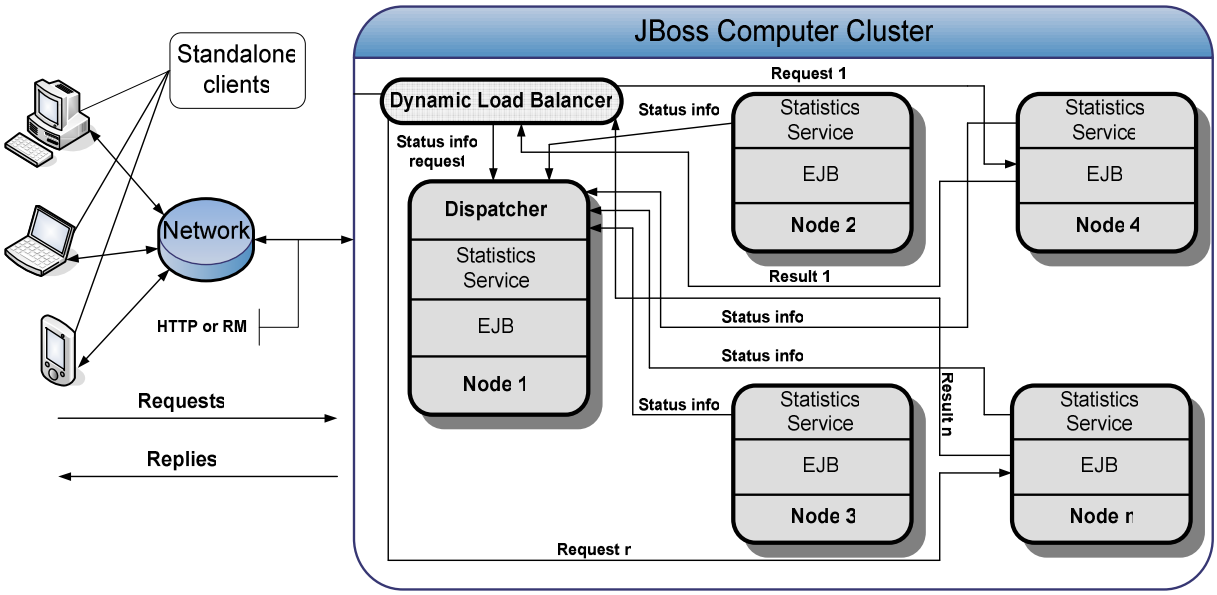

Figure 2.
JBoss Load Balancer Architecture

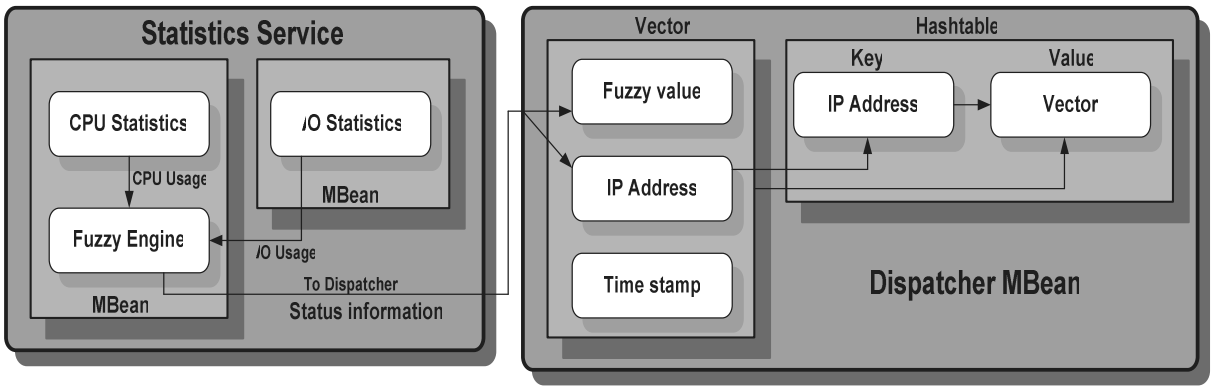

Figure 3.
Elements of Statistics Service

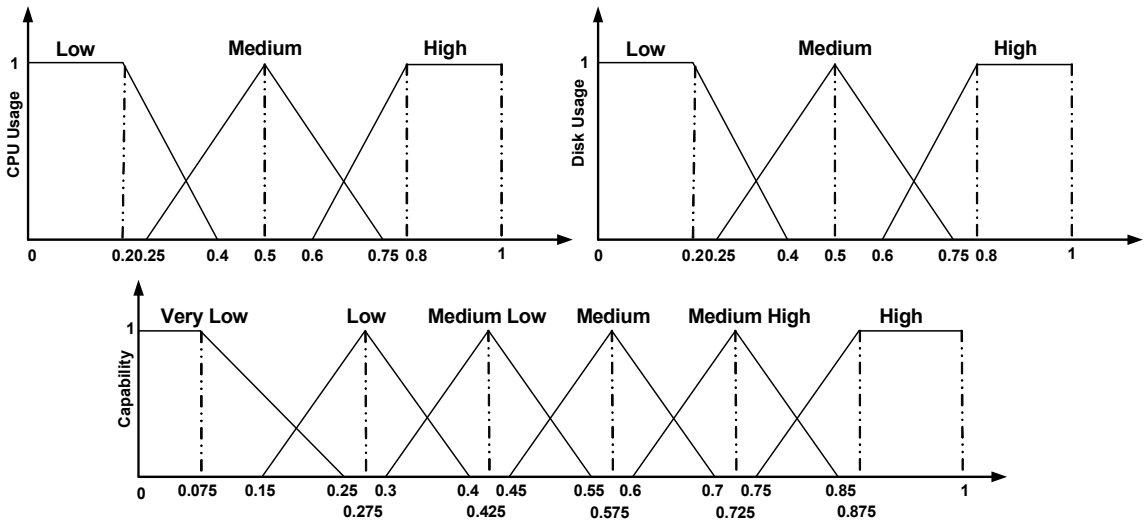

Figure 4.
The Fuzzy Engine Linguistic Variables

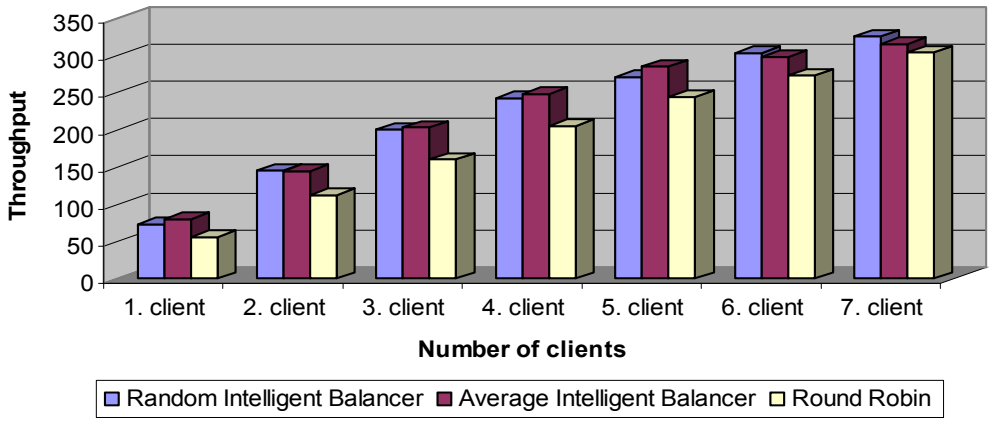

Figure 5.
Test results

Table 1
Balancing Algorythms comparison

Balancer Type	Speed Improvement / client						
	1. client	2. client	3. client	4. client	5. client	6. client	7. client
Random Intelligent Balancer	25%	23%	20%	16%	10%	10%	7%
Average Intelligent Balancer	30%	23%	21%	18%	14%	9%	4%