Production Systems and Information Engineering
Volume 3 (2006), pp. 57-71.

FUZZY BASED LOAD BALANCING FOR J2EE APPLICATIONS

PETER MILEFF
University of Miskolc, Hungary
Department of Information Engineering
mileff@ait.iit.uni-miskolc.hu

KAROLY NEHEZ
University of Miskolc, Hungary
Department of Information Engineering
nehez@ait.iit.uni-miskolc.hu

[Received: November 10, 2005]

Abstract. The growth of Internet services during the past few years has increased the demand for scalable distributed computing systems. E-commerce 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 the client request. Load balancing mechanisms distribute client workload 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 J2EE technology and can be practical in J2EE application servers. The paper discusses in detail both the theoretical model of 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.

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

1. Introduction

As the number of concurrent requests is increased on a standalone server, so the application exceeds the pre-estimated respond time, because the work load is too much on the server machine. At this time, there are two options to solve this problem: using faster machines or using multiple machines parallel. The first solution can be expensive and limited by the speed of a standalone machine. The 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 that 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 system performance by providing better utilization of all resources in the whole system, which consists of computers connected by local area networks. The main objective of load balancing is to reduce the mean response time of requests by distributing the workload [5].

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.

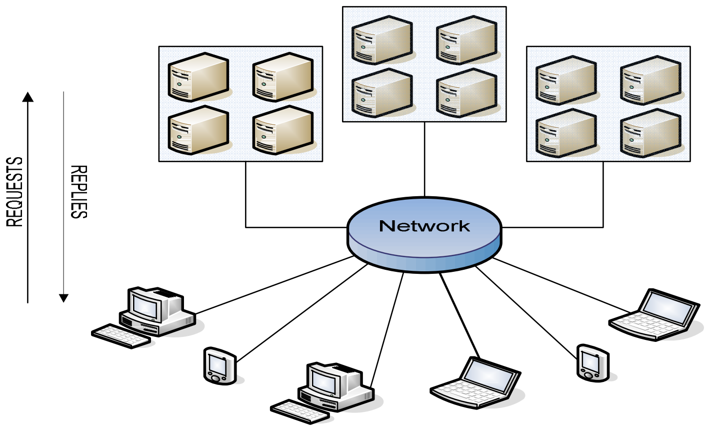

Figure 1. Horizontal load balancing

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.

2. The practical approach of balancing problems

After we have surveyed the theoretical bases of load balancing, we direct our attention to a 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.

This paper presents a new adaptive load balancing method, which efficiency are verified with help of many simulations.

2.1 Problem of real-time load balancing

Client requests arrive over the network and start a new process in memory. Each process runs separated from one another and rivals in gaining available resources. The objective of load balancers is to distribute these processes among the individual server instances in such a way 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 tries to distribute tasks in an intelligent way using *load information* of the nodes. This is a very difficult objective, because balancer must conform to the given job. If it could be known in advance what type of task will be arrive, then the scheduling algorithm could easily choose the most suitable server for the task. However, the type of 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] analyzed 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] analyzed 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. If the running time of the job is rather short, then load balancing can loose its importance.

3. Concept of an Intelligent Load Balancer

Creating an efficient Load Balancer is a very difficult objective. Of course, there are many theoretical load balancing methods, but many times the practical model does not make these implementation and efficiency possible. Finding suitable and optimal method for balancing, it is essential to have the deepest knowledge of the specific system. Our aim was to develop a new load balancer for JBoss application servers, because only three types of load balancers are available in JBoss cluster: Round Robin, First Available, and Random balancer.

Before we examine the theoretical model of the new Load Balancer, we make a short overview of JBoss cluster.

3.1 The JBoss cluster

JBoss is an extensible, dynamically configurable Java based application server which includes a set of J2EE compliant components. JBoss is an open source middleware, in the sense that users can extend middleware services by dynamically deploying new components into a running server.

A cluster is a set of nodes. These nodes generally have a common goal. A node can be a computer or, more simply, a server instance (if it hosts several instances). In JBoss, nodes in a cluster have two common goals: achieving Fault Tolerance and

Load Balancing through replication. These concepts are often mixed. JBoss currently supports the following clustering features [9]:

- Automatic discovery. JBoss cluster nodes automatically discover each other when they boot up with no additional configuration. Nodes that join the cluster at a later time have their state automatically initialized and synchronized by the rest of the group.
- Fail-over and load-balancing features for:
    - JNDI,
    - RMI (can be used to implement your own clustered services),
    - Entity Beans,
    - Stateful Session Beans with in memory state replication,
    - Stateless Session Beans

- HTTP Session replication with Tomcat (3.0) and Jetty (CVS HEAD)

- Dynamic JNDI discovery. With its JMX-based Microkernel architecture JNDI clients can automatically discover the JNDI context.

- Cluster-wide replicated JNDI tree. It is replicated across the entire cluster. It requires no additional configuration and boots up with a cluster-enabled JBoss configuration. Remote JBoss JNDI clients can also implicitly use multicast to discover the JNDI tree.

- Farming. JBoss farming takes this hot-deployment feature cluster-wide. Copying a deployable component to just one node's deployment directory causes it to be deployed (or re-deployed) across the entire cluster. Removing a component from just one node's deployment directory causes it to be undeployed across the entire cluster.

- Pluggable RMI load-balance policies. We used this feature to develop our load balancer.

JBoss uses an abstraction framework to isolate communication layers like JavaGroups. This was done so that other third-party group communication frameworks could be incorporated into JBoss seamlessly and easily. This framework also provides the tools and interfaces to write own clusterable services and components to plug into the JBoss JMX backbone.

Utilizing these flexibilities of the JBoss system we developed a new load balancer, which will be presented in details.

3.2 Architecture of the balancer

Before going into the details, first we examine the theoretical model, which is shown in *Figure 2*:

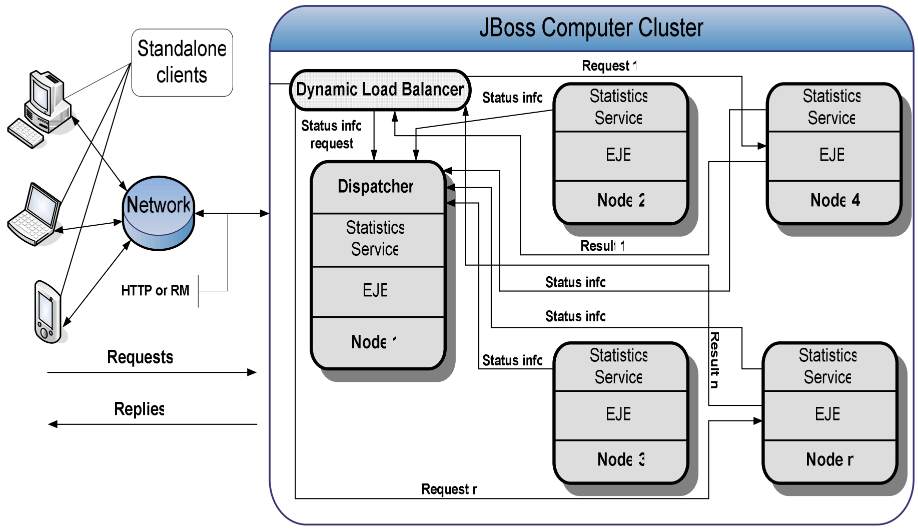

Figure 2. JBoss Load Balancer Architecture

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 computers [9]. JBoss application server runs on these in cluster mode. Of course, more clients can initiate a request at the same time to the cluster, so the cluster must fulfill more than one request parallel. Incoming requests are received and directed to the compliant node by the intelligent load balancer. So its objective is to choose the most ideal node, based on the collected *load information* by the Dispatcher MBean. To choose the ideal node is not an easy task. The main objective of the balancer is to realize a more effective task-division, which response time can be better than former algorithms. In the following, we show a detailed explanation of the practical realization.

3.3 Components of the 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 (within one JVM), none of them can operate without the others. At present, component connections work on the concept of *Remote Method Invocation (RMI)*, but the further objective is to change the entire communication or part of it to a new TreeCache method of JBoss [7]. Utilizing TreeCache response time may be shorter because it uses multicasting.

3.3.1 Statistics Service

We can consider from the description above, that Statistics Service is responsible for load information. Naturally, this unit has to be started on each node. When a new node joins the cluster, Statistics Service starts immediately on it, because the JBoss cluster deploys this MBean [9][1] automatically. This service attempts to find the Dispatcher and provides data to it. *Figure 3* shows the architecture of the Statistics Service and the Dispatcher:

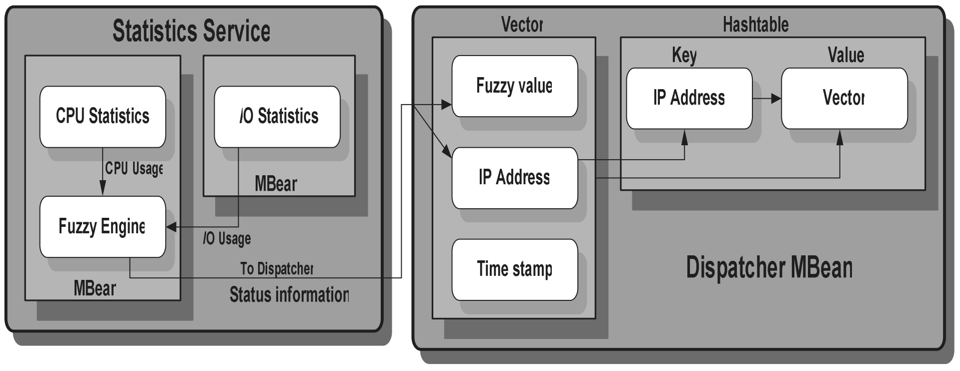

Figure 3. Elements of Statistics Service

Figure 3 shows that Statistics Service consists of three subcomponents: *CPU* -, *I/O Statistics* and *Fuzzy Engine*. The functionality of these arises from those names: CPU Statistics provides CPU usage and I/O Statistics provides information about I/O usage of a specific node. CPU Statistics and Fuzzy Logic components are represented collectively an MBean *(Managed Bean)*, however I/O Statistics is an another separate MBean. In JBoss system, each MBean can be considered as services. The sufficient node-information is 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 Java 1.5 appeared on the market, it become possible to measure CPU average usage with Java Management Extension technology, using the built in OperatingSystemMXBean class. It has a function called *getProcessCpuTime()*, which can query the CPU time of the specific JVM in nanoseconds, from which the average CPU usage can be computed. The CPU usage can be query direct from the operating system as well, but in this case the efficiency of the balancer can degrade to a great extent. The reason of this is that: MS Windows operating system updates the data of the *Performance Monitor* every 1000 milliseconds (one second), which makes impossible to schedule short tasks. JBoss system can work with 50 ms sample time, but in this instance data acquisition is fulfilled in every 100 ms.

Acquiring I/O information is much harder task. Getting the required information we need to call operating system level methods via JNI *(Java Native Interfaces)* technology, which makes possible to merge the C/C++ and the Java programming language. However operating system is a limiting factor again, because data are only updated in every 1000 milliseconds. If client I/O 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 average usage of these, is smaller than 100%. This is the most essential condition of the operating of the balancer that will be detailed below.

The Fuzzy Engine is responsible for the part of adaptivity of the balancer. It is integrated in the Statistics Service and gathers information sent 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 as input variables and third one as output variable. Both input variables are defined with three membership functions. Output server capability is defined with 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.

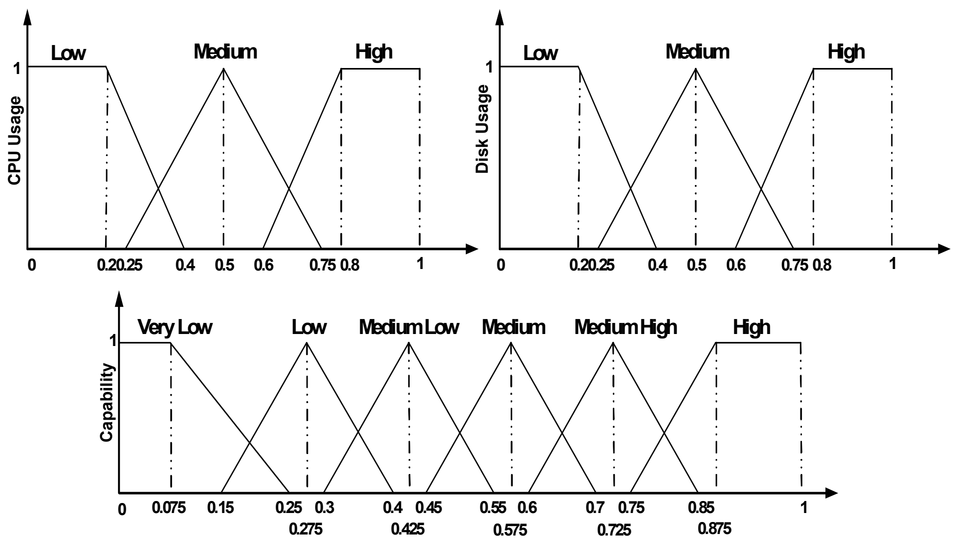

Figure 4. The Linguistic Variables of the Fuzzy Engine

3.3.2 The Dispatcher

The Dispatcher is the second most important part of the Load Balancer. It is also realized by MBean. Its objective is to store status information sent by the nodes in hashtable structure. *Figure 3* shows the architecture of the Dispatcher. The sent and forwarded information consist of two parts: fuzzy values and the IP address of the specific node. IP address is essential to identify the nodes. The information gets into a hashtable entry as a vector, together with the time of arrival (*time stamp*). As *Figure 3* shows, the key of the hashtable is the IP address, because it is always individual. By the discussion of the Statistics Service we have mentioned, that there is a condition, whereas a node only send information to the Dispatcher, when its load is fewer than 100%. This effects in Dispatcher that, the belonging stored information of the hashtable entry will not be updated. The balancer will make a decision based on the timestamp value, which information is current and which is not.

The Dispatcher is deployed only on one node in the cluster. It makes no difference on which one, but starting on the fastest node is the best. The connection between the Statistics Services and the Dispatcher is dynamic. At startup time, each node finds and stores the network address of the node, on which the Dispatcher runs.

3.3.3 The Balancer

After preparation of 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 workflow is the following: The balancer makes decision on the bases of the status information collected from the server nodes. It considers those information valid, which arrived within 150 ms. The highly loaded nodes do not send any information to the Dispatcher, so naturally the balancer does not give to one of them a new task. The balancer will choose the node with the best fuzzy engine value. However in case of a big loaded cluster it can often occur that all of the nodes are loaded fully and none of them makes a sign. Nevertheless, at this time the balancer have to choose one of them, but the question is which one.

Many solution methods have sprung up, because this case needs more consideration. Such method is needed, which can efficiently distribute the work among the highly 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 random balancer gives the work to a node, which is slower than the others, and of course also loaded on 100%, the response time of the system will be very low. We implemented this method as *Random Intelligent Balancer*, the results can be seen in *Table 1*. The method is proved a little better, which gives the work to that node, which average non-response time is the least, if every node are out of time constraint (*Average Intelligent Balancer*).

A very important element of the balancer is the following: in current version of the balancer a node can only get a work twice one after the other, if its CPU usage does not correspond to the stored value at the giving out of the previous work and also this value is 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 could not update so quickly.

4 Test and results

The testing process has been carried out on a JBoss cluster, consisting 7 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’.

Simulated client requests were 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 client requests and other applications that are parallel launching on the server nodes.

We have started simulations with one client and 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 the results of the Round-Robin in every case fell short of the results of the Intelligent Balancer.

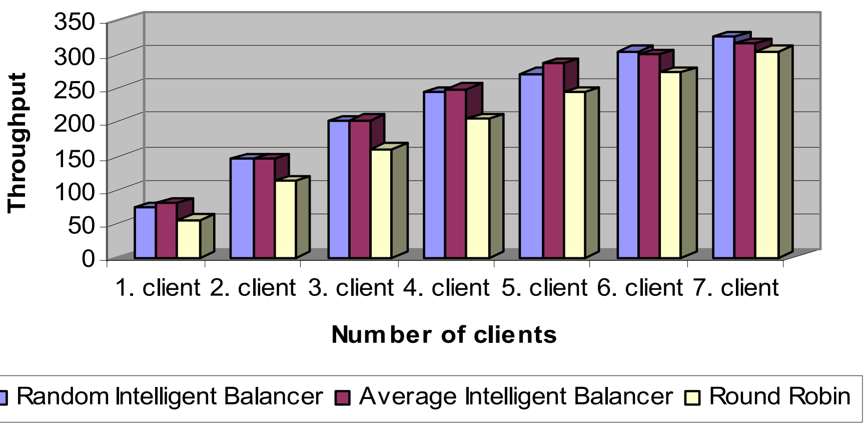

Figure 5. Test results

If we examine the results, we can see that the value of the Throughput is raised 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, for this reason there is no point about increasing the number of clients - like number of nodes - in the course of simulations, because at this time scheduling loses its importance.

Of course, it depends on the type of the tasks requested the clients and on that, in what extent they require the resources. In the course of seven homogeneous nodes the optimal distribution is, if all of them get one task. Artificial loads run in random time on the nodes independently from the client requests, which load the nodes for a period of time and to a certain extent. It is possible, if there are many client requests at the same time, then the cluster will be overloaded. At that time every node are at maximum load. Whereas at such time scheduling is impossible, therefore the best solution is that, if we distribute the tasks optimal among the nodes till then, while scheduling will be become possible. The Balancer does it in two ways: with random node-choosing (Random Intelligent Balancer) and with using average response time. One node could not get two tasks one after another.

The Figure 5 shows the results both of the algorithms. It is easy to see that increasing the number of clients - which means that more task get into the system – the response time of Round-Robin and the Intelligent Balancer approach better and better approximate the theoretical maximum.

In case 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 the new Balancer compared with Round-Robin algorithm. Results highly depend on the type of tasks: a task to what extend claims the capacity of a node. In our test environment, execution time of a task was 500 ms on a non-loaded server node. Client requests followed each other within 500ms time interval and plus-minus 200ms uniform random time. The aim of random interval 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. Balancing Algorithms 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%

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

5. 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.

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 afore-mentioned source is gratefully acknowledged.

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