SIMULATION OF LOAD BALANCING ALGORITHMS: A Comparative Study

Milan E. Soklic

Abstract

This article introduces a new load balancing algorithm, called diffusive load balancing, and compares its performance with three other load balancing algorithms: static, round robin, and shortest queue load balancing. The comparison of load balancing algorithms is made in three simulated client-server environments: a small-scale, intranet, and Internet environment. Experimental results of performance modeling show that diffusive load balancing is better than round robin and static load balancing in a dynamic environment, which manifest in frequent clients' object creation requests and in short objects' lifetimes. In this research, the diffusive load balancing algorithm is discussed in juxtaposition with the distributed client-server architectures.

Keywords: simulation, performance modeling, adaptive load balancing, distributed systems, client-server.

Introduction

In a client-server environment it is common to have more than one server processing clients' requests. In such cases there must be a mechanism to distribute the clients' requests to the servers. Depending on how this mechanism works servers may have different workloads. Workload of a server is determined by the amount of processing time needed to execute all clients' requests assigned to that server. To achieve best performance results a method applied needs to minimize workload differences between the servers. Load balancing is the term given to any mechanism that tries to achieve this.

Performance in a client-server environment can be measured by the response time defined as the duration of time starting when the client submits a request to the servers and ending when the servers notify the client, that the request has finished execution. It was shown by Winston [1] that the best mechanism for achieving optimal response time is to distribute the workload equally among the servers.

There is no known way to determine how much processing time a request will take prior to actually executing it. As a consequence, it is unlikely to accurately determine how much workload a server actually has, so approximate estimations are to be made for the workload, determined by the CPU utilization or the CPU wait-queue length.

An object created on a server, on behalf of the client, has its pertinent information about its state stored in the server's memory. During the object's lifetime, i.e. from the creation of the object to its destruction, the object may remain idle for periods of time, waiting for clients' requests, or it may be executing. A decision to assign an object to a server based on the current conditions could turn out to be incorrect, if it has not been considered how the object would be used by its clients in the future, i.e. when the object will be executing and when it will be idle. To achieve a full potential of all servers, a load balancing mechanism is needed to distribute equal workloads to the servers.

Architecture for Diffusive Load Balancing

The hardware architecture consists of a set of clients and a network of servers. The network of servers is described in terms of a graph $G=(V, E)$, where vertices V represent servers - nodes, and edges E represent communication links. Direction of edges indicates possible propagation of load requests to or from a given server. A client selects a server, via a network router, from the list of all available servers.

The software architecture for diffusive load balancing consists of two DCOM software components, known as the *object factory* and the *request acceptor*. When a client needs to create an object, it contacts the object factory component on one of the servers, by examining a list of all servers and choosing the one which has not been called for the longest time from that client. After the selected server is contacted, the object factory component on that server initiates a search for a granting server, which will accept the client's object creation request. The search for a granting server starts with the invocation of the request acceptor component on the selected server, which invokes request acceptors on its directly connected neighboring servers.

The request acceptor forwards the request to the server with the smallest amount of workload only, if the workload differential between that server and the server where the component is executing is greater than a specified threshold. Workload of the server is determined by counting the number of DCOM object tasks in the server's ready queue. This number might be less than the overall number of objects bound to a server, since some of the objects may be idle, e.g. waiting for client requests. The search for a granting server causes a traversal of the network along the directed edges in a diffusive fashion, i.e. along edges leading to less loaded servers, until the granting server is found.

Once a request acceptor component on a granting server accepts the client's request, the object factory component on the selected server creates the object on the granting server, and returns a pointer of the created object to the client. Having a pointer to an object, a client can call its methods directly. DCOM handles the internal communication, transport of arguments, and return values between the client and the server.

The Simulation Model

The simulation model described below provides a virtual environment for clients and a network of servers. Specifically, it mimics clients' behavior, it simulates different client/server hardware architectures, and responds to clients' behavior in the view of four load balancing techniques: static, round robin, diffusive, and shortest queue. It keeps track of all events in the simulation environment, using log files, in order to be able to analyze and plot various results for comparison of the four load balancing techniques.

Clients are single-user single-task processes operating independently of each other. During the simulation run a client creates, calls, and destroys a number of objects sequentially. A client issues an object creation request to the network of servers and then waits for the network to return a pointer to the created object.

Once the client obtains a pointer to the object, the client can start calling methods of this object. By calling a method, the client remains idle until the server notifies that the call has been completed. A list of possible client's calls is described below:

- *Wait(d)* causes a client to wait for *d* time units;
- *Create(O)* represents a client's request to create an object *O* on a server;
- *Method(t)* is a client's call for a method of the object created on the server;
- *Destroy(O)* represents a client's request to destroy object *O* currently held on that server.

Each client's event starts with a *Wait()* call and terminates after a *Destroy()* call is complete. Figure 2 shows an example of *m* events created by a client. Each client's call has a different number to signify that the parameters to each call are different and unrelated. The first event starts with Wait(d_1) and terminates after Destroy(O_1) call is complete.

Wait(d_1)	Wait before creating an object
Create(O_1)	Create object number one
Method($t_{1.1}$)	1-st method call to object number one
Wait($d_{1.1}$)	Wait between previous and the next call to object number one
…	
Method($t_{1.n}$)	n-th method call to object number one
Wait($d_{1.n}$)	Wait before destroying object number one
Destroy(O_1)	Destroy object number one (after n calls to it)
…	
Wait(d_m)	Wait between destruction of object m-1 and creation of object m
Create(O_m)	Create object number m
…	
Wait($d_{m.k}$)	Wait before destroying object number m
Destroy(O_m)	Destroy object number m (after k calls to it)

Figure 2: The sequence of a client's events as an ordered sequence of calls.

Sequences of calls represent a client's general behavior. However, client's specific behavior is governed by a set of simulation parameters which in effect fine-tune client's events. Most of these parameters are averages, and the actual values used are based on random numbers with specified probability distributions [2].

Simulation parameters governing the generation of clients' events are summarized below:

- *Clients* is the number of clients;
- *Objects/client* is an average number of objects created by a client;
- *Delay between objects* is the average time between an object destruction and creation of another one;
- *Calls/object* is the average number of method calls made to each object;
- *Delay between calls* is the average time delay between calls made to each object;
- *Method time* is the average server processing time for completion of a single method call;
- *Method probability* is the probability that the method associated with this parameter will be called.

A network of servers is described by two parameters:

- *Servers* is the number of servers in a network that can process requests from clients;
- *Node-node delay* is the time for a message to pass from one server to another one.

Load Balancing Techniques

In order to compare and contrast the proposed diffusive load balancing technique with other techniques, the following four load balancing techniques were simulated: static load balancing, round robin balancing, diffusive load balancing, and shortest queue load balancing. In each simulation run, the same behavior of clients is used in order to compare responses of the load balancing techniques.

Static Load Balancing requires no extra hardware or software components to implement. Clients are statically bound to the servers, and do not access any other server. The servers act independently of one another. The only parameter specific to this simulation is *maximum clients* and represents the maximum number of clients.

Using Round Robin Load Balancing a client creates an object by sending a message to the router. After a specified delay, the router forwards the message to one of the servers, in a round robin fashion. The delay associated with the router is *router delay.*

Using Diffusive Load Balancing technique, a client requests object creation by sending a message to a round robin router, which forwards the message to one of the servers. After a method on the object is evoked for the first time, a search for the granting server takes place in the *G(V, E)* representing the *network topology*. Request is moved from one server to its neighboring server, if the difference in workload between the server and its neighbors is higher than the *propagation differential*, a threshold value, which determines whether to forward a server's request to another server, or not. The workload of a server is approximated using the number of client tasks in the ready queue.

In Shortest Queue Load Balancing new client requests are forwarded to the server that has the least workload. This is achieved by having a single entry-point to the network of servers which keeps track of how much processing time is needed by the clients' requests already running on each server. Objects are not bound to a specific server during their lifetimes, and can move from one server to another, which in effect achieves a very fine-grained load balancing. Since it is impossible to determine in advance how much processing time a client request will take, this technique has only theoretical significance. It is added to the simulation as a benchmark to make comparisons to the other simulated load balancing techniques.

Simulation Results and Discussion

The software simulator was designed and implemented to model the four different load-balancing techniques in the three different client-server environments. The comparative study of load balancing techniques was made using values of the simulation parameters shown in Table 1.

The clients' behavior is specified by parameters in columns number one through nine. The standard deviation in row number three is used for determining averages of the first two parameters, whereas the standard deviation in row seven is used to determine averages of the parameters four through six. Network of servers is represented by parameters 10 and 11. To simulate static, and diffusive load balancing techniques, parameters 12 and 13, are used, respectively. In the small-scale environment, a single method time was used with certainty of 100%, whereas, in the intranet and in the Internet environments, two different method times were used each with 50% of their corresponding method calls.

#	Simulation parameters	Small-scale	Intranet	Internet
1	Clients	20	50	100
2	Objects/client	1	5	5
3	Standard deviation	0	2	5
4	Delay between objects	100	10000	20000
5	Calls/objects	15	5	1
6	Delay between calls	5000	1000	100
7	Standard deviation	3	2	10
8	Method time	1000	1000 750	500 750
9	Method probability	100	50 50	50 50
10	Servers	5	5	5
11	Node-node delay	100	100	100
12	Maximum clients	20	55	120
13	Router delay	5	5	5
14	Propagation differential	20	20	20
15	Network topology	G1	G1	G1

Table 1: Simulation parameters and their values used in three different operating environments.

The diffusive load balancing uses parameters 14 and 15. The topology of the network is given in the graph *G1(5, 8)* depicted in Figure 4. There are five servers *V={0, 1, 2, 3, 4}* with 8 point-to-point connections between them. Connections between two pairs of servers (1, 3), and (1, 4) are bi-directional, whereas the other six connections are unidirectional.

$$G1 = \begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{vmatrix}$$

Figure 4: Network topology used in the simulation model.

Behavior of clients is generated randomly for each different operating environment. The same behavior is used in all environmental conditions in order to compare the different load balancing algorithms. The start time, end time, and delay of each client's request submitted to the server are recorded, as well as the workload of each server. Workload of a server is measured in time units, and is defined as the total processing time needed to finish all the clients' requests running on that server. Pertinent data of the simulation run are summarized in Table 2.

For each load balancing algorithm in each environment the data *Te* and *Td* were compiled. *Te* is elapsed time, from the start of the first client call until all the clients' calls in a specified environment have been completed, measured in time units. *Td* is delay time, representing the sum of all the delays associated with the clients' requests, also measured in time units.

In Table 2, static load balancing performs well in the small-scale environment because in this environment a good estimate can be made for maximum number of clients. Round robin load balancing achieves moderate results in all three environments. Diffusive load balancing achieves excellent results in the Internet environment, with an elapsed time almost 25% shorter than round robin, and only 3% longer than shortest queue load balancing, the ideal algorithm. Because of the dynamic nature of this environment, and due to the fact that objects' lifetimes are shorter, diffusive load balancing is able to quickly adapt and balance the load better than the other techniques. As expected, the shortest queue load balancing technique has the shortest elapsed time in all the environments.

Load balancing	**Small-scale**		**Intranet**		**Internet**	
	Te	Td	Te	Td	Te	Td
Static	60406	646479	224925	5409739	204454	4812739
Round robin	66038	668117	210496	5280500	193591	4044099
Diffusive	77415	697549	198188	5256605	155941	3574813
Shortest queue	58305	649692	193338	5143903	150313	3508364

Table 2: Performance summary of simulation results.

Simulation results show that diffusive load balancing is better than round robin and static load balancing in a dynamic environment. However, in a static environment where the number of clients can be known in advance, static load balancing does very well due to its low overhead. Static load balancing is a good choice in such an environment since it is relatively simple to implement. In intranet environment, the round robin load balancing is cheaper and simpler to implement, so the benefits obtained from diffusive load balancing may not justify its usage in this environment.

The difference in performance between diffusive load and round robin load balancing becomes more apparent and in favor of diffusive load in the environments with short objects lifetimes. In such cases, diffusive load balancing is able to dynamically adapt more quickly than in other environments. This adaptability makes it distribute the load more equitably than round robin. On the basis of the performance modeling, it can be concluded that dynamic and adaptable algorithms do significantly better than static load balancing algorithms, which is in agreement with theoretical results obtained by Nelson [3].

Concluding Remarks

This paper examines a new method for improving the performance of a distributed system through load balancing a workload in the distributed client-server architecture. In order to determine the suitability of the new algorithm in different environmental scenarios other known algorithms were used for comparison and contrast.

The evaluation of the diffusive load balancing method is done using simulation runs in which the performance of four different load balancing approaches (diffusive load, static, round robin, and shortest queue) were compared in three different client environments (small-scale LAN, company intranet, and Internet). The shortest queue balancing, being a theoretical technique, is appropriate for benchmarking. Experimental results of simulation run show that the diffusive load balancing is more efficient than static and round robin load balancing in a dynamic environment which manifest in frequent clients' object creation requests and in short objects' lifetimes.

References

[1] W. Winston. *Optimality of the Shortest Line Discipline.* Journal of Applied Probability, 14 (1), 1977.

[2] William Mendenhall, Dennis D. Wackerly, and Richard L. Schaeffer. *Mathematical Statistics with Applications*, 4th edition. Duxbury Press, 1990.

[3] Randolph D. Nelson and Thomas K. Philips. *An Approximation to the Response Time for Shortest Queue Routing. Performance Evaluation Review*, 17(1):181-189, May 1989.

About the Author

Milan E. Soklic is with Software & Electrical Engineering Department at the Monmouth University. His academic and research interests are real-time systems, operating systems, software engineering, and computer systems architecture.