A new fuzzy-decision based load balancing system for distributed object computing

Yu-Kwong Kwok* and Lap-Sun Cheung

Department of Electrical and Electronic Engineering, The University of Hong Kong, Pokfulam Road, Hong Kong, Hong Kong

Received 24 September 2001

Abstract

Distributed object computing systems are widely envisioned to be the desired distributed software development paradigm due to the higher modularity and the capability of handling machine and operating system heterogeneity. Indeed, enabled by the tremendous advancements in processor and networking technologies, complex operations such as object serialization and data marshaling have become very efficient, and thus, distributed object systems are being built for many different applications. However, as the system scales up (e.g., with larger number of server and client objects, and more machines), a judicious load balancing system is required to efficiently distribute the workload (e.g., the queries, messages/objects passing) among the different servers in the system. Unfortunately, in existing distributed object middleware systems, such a load balancing facility does not exist. In this paper, we present the design and implementation of a new dynamic fuzzy-decision-based load balancing system incorporated in a distributed object computing environment. Our proposed approach works by using a fuzzy logic controller which informs a client object to use the most appropriate service such that load balancing among servers is achieved. We have chosen Jini to build our experimental middleware platform, on which our proposed approach as well as other related techniques are implemented and compared. Extensive experiments are conducted to investigate the effectiveness of our fuzzy-decision-based algorithm, which is found to be consistently better than other approaches.

Keywords: Distributed object computing; Load balancing; Fuzzy decision; Java; Jini; Remote method invocation; Middleware

1. Introduction

With the great advancement of hardware technologies, powerful distributed computing systems are becoming ubiquitous. Indeed, with commodity hardware components, a high performance network of PCs can be set up to execute applications developed using new software structuring paradigms, such as object-based systems and object brokerage protocols, which have also advanced tremendously parallel to the development of hardware technologies. Such new distributed software development paradigms, while having the advantages of modularity and capable of handling platform heterogeneity, were conceived as impractical mere 5–10 years ago because many complex operations such as object serialization and data marshaling, were too time consuming to be efficiently run on the hardware platforms. Currently, many commercial software projects are using message-passing distributed object based approaches such as CORBA (common object request broker architecture), DCOM, Java RMI (remote method invocation), and, more recently, Web services. Using a distributed object based approach, an application is constructed as a group of interacting objects. These objects are distributed over multiple machines which interact with each other by passing structured messages (e.g., in the case of Web services, self-describing XML documents) through well-defined protocols (e.g., SOAP in Web services, RMI in Java). Usually, the interactions are queries or remote services invocation [31]. For example, as shown in Fig. 1, a word processing object might interact with a remote printer...
found in a distributed Web-server architecture [3]. DNS based on DNS (domain name system) is commonly used to resolve client requests and route the requests to distributed machines. The load balancing mechanism is limited by the size of the system. For example, in the well-known distributed operating system Amoeba, a new process consults kernel to assign the CPU with the lightest workload. Thus, workload can be shared among the multiple CPUs in the system. However, such a load balancing mechanism is more suitable for the fine-grain tasks assignment rather than the requests workload sharing we consider in our study. Furthermore, such kernel level services are not commonly available in popular desktop OSs such as Windows and Linux, on which our distributed object system is built.

Middleware level approach: Standard infrastructure for distributed object computing architecture, such as CORBA introduced by the Object Management Group, Microsoft’s DCOM, and Sun Microsystems’ RMI have gained wide acceptance in the distributed application market. These middleware systems perform common network programming tasks such as object marshaling/de-marshaling, and fault detection and recovery. Middleware shields the underlying OS from the application, and provides many features through which an application programmer can use it for implementing a load balancing solution. For example, middleware-based load balancing can be applied in CORBA using Object Request Brokers (ORBs). Specifically, ORBs receive client requests and route the requests to distributed objects located in different remote servers. The routing decisions can be based on different load balancing algorithms, which are again commonly round-robin or random approaches, incorporated in ORBs.
In view of the above brief survey, we find that the load balancing requirements and environments are very different from those assumed in classical load balancing algorithms [24]. Most importantly, the existing load balancing algorithms used in distributed object computing systems are usually based on simple techniques such as round-robin or random, which may not give optimized performance. Thus, in our study, we propose, implement, and evaluate a new approach for load balancing in a distributed object system. Specifically, we focus on middleware-based load balancing on a distributed Java network. We incorporate our load balancing scheme in a dynamic network service system called Jini [22]. Our proposed load balancing algorithm employs fuzzy-decision control [23].

An effective load balancing scheme requires the knowledge of the global system state (e.g., workload distribution). However, in a distributed computing system, the global state is swiftly and dynamically changing and it is very difficult to accurately model the system analytically [11, 16]. Thus, in order to tackle the load balancing problem in such an environment where state uncertainty is unavoidable, we employ a fuzzy-decision approach to model those state variables that cause uncertainty in global states. Our approach is novel in that it is seamlessly incorporated into the Jini middleware. Our fuzzy-decision control is also effective and robust, as evident in the experimental results in a real Jini network, due to the concise fuzzy rules set.

This paper is organized as follows. In the next section, we discuss in detail about the dynamic load balancing problem in its classic form and in a modern distributed system environment, the Java-based middleware system, and the requirements of integrating a load balancing service in such a system. In Section 3, we describe our proposed fuzzy-decision-based approach. Section 4 contains the experimental results we obtained in our Jini-based testbed built on a Pentium PCs network. We discuss some other related work in Section 5. The last section concludes this paper.

2. Background

In this section, we describe the components of a general dynamic load balancing algorithm, as commonly used in classical load balancing research [24]. We then describe the target object-based environment, built using Java RMI and Jini middleware, in which our proposed load balancing scheme is incorporated. Afterwards, we delineate the requirements of a dynamic load balancing algorithm to be integrated in such a distributed object system, and describe how such a load balancing algorithm can be integrated in a real environment.

2.1. Dynamic load balancing

The load balancing problem in a distributed computing system has been addressed extensively by researchers for many years [2,4,8,9,13,24,26,34]. Because the load balancing problem is NP-hard [4,15], heuristics are sought to tackle the problem. Researchers have proposed many different kinds of load balancing algorithms which can be mainly divided into two categories: static and dynamic [4,19,24,36]. Dynamic load balancing algorithms use system state information (the load levels at different machines) to improve the quality of load balancing decision, while static algorithms make no use of such information. In static algorithms, decisions are hardwired using a priori knowledge of the system. Since static approaches usually cannot cope with the dynamically changing system conditions, dynamic algorithms are used in practical situations. However, dynamic algorithms must collect, store, and analyze state information, and thus, they incur more overhead than their static counterparts. In our study, we focus on dynamic approaches only.

Simply put, system performance can be improved by relocating tasks from heavily-loaded machines to lightly-loaded ones. Using this simple principle, in the past two decades, researchers have proposed many techniques to allocate jobs dynamically in a LAN to improve performance [4,18,19,36,37]. In general, load balancing algorithm can be analyzed in a framework with four dimensions: selection policy, transfer policy, information policy, and location policy [24,32]. Specifically, information and location policies have the most important roles.

- Location policy: The objective of this policy is to find a suitable transfer partner for a machine, once the transfer policy has decided that the machine is a heavily-loaded state or lightly-loaded one. Common location policies include: random selection, dynamic selection, and state polling [24].
- Information policy: This policy determines when the information about the state of other machines should be collected, from where it has to be collected, and what information is to be collected. Common approaches are: no exchange of states, state probing (or demand-driven) in process of load balancing, periodic exchange (information gathered periodically), state-change broadcasting, and conditional and limited multicasting [24].
- Transfer policy: A transfer policy determines whether a machine is in a suitable state to participate in a task transfer, either as a sender or a receiver. For example, a heavily loaded machine could try to start process migration when its load index exceeds a certain threshold.
Selection policy: This policy determines which task should be transferred. Once the transfer policy decides that a machine is in a heavily-loaded state, the selection policy selects a task for transferring. Selection policies can be categorized into two policies: preemptive and non-preemptive. A preemptive policy selects a partially executed task. As such, a preemptive policy should also transfer the task state which can be very large or complex. Thus, transferring operation is expensive. A non-preemptive policy selects only tasks that have not begun execution and, hence, it does not require transferring the state of task.

2.2. Overview of Java RMI and Jini

Java Remote Invocation (RMI) is a Java enabled extension to the traditional remote procedure call mechanisms. RMI allows not only the data to be passed from object to object around the network but full objects, including the code, can also be transferred. Fig. 2 shows the three independent layers that constitute the RMI system. These three layers are:

- Stub/skeleton layer: The interface between the application layer and the rest of the RMI system.
- Remote reference layer: This layer is responsible for carrying out the semantics of the invocation and sits on top of the low-level transport layer.
- Transport layer: This layer is responsible for the setup and management of the connection and dispatching the requests to the remote objects within transport layer’s address space.

RMI supports dynamic class loading—the classes required to handle method invocation can be dynamically loaded at run-time. Built on top of Java and RMI, Jini [20–22] is a new computing paradigm introduced by Sun Microsystems that can provide a network wide plug-and-play environment. Jini can be used to federate groups of devices and software components into a single but dynamic distributed system. The overall goal is to turn the network into a flexible, easily administered coherent system in which resources can be conveniently located and used by human and computational clients [22].

A Jini system can be divided into three components: infrastructure, programming model, and services. The infrastructure is the set of components that enables building a federated Jini system. The programming model includes a set of interfaces that enables the construction of reliable services such as discover, lookup, leasing, remote events, and transactions. The services are the entities that can be used by a person, a program, or another service within the federation. The infrastructure supports the discovery and join protocol that enables services to discover and register with lookup services. The lookup service is where services advertise their availability so that clients can find and use them.

There may be one or more lookup services running in a network. In discovery and join protocol, discovery is the mechanism by which a service locates lookup service on the network while join is the mechanism by which a resource registers the services it offers with lookup services. Service clients can also use this protocol to locate and contact services. A client first gets the registrar of the lookup service and then queries it to find services available in the Jini network. Once matched, the client can download the proxy of the service from the lookup service and use the proxy to directly communicate with the service. Each proxy object in the lookup service can have service attributes [20,21], which are objects that describe features relevant to distinguishing one service from another in ways that are not reflected by the interfaces supported by the
service. Fig. 3 shows the service item which include service attributes that services stored in the lookup service upon register with it.

Services communicate with each other using RMI. Moreover, the Jini architecture supports distributed events. The Distributed Event and Notification System [22] defines a set of interfaces which allow an object in one Java virtual machine (JVM) to register interest in the occurrence of some event in some other JVM. The former JVM will be notified when the event occurs.

2.3. Load balancing service in a distributed object computing system

Because a distributed object based system is substantially different from a traditional static client–server-based distributed system, the requirements of a load balancing service are also quite different. Given below are the major requirements of a load balancing service in the object-based system considered in our study.

- **Client transparency**: Client programs need not be modified in order to use load balancing service. The client should not be aware of the changes due to incorporation of a new load balancing algorithm in the load balancing service.
- **Avoid changing middleware layer**: Middleware layer should not be modified. Once the middleware layer has been modified, it will become proprietary and incompatible with other existing applications developed using the original middleware.
- **Server transparency**: Like client transparency, server applications need not be changed so as to take the advantage of load balancing service.
- **Scalability and fault-tolerance**: To avoid a single point of failure, several load balancers can coexist in a federation. Load balancers can be configured to work cooperatively to form a single logical load balancing service.
- **Integrating new load balancing algorithms**: The load balancing service should be designed in such a way that different load balancing algorithms can be easily integrated without extensive modification of the source code of the load balancing service, i.e., modular design pattern should be employed to develop the load balancing service.
- **Minimal overhead**: Network overhead caused by load balancing service should be minimized. A load balancing service should be designed such that unnecessary message/object exchanges between the load balancing service and other network components should be avoided; otherwise, the overall system performance cannot be guaranteed.

3. The proposed fuzzy-decision-based load balancing scheme

In this section, the design of our proposed fuzzy-decision-based load balancing service is described. We first give an overview of integrating load balancing in Jini-based middleware environment. We then describe the fuzzy logic controller, which is the core part of the fuzzy load balancing service. The dynamic interactions between the fuzzy load balancing service and other components are discussed in detail in the subsequent subsections.

3.1. Integrating a load balancing system in a Jini environment

In general, Jini application consists of a set of clients requesting a set of services. Services themselves can act as clients of other services. Services can also be grouped together to provide some higher-level functionality. Services are defined via an interface such that the implementation of a proxy supporting the interface that will be seen by the service client will be uploaded into the lookup service by the service provider. The client has to download the service proxy in order to use the service.

The execution of service can be either on the client side or the server side. Proxies can take a number of different forms [22]. A smart proxy is a proxy which consists of a set of local methods and a set of one or more remote object references (stubs). A client can execute local methods to access the method of the remote objects referenced in the proxy. In our study, we assume that clients use smart proxies to communicate with services and the execution of method is on the server side. Furthermore, the set of services executed on the server is assumed to be homogeneous.

In our study, we have considered different approaches to integrate load balancing functionality into a Jini environment.

- **Client-based approach**: A client can use the proxy to monitor the machine states and network delays. The load information obtained from the server can be used to determine whether to continue using the service or switch to another available service. However, the major weakness in this approach is that the message exchanges among each client and server will significantly increase the network traffic. A different client-based approach is to select a service randomly. When a list of services are returned from the lookup service, the client can randomly select one of them. On an average, we can expect the load to be evenly distributed among the servers. However, the overall performance of the system may deviate considerably from the expected load balanced situation.

- **Modified lookup service approach**: Lookup service can be modified in order to perform load balancing.
A modified lookup service can maintain a table which records the most updated load information returned by services registered with it. Lookup service only returns the most lightly loaded service to clients in a round-robin fashion. The major advantage of using this approach is that it enables fine-grained load balancing. The lookup service has full control on which services should be executed. However, the required modification, which is quite substantial, may cause incompatibility problems when the modified lookup service is used together with other Jini clients. Furthermore, as centralized approach is used, lookup service will become a bottleneck and a single point of failure. One possible remedy is to employ several lookup services in the service domain.

**JavaSpace approach:** JavaSpace can be employed to perform load balancing [14]. A client can act as a master and assign task entries to its slaves (services). When the master has a certain number of identical tasks to be done, it can place them into the JavaSpace. Services can cooperate to finish the overall work by randomly retrieving tasks from the JavaSpace. After a service has finished working on a particular task, it can put the result entry back to the JavaSpace so that the master can combine the result with the partial results of other tasks later. Load balancing can be achieved using this approach because the services keep on taking the task entries and writing the result entries in parallel. However, for this approach to be effective, applications should be designed as highly parallel programs so that there are enough independent tasks to be worked on by different servers. Both service and client programs have to be explicitly specified in order to adopt this replicated worker pattern. A Java-based Web server cluster controller has been developed by Concept Technologies [17] where JavaSpace is employed to balance workload among Web servers. One drawback of this approach is that the machine holding the JavaSpace will be overloaded if large number of workers take and write entries into it. In addition, out of memory, exception may occur if the memory is used up by the JavaSpace when too many tasks are inserted.

Thus, the above three approaches all have some deficiencies. We propose to use a dedicated service approach, in which load balancing by itself is also a service and will be invoked by the cooperation between the clients and the servers. This is detailed in the following sections.

### 3.2. Fuzzy logic controller

To tackle the load balancing problem, conventional control theory can be applied to restore system equilibrium. For instance, a sudden increase in client requests can be modeled as external force which attempts to destabilize the network and the end-hosts. The stability of the network can be maintained by using feedback control which performs suitable adjusting actions to minimize the effect of the external force. In order to design a load balancing algorithm based on conventional control theory, one has to develop a mathematical model of the system to be controlled and determine the characteristics of the model by applying various analytical techniques. Indeed, with the incorporation of some simplifying assumptions, a model with linear relationships between a few variables can be easily set up. However, to handle a complex system such as a high speed computer network where a lot of uncertain parameters exist, a model with complex and non-linear relationships between a lot of variables has to be devised. This limitation makes it very difficult, if not intractable, to apply conventional control theory to balance load in the computer network. To overcome this problem, fuzzy logic control theory [23] can be applied instead of the conventional one.

Fuzzy logic control attempts to capture intuition in the form of IF-THEN rules, and conclusions are drawn from these rules [23]. Based on both intuitive and expert knowledge, system parameters can be modeled as linguistic variables and their corresponding membership functions can be designed. Thus, nonlinear system with great complexity and uncertainty can be effectively controlled based on fuzzy rules without dealing with complex, uncertain, and error-prone mathematical models [23].

The architecture of the fuzzy logic controller shown in Fig. 4 includes five components: Fuzzifier, Rule Base, Membership functions, Fuzzy Inference Engine, and Defuzzifier. The fuzzifier is the input interface which maps a numeric input to a fuzzy set so that it can be matched with the premises of the fuzzy rules defined in the application-specific rule base. The rule base contains a set of fuzzy IF-THEN rules which defines the actions of the controller in terms of linguistic variables and membership functions of linguistic terms. The fuzzy inference engine applies the inference mechanism to the set of rules in the fuzzy rule base to produce a fuzzy set output. This involves matching the input fuzzy set with the premises of the rules, activation of the rules to deduce the conclusion of each rule that is fired, and combination of all activated conclusions using fuzzy set union to generate fuzzy set output. The defuzzifier is an output mapping which converts fuzzy set output to a...
crisp output. Based on the crisp output, the fuzzy logic controller can drive the system under control.

The fuzzy rule base contains a set of linguistic rules. These linguistic rules are expressed using linguistic values and linguistic variables. Different linguistic values can be assigned to a linguistic variable. For instance, very long or moderately short can be used in the variable remote method invocation time. These linguistic values are modeled as fuzzy sets. Based on the linguistic values, their corresponding membership functions can be expressed based on application requirements.

In the Jini computing model, client objects basically have no idea on which service they should send requests to in order to achieve the best QoS (quality of service) or more specifically, the shortest response time. Generally, a request router or load balancer can be implemented to route client requests to the most appropriate service. The request router can make such decisions based on the current state of server objects. However, such state information may not be updated and reliable as client requests reach servers [28]. That is, the state information cannot reflect the state of servers accurately due to network delay. The request router needs to use approximate reasoning to handle the fuzzy information so as to make the system efficient. In order to make a correct routing decision, linguistic variables, server load, remote method invocation time, and service rank, are used in the fuzzy logic algorithm and are defined as follows.

3.3. Server load

We define server load, denoted as SL, with the fuzzy set definition: \{low (L), medium (M), high (H)\}. Accurate estimate of load is notoriously difficult to obtain [25,27]. We employ an indirect approach in determining SL. Instead of directly measuring each process execution time, we measure the execution time of a benchmark program which consists of several benchmark kernel loops. The benchmark program runs perpetually without stopping in the system as a background process. By observing the running times of the benchmark program, we can infer the instantaneous load level in the system. Fig. 5 shows the membership graph for SL.

3.4. Remote method invocation time

In order to measure the responsive of the servers to client requests as well as the overhead introduced into the network by distributed processing (i.e., message/object exchanges), network utilization needs to be determined. This is done by measuring the time for remote method invocation (RMI). We define a benchmark remote method which simply returns a primitive data type from server to our fuzzy logic load balancing service to measure the RMI time:

```java
int n; public int getNumber() {return n;}
```

The method `System.currentTimeMillis()` is used to measure the time elapsed during remote method invocation in milliseconds. In our measurements, it is found that the time needed to execute the remote method is about 2–3 ms when network utilization is low and server is lightly loaded. The time becomes longer when the system load increases. Thus the benchmark remote method can approximately reflect the network load and the responsiveness of the servers. The fuzzy set of remote method invocation time (RMIT) is defined as: \{short (S), medium (M), long (L)\}. Fig. 6 shows the membership graph for RMIT.

3.5. Service rank

We use service rank (SR) to classify services into six different categories. The fuzzy set of SR is: \{very low (VL), low (L), medium low (ML), medium (M), medium high (MH), high (H)\}. The membership graph of service rank is shown in Fig. 7. The higher the rank that a service gets, the more appropriate it can accept client
requests. After defining the above fuzzy variables, a set of inference rules is defined as shown in Table 1.

By applying the fuzzy inference rules in Table 1, a decision can be generated based on both antecedents. That is, if RMIT is short and SL is low, then SR is high. Having these fuzzy inference rules and membership graphs, the fuzzification and defuzzification processes can be carried out as follows. First, the input values of RMIT and SL are mapped to their respective membership degree values on their membership graphs. These degree values are compared and the minimum of the two is then projected onto the membership function of their consequence graph. The output graph, usually in the shape of a trapezium [23], then represents the output of one inference rule. After the output graph is generated, defuzzification of the fuzzy output into a crisp or numeric value can be carried out. We used the centroid method [23] to defuzzify the output. The overall centroid of overlapping areas $A_i$ for $i=1,2,\ldots,N$ is given by:

$$\bar{x} = \frac{\sum_{i=1}^{N} \bar{x}_i A_i}{\sum_{i=1}^{N} A_i}$$

where $A_i$ and $\bar{x}_i$ are the overlapped area and centroid from the triangles or trapeziums obtained in the $i$th rule, respectively. The centroid and area are calculated for each triangle or trapezium. This process is repeated for other inference rules where the inputs are applied to obtain an area composed of overlapped trapeziums. The defuzzification process generates a centroid value which represents the rank of a service. Fuzzy logic load balancing service can then route client requests to the best service based on the rank obtained. The overall operating sequence of the fuzzy logic controller can be represented by the fuzzy state pattern shown in Fig. 8.

It has been noted that the herd effect [29] may occur if the load balancer immediately forwards all requests to a server that is assigned to the highest service rank. It will lead to a sudden degradation in the overall system performance. In order to minimize such effect, the fuzzy logic load balancing service schedules client requests to servers based on the following prioritized round-robin algorithm.

---

Table 1

<table>
<thead>
<tr>
<th>RMIT</th>
<th>SL</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>M</td>
<td>MH</td>
</tr>
<tr>
<td>L</td>
<td>M</td>
</tr>
</tbody>
</table>

---

Fig. 7. The membership graph for service rank (SR).

Fig. 8. The state pattern.
Prioritized round-robin algorithm:

Variables:

- BS: the best server with the highest service rank
- NS: the server next to BS in the round-robin queue
- PS: the previous server to which last client request is sent

(1) While (true) {
(2)   if BS equals to PS
(3)     Load balancer sends request to NS;
(4)   else
(5)     Load balancer sends request to BS;
(6)   Set PS = BS;
(7) }

In our implementation using the Jini platform, the system consists of a lookup service, a fuzzy logic load balancing service, several server objects (providing services) and client objects. In the following, we discuss the structure and the function of each component.

3.6. Components residing in the server host

Each standard Jini service consists of basic functions such as finding lookup service and registering to it. These actions can be accomplished by installing a JoinManager, which is a standard Jini feature, into the service. Fig. 9 shows the services residing in the server host.

In our implementation, there is a general service in which clients are interested. The service consists of a JoinManager, Remote Service Object and Remote Service Admin Object. Remote Service Object is a remote object which implements the remote interface that is known to the client. The client can invoke methods on service object based on the remote interface.

Remote Service Admin Object provides an interface so that the service attributes can be modified by other administrative components in the network.

We also incorporate a load monitor service in the server host. Like other Jini services, load monitor service registers itself with the lookup service with the help of JoinManager. The remote load monitor service object is being used when the fuzzy logic load balancing service registers as a listener for the remote event generated by load monitor service. The event consists of information such as server location and server load.

3.7. Components residing in the client host

Basically, a client locates lookup service and gets the services proxies available in the lookup service. Then the client, based on the service interface, generates a request to fuzzy logic load balancing service and gets the result after the service has finished the execution. The structure of the client is shown in Fig. 10. The client consists of a LookupCache which is created by ServiceDiscoveryManager. Inside the cache, it stores the fuzzy logic load balancing service proxy to which the client can send request. ServiceDiscoveryManager is a standard Jini feature of which the main use is to help clients to locate services and cache service proxies.

3.8. Components residing in the fuzzy logic load balancing service host

Fuzzy logic load balancing service is the core part of the system. Its main function is to analyze information passed from the load monitor, and then make a decision to forward client request to appropriate server. The structure of the fuzzy logic load balancing service is shown in Fig. 11.

The fuzzy logic load balancing service performs actions such as registering itself to the lookup service and obtaining service proxies available in the lookup
service. Thus, it acts as both a server and a client as it consists of both the JoinManager and ServiceDiscoveryManager. Our proposed fuzzy logic load balancing algorithm is implemented in the fuzzy logic controller.

3.9. Deployment scenario of the fuzzy logic load balancing service

Fig. 12 shows the component diagram of the fuzzy logic load balancing system. Each component interacts with other components located in other machines. In the following, each of these components and their interactions are described.

3.9.1. Load monitor

Fuzzy logic load balancing service has to collect information such as server load for analysis in order to make a routing decision. Thus, it has to register with a load monitor so as to receive this information. Load monitor measures the load on a server and reports various server loads to a fuzzy load balancer in the form of an asynchronous distributed event. It also provides an interface to the fuzzy load balancer to measure remote method invocation time from fuzzy load balancer to server.

Fig. 13 shows the interaction between the fuzzy logic load balancing service and the load monitor. First, a fuzzy logic load balancing service (event consumer) registers with a load monitor (event producer). Instead of directly contacting the load monitor backend object, the fuzzy logic load balancing service performs registration by contacting a load monitor proxy, which will in turn contact the backend load monitor object as illustrated in Fig. 12. Second, the load monitor estimates server load by measuring execution time of a benchmark program which runs as a background process. Third, after measuring a load value, the load monitor sends a notification to the load balancing service asynchronously. Asynchronous notification means that notifications of the server load in the server are not sent at scheduled times with respect to the normal flow of programming logic of the fuzzy logic load balancing service. Note that instead of adopting polling policy in which the load balancer periodically queries server load from the load monitor, pushing policy in which the load monitor pushes measured load value to the load balancer is used.

3.9.2. Load monitor proxy

Load monitor proxy is a Java object, which represents the load monitor residing in the server. Fuzzy load balancing service downloads the load monitor proxy from lookup service. Then it communicates with the remote load monitor object through the load monitor proxy. The load monitor proxy provides an interface which contains all required method signatures to the fuzzy logic load balancing service. Based on the method signatures, the load balancing service can measure remote method invocation time from the load balancer.
to all the servers and register itself with the remote load monitor to receive server load event.

3.9.3. Service proxy
The fuzzy logic load balancing service communicates with each backend service through the corresponding service proxy. Each service has its own unique proxy. All the service proxies are cached from the lookup service to the fuzzy logic load balancing service by means of ServiceDiscoveryManager. The fuzzy logic load balancing service can then select the desired service proxy to which client request should be sent. The required arguments or parameters will be sent to the remote service implementation based on a predefined backend communication protocol.

3.9.4. Fuzzy logic load balancing service proxy
This proxy provides an interface for clients to send requests to the fuzzy logic load balancing service. The content of the interface includes the method signatures of the service in which the clients are interested. A client can execute remote method calls based on the interface provided by the proxy. The method calls are sent to the fuzzy logic load balancing service for routing to the appropriate server as if they are directly forwarded to the required backend service object from the point of view of the client. This ensures client transparency.

3.9.5. The dynamic interactions in fuzzy logic load balancing system
To summarize, the dynamic interactions of the fuzzy logic load balancing system are shown in Fig. 14. The interactions are as follows:
1. A client obtains a fuzzy logic load balancing service proxy and sends request to the service.
2. Load monitor asynchronously sends information to fuzzy logic controller for analysis.
3. Fuzzy logic load balancing service periodically measures the remote method invocation time from the load balancer to the server.
4. When a new piece of information such as remote method invocation time or server load arrive at the fuzzy logic controller, the fuzzy inference engine will start to analyze and assign ranks to different servers.
5. After determining which server is the appropriate candidate, the fuzzy logic controller forwards the client request to that server.

4. Performance results
To evaluate our approach, we have implemented a distributed object platform using Jini and experiments were performed to analyze the client response time and throughput of different load balancing schemes. In order to simulate real client access patterns, a request sequence was generated by using a random number generator to place requests in a given time interval. The request sequences consist of request bursts and intervals of silence. For comparison, we also implemented a system with load balancing algorithm using random and round-robin load distribution by incorporating these algorithms into our fuzzy logic load balancing service.

4.1. Hardware/software platform
We have set up the testing environment as shown in Fig. 15. Six machines are assigned as Jini services and two others are assigned as Jini clients. All the machines are connected by an Ethernet hub with a bandwidth of 10 Mbps. The configuration of six server machines are: (1) two 500 MHz CPU Intel Pentium III workstations, (2) two 667 MHz CPU Intel Pentium III workstations, and (3) two 450 MHz CPU Intel Pentium III workstations. All the machines are equipped with 128 MB memory. We have another two machines, which are 600 MHz CPU Intel Pentium III workstations with 128 MB memory, holding lookup services and fuzzy logic load balancing service. The client machines we used are all 200 MHz CPU Pentium with 64 MB memory. All the machines are running Red Hat Linux 7.0 as their operating systems. Java Development Kit version 1.3 and Jini Technology Starter Kit 1.1 are used to develop all system components. A stateless service, Fibonacci function, is chosen as our benchmark program to simulate consumption of CPU clock cycle in the server machines. Fibonacci function provides a
suitable workload for our load balancing tests since each operation can run for a relatively longtime.

4.2. Average client response time and throughput comparison

The average client response times of the three load balancing algorithms as a function of the number of servers are shown in Fig. 16, which illustrates that the fuzzy-based approach outperforms the other algorithms consistently for different number of servers. The average client response time of random load balancing algorithm is the highest in all the cases because uneven distribution of load exists in the random load balancing algorithm. A server with less computing power causes a higher response time when it is suddenly overloaded. This effect deteriorates the overall performance and causes the highest response time.

Fig. 17 shows how the average throughput differs between each load balancing strategy. In this measurement, 5–20 clients were used and each client generated 50 requests. Each client request will generate a computational task using Fibonacci function. The experiment is repeated 100 times for different number of servers. As can be seen from Fig. 17, the throughput increases as the number of servers increases. Again, the throughput of random load balancing algorithm is the worst among the three algorithms due to the fact that overloaded computing machines will lengthen the completion time of a task and thus, reducing the overall throughput. For throughput-sensitive application, random load balancing algorithm is not suitable. On the other hand, the throughput of fuzzy-based approach performs the best among the three. The reason is that our approach assigns more requests to the machines with better performance based on fuzzy analysis. This significantly reduces the completion time of a task. This observation will be further illustrated in the following subsections.

4.3. Load distribution

In this section, load distribution of the three load balancing strategies are described. As mentioned previously, instead of directly measuring each process execution time, we measure the execution time of a benchmark program to infer the instantaneous load level in the system. It is obvious that the execution time of the benchmark program is longer in the machine with smaller computing power. Moreover, the execution time of the benchmark program will be prolonged if the machine handles multiple requests simultaneously. In the following, we will discuss the load distribution scenario of different load balancing schemes on five and six servers. We have performed an extensive testing with different combinations of numbers of clients and servers, which vary from 5 to 20 and 3 to 6, respectively. These results can be found from a detailed report [5].

4.3.1. Round-robin load distribution

In round-robin load distribution approach, requests are distributed to server machines in a round-robin fashion. The results are shown in Fig. 18. Initially, the execution times of the benchmark program are about

Fig. 16. Average response times of the clients.
500 ms on the PIII 450 MHz machine, 200 ms on the PIII 500 MHz and PIII 667 MHz machines as there is no client request. Then, client requests are distributed to the five servers in a round-robin fashion. Each server handles the same amount of requests. It can be seen that the execution time of the benchmark program increases drastically for Server 5 (PIII 450 MHz) whereas Server 3 and 4 (PIII 667 MHz) sustain a relatively lower value. The execution time of the benchmark program and the completion time of the client request in Server 3 and 4 is much shorter than Server 5. This illustrates that Server 3 and 4 are capable of handling more requests to improve overall system performance. Fig. 18(b) shows the scenario when one extra server, Server 6 (PIII 450 MHz) is added to the experiment. The load level of all servers decreases as the extra server handles a portion of the client request. In order to benefit from round-robin load scheduling, homogenous machines with the same processing power should be used; otherwise, a machine with less processing power will degrade overall system performance.

4.3.2. Random load distribution

In random load distribution approach, requests are distributed to server machines by the fuzzy-load-balancing service randomly. As can be seen in Fig. 19, there is an uneven distribution of load due to herd effect. Some machines are overloaded while the others are only lightly loaded. This effect causes the random load balancing algorithm to behave worse in performance analysis of average client response time and throughput.

Fig. 17. Average throughput of the clients.

Fig. 18. Load distribution of round-robin with 20 clients.

Fig. 19(b) shows the random distribution with 6 servers. The situation remains the same even if one extra server is added to the experiment.
4.3.3. Fuzzy-based load distribution

In our approach, requests are distributed to server machine by using the fuzzy logic load balancing algorithm equipped with the prioritized round-robin algorithm. As shown in Fig. 20, initially the benchmark readings of all servers remain steady, as there is no client request. At point A, we assign a burst of client requests to Server 1 which causes a sudden jump in the benchmark reading. The effect is remedied as the fuzzy logic load balancing service routes further client requests to other servers. Similarly, a burst of client requests is assigned to Server 2 at point B. The fuzzy logic controller again successfully balances the workload. It is noted that all the benchmark readings fluctuate within the range of 400–1200 ms. Compared with the roundrobin scheme, Server 5 (450 MHz) can now maintain a load value which is similar to other servers rather than sustain a higher load value. Fig. 20(b) shows the fuzzy-based load distribution with 6 servers. The six servers can maintain a similar load range because the more powerful machines can serve the more client requests. The machines with less processing power can now serve client requests more efficiently and shorten the overall average client response time.

5. Related work using fuzzy-logic-based approaches

There have been some recent attempts in using fuzzy-based approaches for load balancing [1,6,7,10,12,30,33,35], but those approaches are either too restrictive or not suitable for a middleware-based environment considered in our study. Shaout et al. [30] present a batch job scheduler which schedules jobs in a distributed processing environment based on fuzzy logic. The scheduler balances the load on individual processors and maximizes the throughput of the overall system. The scheme works fine in the mentioned traditional distributed processing environment. However, the performance of the system in an object-based configuration has not been examined. Chulhye et al. [7] propose an approach to characterize the global state uncertainty in a large distributed system in terms of the fuzzy set theory, and present a fuzzy-based distributed load balancing algorithm that explicitly reflects the effect of the uncertainty in the decision-making process. However, the effectiveness of their algorithm in real distributed object system has yet to be determined. Damiani [10] proposes a fuzzy load distribution algorithm which is designed for object-oriented servers in the framework of the CORBA reference model. However, no comparison is made against other load balancing algorithms. The experiment results only show the distribution of requests which is insufficient to show the usefulness of the algorithm.

Wong [35] also uses fuzzy logic to make routing request decisions in client/server architecture. In his algorithm, fuzzy logic is employed to estimate machine states based on past information. Mathematical examples rather than experiment results are used to demonstrate...
the algorithm. The usefulness of such an algorithm in a real environment is unknown. In [33], Unger et al. present a load sharing heuristic for distributed computing on workstation clusters. The approach uses a fuzzy logic controller to make tasks placement decision. They show the implementation of such an algorithm based on a message passing system. However, such an approach requires incorporating many specialized system components to the system which may consume extra system resources. Dierkes [12] uses fuzzy logic to estimate the effects of using different load balancing algorithms in certain situations. Based on the analysis, a suitable load balancing algorithm can be chosen. However, expert knowledge is required in manually setting a fuzzy rule so as to choose a suitable algorithm otherwise unpredictable results may appear. In [6], Cheong et al. proposed a generalized dimension exchange (GDE) method which combines the fuzzy approach and a genetic algorithm to equalize workload among servers. However, the performance of the system in a large-scale computing environment has not been shown.

6. Conclusions

Load balancing is an old problem. But new solutions are required in modern distributed object computing platforms, which are increasingly being used in developing many commercially distributed applications. Web serving is the most notable example of those [3]. In this paper, we describe the design, implementation, and evaluation of our proposed fuzzy-decision-based load balancing algorithm incorporated in a distributed object middleware based on the Jini platform. Our fuzzy-decision load balancer is motivated by the fact that classical and recent load balancing algorithms are inadequate for use in the target platforms considered in our study because there a multitude of new requirements exists. Our fuzzy-decision load balancing service, based on concise and rather easy-to-implement rules, is found to be very effective in our extensive experimental studies using a real Jini-based testbed. Nevertheless, further work is required to investigate the effectiveness of incorporating fuzzy-decision in other Java-based (e.g., JavaSpaces [14]) and CORBA-based approaches.

Acknowledgments

The authors would like to thank Professor Kai Hwang for his valuable suggestions.

References


