Integrating Trading and Load Balancing for Efficient Management of Services in Distributed Systems[*]

Dirk Thißen[1] and Helmut Neukirchen[2]

[1]Aachen University of Technology, Department of Computer Science, Informatik IV
Ahornstr. 55, D-52074 Aachen, Germany
thissen@i4.informatik.rwth-aachen.de

[2]Medical University of Lübeck, Institute for Telematics
Ratzeburger Allee 160, D-23538 Lübeck, Germany
neukirchen@itm.mu-luebeck.de

Abstract. Due to the requirements of open service markets, the structure of networks and application systems is changing. To handle the evolving complex distributed systems, new concepts for an efficient management of such systems have to be developed. Focussing on the service level, examples for existing concepts are trading, to find services in a distributed environment, and load balancing, to avoid performance bottlenecks in service provision. This paper deals with the integration of a trader and a load balancer. The allocation of client requests to suitable servers is adaptable depending on the current system usage and thus the quality of the services used is increased in terms of performance. The approach used is independent of the servers' characteristics, as no provision of additional service properties to cover load aspects is necessary for the servers involved. Furthermore, it may be flexibly enhanced, as the concept of 'load' used can be varied without modification of trader or load balancer. This approach was implemented and evaluated in several scenarios.

1 Introduction

The integration of small isolated networks into bigger ones, distributed all over the world, caused a change in the design of application systems. Single applications are increasingly becoming distributed, thus can use resources more efficient and enable more flexible team work. The distribution of subtasks to different nodes enables an increase in a system's performance. By using middleware concepts, e.g. the *Common Object Request Broker Architecture (CORBA)*, it is possible to create an open service market. Different application objects, which provide services with known service types, can be used to compose the functionality of a new application. A trader supports the search for objects providing a required service. If the same service can be provided by several objects, it is possible to distribute the requests among them. Nevertheless, a system can by partly overloaded, if one resource is used intensively. Here, the concept of load balancing can be used to distribute tasks uniformly to available servers.

[*] This work has been funded by the German Research Council (DFG) under SFB 476.

This paper addresses the integration of a load balancing component into a trader. This enhancement enables the trader to consider performance aspects when selecting a server. The enhanced trader selects a service which is optimal in two aspects. First, the service quality determined by the service properties is taken into account. Second, the load of the corresponding server is considered. The concept was implemented and evaluated on a CORBA basis. The concepts of trading, load balancing, and a combination of both are briefly explained in chapter two, and related work is presented. Chapter three introduces our approach. In chapter four, some analyses show the usefulness of our approach. Finally, chapter five concludes the paper and addresses some perspectives for further work.

2 Trading and Load Balancing

If the same service is offered by several servers, a client will have to be supported in choosing one of them. Two powerful mechanisms are *trading* and *load balancing*. Whereas in trading the choice is driven by a client's requirements, the load balancing approach applies on the server side.

2.1 Trading

The trading service can be seen as an enhancement of the naming service, which gives a client more flexibility in specifying the service it needs. Whereas with the naming service a server and the service this server offers, respectively, must be assigned a unique name, the trading concept describes a service by a service type and service properties. The *service type* defines the functionality of a service. This type contains the *interface type* definition of a service, thus enabling an open service market. Yet, the interface type is not sufficient to describe a particular service. Therefore, the service type also includes a set of *service properties* which can be used to describe non-computational aspects of a service. A server which wants to make a service available in a distributed system passes a description of this service in terms of a service type instance, i.e. the object reference and the corresponding service property values, to the trader. Such a server is called exporter. It has to be considered that some service properties are dynamic, i.e. their values are varying over time. For these properties, the values are not registered at service export time, but when a service is requested. If an importer, i.e. a client searching for a service, contacts the trader it has to specify the service type, too. It can formulate restrictions on the service properties to express its needs, as well as criteria to determine an order on the services found, for example a minimisation of a property's value. The trader matches the importer's description against all recorded services and returns a list of server references. Traders were implemented in various environments [4], but they never became very popular. However, with the adoption of trading as a CORBAservice it became of general interest. Today, many trader implementations exist for CORBA platforms.

2.2 Load Balancing

Load balancing aims at a uniform utilisation of all available resources in a system by distributing tasks according to a given strategy. An optimal strategy would achieve a minimal response time of the system and the servers involved. Simple strategies are *static*; usually, such strategies are referred to as load sharing. New tasks are distributed to the available servers using a fixed schema. Strategies of this category include the cyclic assignment of tasks to the given servers or random server choice. They are easy to implement, but cannot adapt to special situations. The more promising strategies adapt to a system's load, and can react on changes in the system. Tasks are distributed according to the load of the available servers. Such d*ynamic* load balancing strategies have to consider the fact that measured load values only reflect the past. By updating the load values more often this problem could be minimised, but the network load for transmitting load information to the load balancer would increase too much. A compromise between communication overhead and relevance of the data has to be made. Furthermore, *semi-dynamic* strategies are possible, which use the load balancer's knowledge about the past distributions, but not the current servers' states. Much work has been done mostly on homogeneous systems. For example, [2] came to the result that simple strategies with small communication effort are most suitable. Strategies which were based on a threshold showed the best results. In these strategies servers are randomly chosen for the next task if their load is lower than the threshold. More recent strategies use more complex techniques, for example fuzzy decision theory [1]. These works contradict the advantage of simple strategies. On the other hand, [3] affirms the earlier results on simple strategies.

2.3 Combining a Trader and a Load Balancer

Trading is a good concept to support the binding between clients and servers in large open systems. But if one service type is searched for frequently, and each client wants the 'best' service instance, single servers can be overloaded. On the other hand, a load balancer tries to realise a perfect distribution of the clients' requests to the available servers. Yet, it can only select one server in a particular group; in large open systems, where many services with different types exist, the load balancer would have to know the type of the service to select a server. Additionally, the load balancer can only select a server by its load, not by considering service properties. Thus, a combination of trader and load balancer seems to be a suitable solution for a load-oriented assignment of clients to servers in a distributed environment. The direct approach would be the usage of *load values* as *dynamic service properties*. The trader could do a load distribution based on these attributes. Yet, it could be hard or even impossible for a service provider to provide an additional interface where the trader could request the information for dynamic attributes, especially in cases where legacy applications are used. Furthermore, this concept is inflexible, as a more differentiated interpretation of the 'load' value, also taking into account other load information aspects, would be hard.

Although both trading and load balancing are current research topics in distributed systems, only little work has gone into combining these approaches. The first were [8] and [11], which did simulations for the usage of a fair service selection strategy which considers global system aspects when mediating a service. Such a strategy does not guarantee an optimal selection for each client, but tries to optimise the global behaviour of a system. They confirm the results of earlier work regarding the usage of simple strategies. Additionally, [11] mention the risk of an oscillating overload of single servers. As a solution, a dynamic strategy with a random component is proposed. A main topic in [11] is the use of the knowledge of a server's load a trader has from past service mediation. It was shown that a cyclic assignment of clients to the servers, and using load balancing, are similar with respect to the quality of load distribution. Yet, this approach is based on knowledge of the service time for each task. An approximation of this time, considering the server performance and the service type, is not feasible in heterogeneous and open systems. Furthermore, it has to be considered that not every service utilisation is arranged by the trader; on each host, there will be a load independent from the trader's activities.

In [5], the co-operation of a trader and a management system is discussed. On each host a management agent gets management information from the components hosted locally. A trader is implemented on top of the management system and uses its functions to request the values of dynamic attributes or a server's load. Dynamic attributes are obtained by mapping management attributes onto service properties. By specifying complex selection and optimisation criteria for a service selection, a load distribution can be made. Disadvantages of this approach include the dependency on the management system and the lack of a special load balancing component. Furthermore, a load distribution is not transparent for a user, who must specify special optimisation criteria. In [10], an integration of load balancing for middleware platforms with an interface definition language is proposed for the distribution of load in a heterogeneous distributed system. The stub generated from the interface definition is enhanced by a sensor component. This sensor transmits load information to a local load balancer, which propagates all information to the other local balancing components to achieve an overall view of the system load. A naming service uses a local load balancer if a client requests a service. This concept has some disadvantages, too. A naming service, as opposed to a trader, is used here, which is a less powerful approach. The stub manipulation is inadequate because the source code must be known, and no transparent integration into servers is achieved.

3 Architecture of the Enhanced Trading System

For our enhancement of a trader with a load balancing mechanism we specified some design issues [9]. It must be possible to *use the trader in the normal way*, without making a load distribution, as well as to *combine the ordering of the services made by the client's constraints with the ordering of servers with respect to their load*. The load distribution process has to be *transparent for the user*, but he should have the option to *influence the process*, e.g. by defining special service properties. Such properties could refer to the information if load balancing should be performed at all, or which influence the load parameter should have compared to the service quality. The load balancer should be integrated into the trader to achieve a *synergy effect* by exchanging knowledge between trader and load balancer. Furthermore, the load balancer should be flexible to enable the use of *several load balancing strategies and load meanings*.

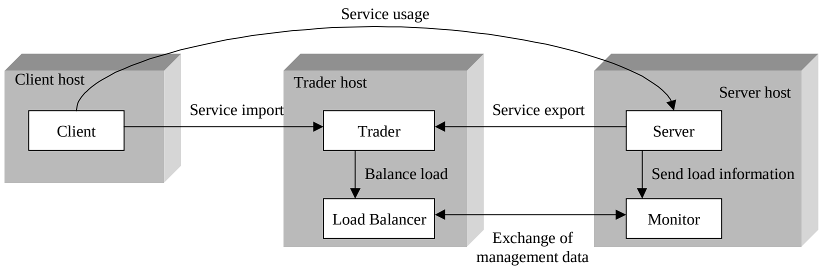


Fig. 1. Architecture of the trader-load balancer system

The architecture of the enhanced trading system is shown in figure 1. On each server host a monitor is installed to observe all local servers. The monitor is connected to the load balancer, which is located on the trader host. A client imports a service from the trader and uses the selected server. The work described in the following is based on a trader implementation done in our department using IONA's middleware platform Orbix 2.3.

3.1 Sensors

Service usage can be determined using a variety of metrics, e.g. the CPU load, the network load, or the load caused by i/o operations. Initially, we only considered the CPU load. To determine this load, the servers' queue length, the service time, and the request arrival rate can be used. Each participating server is enhanced by a sensor which collects these information and sends them to a monitor. As most applications used in our scenario are legacy applications, management wrappers were constructed to enhance an application with the necessary functionality [7]. As load information we use the service time in real time, the service time in process time, the usable CPU performance, and the queue length. The load information is passed to the monitor as a struct `LoadType`, see figure 2. This format is used to transfer load information in the whole system.

```
interface loadbalancing_types {

   enum LoadmetricType {SERVICETIME_REALTIME, SERVICE-
                 TIME_PROCESSTIME,
                    PROCESSTIME_REALTIME_RATIO, QUEUELENGTH,
                    ESTIMATED_TIME_TO_WORK, ON_IDLE, REQUEST_RATE,
                    USAGE_COUNT, HOST_LOAD, UNVALID };

   struct LoadType {LoadmetricType loadmetric; float loadvalue };

};
```

Fig. 2. Structure for covering load information

3.2 Monitor

A monitor manages a local management information base of load information and enables the load balancer to access it. It has a list containing all hosted servers together with their load. As the usage of different load metrics should be possible, all load information, which are transmitted by a sensor, are stored. The monitor not only stores the received load values, but also calculates additional, more “intelligent” values. This includes the computation of a floating average value for the load values mentioned above as well as an estimation of the time to process all requests in a server’s queue. This estimation uses the mean service time of the past service usages and the time for the current request to estimate the time the server has to work on all of its requests. As no outdated load information should be used by the load balancer, a monitor uses a caching strategy to update the load balancer’s information at the end of each service usage. Some values, e.g. the queue length and the estimation of the time to work, are also sent upon each start of a service use. Based on the access and change rates for load values, a dynamic switch between caching and polling is possible. This mechanism is shared by load balancer and monitor. In case of the polling strategy, the monitor knows about access and change rates, thus it can switch to the caching mechanism. On the other hand, if caching is used, the load balancer has both information, and can switch to the polling mechanism.

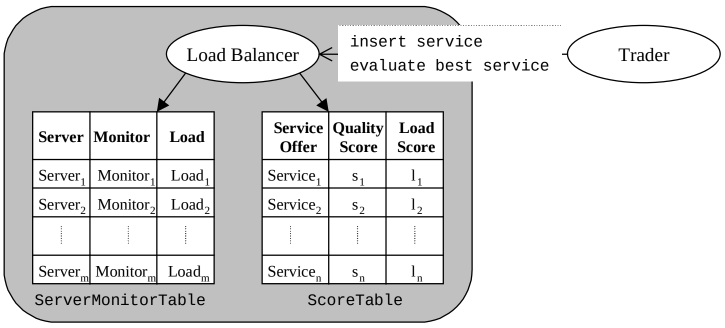
3.3 Trader

Based on a client’s service specification, the trader searches its service directory. Services fulfilling the specification are stored in a result list. In a common trader, this list is sorted relating to the constraints the client has defined, i.e. the most suitable service is the first in the list. This process can be interpreted as a sorting of the services according to the degree of meeting the client’s quality demands. For the integration of a load balancer this sorting is not sufficient, because the servers’ load must also have an influence on this order. Thus, we had to introduce some modifications to our trader. When a new entry is added to the result list, the trader informs the load balancer about the corresponding server. Yet, the load balancer only knows about load aspects, thus it can not do the sorting relating to the client’s constraints. To enable the consideration of both the trader’s sorting and load aspects, the trader must assign a score characterising the degree to which the client’s requirements are met by each service offer. To obtain such a score, a method as that proposed in [6] can be used. When the trader informs the load balancer about a server the quality score for its service offer is computed and also passed to the load balancer. After searching the whole service directory, the trader calls the load balancer to evaluate the most suitable service offer instead of sorting the result list relating to the client’s constraints. To influence the evaluation, information about the client’s weights regarding quality score and load is passed to the load balancer as well as the metric to combine both values. The trader is returned an index identifying a service offer in its list. The object reference identifying this offer is returned to the client.

In addition to the load balancer's mechanisms the trader implements a *random strategy* to determine an order for the services found. This can be seen as a static load balancing strategy.

3.4 Load Balancer

As shown in figure 3, the load balancer manages two tables. The first table contains the management information for the known servers (`ServerMonitorTable`). In this table, each server in the system is listed together with the monitor responsible for measuring the load, and the load itself. The other table, `ScoreTable`, is created when the trader receives a service request. Each service offer found by the trader for this request is recorded in the table together with the trader-computed quality score.



Server	**Monitor**	**Load**
$Server_1$	$Monitor_1$	$Load_1$
$Server_2$	$Monitor_2$	$Load_2$
⋮	⋮	⋮
$Server_m$	$Monitor_m$	$Load_m$

Service Offer	**Quality Score**	**Load Score**
$Service_1$	s_1	l_1
$Service_2$	s_2	l_2
⋮	⋮	⋮
$Service_n$	s_n	l_n

Fig. 3. Structure of the load balancer

After the trader has searched the whole service directory, the load balancing process begins. The approach chosen here consists of two steps. First, the load for all recorded servers is obtained from the `ServerMonitorTable` and inserted into the `ScoreTable`. Getting the load for all service offers at this time implies that no old load information is used. It also has to be mentioned that the 'load' field in the `ServerMonitorTable` does not contain a single value, but a set of load values for all different load balancing strategies. At this time, three strategies which try to minimise the system's load regarding to a particular load metric are implemented:

- *Usage_Count* only counts the number of requests mediated by the trader to a server in the past.
- *Queuelength* considers the current number of requests in a server's queue.
- *Estimated_Time_to_Work* calculates the estimated time a server has to work on the requests currently in its queue.

The load value given by theses strategies is seen as a score for a server, that is, the server with the lowest score has the lowest load. The load values corresponding to the chosen load balancing strategy are copied into the `ScoreTable`. The second step combines the score obtained by the load balancer with the quality score calculated by the trader. Metrics like the euclidean metric are used to calculate an overall score for each service offer, see figure 4.

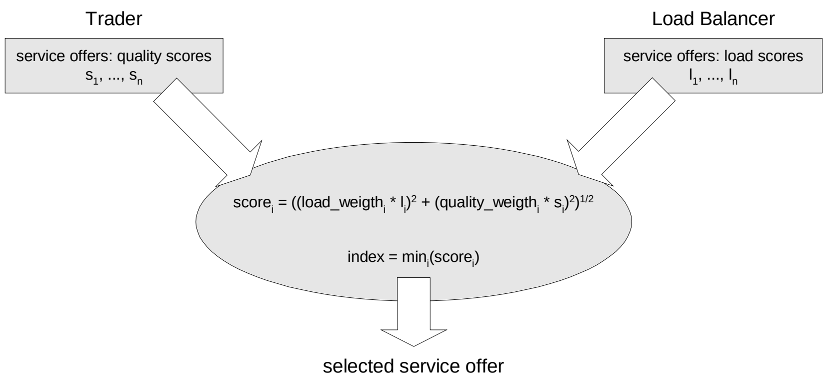

Trader
Load Balancer
service offers: quality scores $s_1, ..., s_n$
service offers: load scores $l_1, ..., l_n$
$score_i = ((load\_weigth_i * l_i)^2 + (quality\_weigth_i * s_i)^2)^{1/2}$
$index = min_i(score_i)$
selected service offer

Fig. 4. Combination of quality score and load score

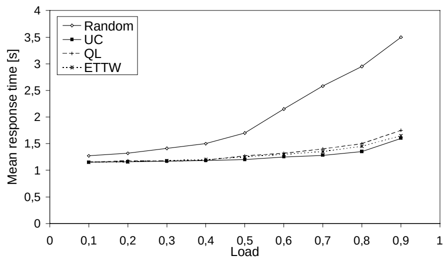
The client can influence the combination by specifying weights for quality and load score. For each service in the `ScoreTable`, an overall score is computed and registered in the `ScoreTable`. After all scores have been computed, a minimisation of these values gives the best suited service offer. The load balancer passes an index back to the trader, which identifies the corresponding entry in its result list.

4 Evaluating the Enhanced Trading Approach

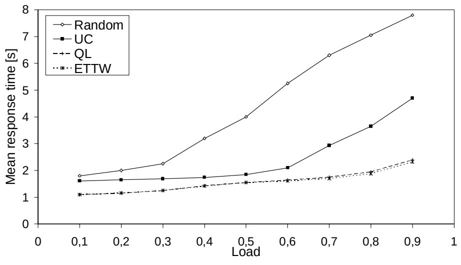
To evaluate our approach, measurements were performed to analyse the effects on server utilisation, response times, and service selection time. To approximate a real scenario, a request sequence was generated by using a random number generator to place requests in a given interval. One restriction was the avoidance of a system overload, but temporary overload situations were desirable. Thus, the request sequence contains request bursts and intervals of silence. This sequence was used in all measurements. The service time for the requests varied from 1.1 to 16 seconds. For load balancing, the strategies *Random*, *Usage_Count (UC)*, *Queuelength (QL)* and *Estimated_Time_to_Work (ETTW)*, as described above, were used. To evaluate the new component, the mean response time of the servers as well as the service selection time of the trader were measured at the client. The measurements were made in a local network (10/100 Mb/s-Ethernet). As a sample for a homogeneous system, four Sun UltraSPARCs with 167 MHz and 128 MB RAM were used. For measurements in a heterogeneous environment, four different Suns with clock rate between 110 and 167 MHz and RAM between 32 and 128 MB were used. Restrictions to a caching strategy to avoid communication overhead was made. To evaluate the benefit of the implemented load balancing strategies, the order determined by the trader computing a quality score, was not considered in the first measurements.

4.1 Homogeneous vs. Heterogeneous Server Performance

At first, a homogeneous system with four equally equipped servers was used. All servers had a service time of 1.1 seconds. In figure 5a, the mean response times of the servers are shown. The usage of the strategies UC, QL and ETTW lead to relatively equal response times, and for load situations below 50% they almost reach the optimum of 1.1 seconds. For higher load situations, response times go up only slightly, whereas the random distribution of requests to servers deteriorates dramatically. For the given scenario, UC achieves the best distribution, since all servers need the same time to process a request. For a system with homogeneous computer performance and no server usage without contacting the trader, a good load balancing is only possible with the trader's knowledge. This is the same result as in [11].



(a) homogeneous system



(b) heterogeneous system

Fig. 5. Mean server response times

The behaviour of the load balancing strategies changes in a heterogeneous environment. As a result of the varying computer performance, the servers had service times of 1.1 to 2.2 seconds. In figure 5b, the mean response times of the servers are shown. In this case as well the random strategy yields the worst behaviour. However, in contrast to the homogeneous case, the UC strategy performs poorer than the dynamic strategies. This behaviour was to be expected, as in this case, servers with less performance receive the same number of requests than more powerful ones. The dynamic strategies achieve equal request distributions, that is, even in situations of high load the mean response time for both strategies is only as high as the service time of the slowest server. To improve the quality of the static strategies, it would be possible to weight the distribution with regard to the servers' performance. However, in this case, the trader would have to know the performance characteristic of all servers in the system, and this might be impossible in a real open environment.

4.2 Using Different Request Classes

Considering different service request classes, i.e. requests to servers with different service times, represents a more interesting example of a real environment. In our analysis, we used four request classes with service times from 0.4 to 8 seconds. The requests were randomly chosen from these classes. This analysis covered two different situations. First, the request classes could be considered as requests for different service types with varying service times. Second, requests could be made for the same service type, but the service time is temporarily delayed by a background load on the server nodes. The mean response times for all request classes can be seen in figure 6. In case of a homogeneous system, the random strategy yielded the worst request distribution as could be expected, see figure 6a. Looking at the other strategies, UC is the worst in high load situations, because it does not consider the service times for the incoming requests. For lower system loads, UC is more suitable than ETTW. As the service times vary heavily, errors may occur in the estimation of that strategy, which than causes a wrong decision for the next request distributions. Only for high load situations this error is smaller than the misdistribution caused by UC. The error in estimation is also the reason for ETTW performing poorer than QL. QL only counts the number of outstanding requests; in this case, doing the distribution without more information about the requests is better than using potentially wrong information.

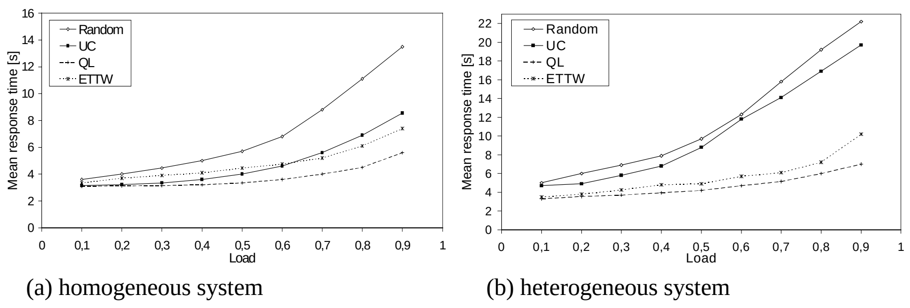
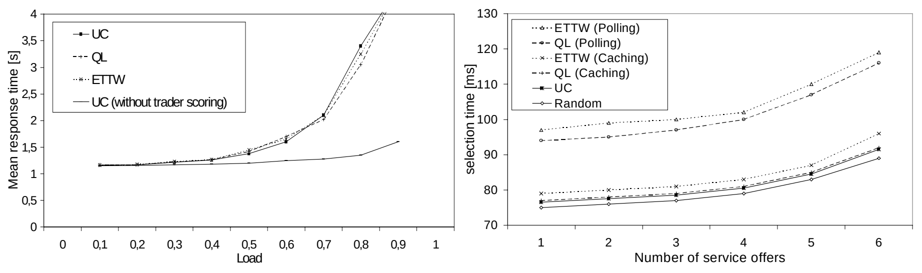


Fig. 6. Mean response time using different request classes

The most interesting scenario is the combination of the other ones, i.e., the usage of heterogeneous server performance and different request classes. The measured response times are shown in figure 6b. In this analysis, the static and the dynamic strategies can be clearly distinguished. With the exception of the random strategy, the results are the exact opposite of those in the first examination. UC is almost as bad as the random strategy, because to many factors influence the response time of a server, but none is considered. For the same reasons as above, the results of the dynamic strategies are equal to the results in the homogeneous case with different request classes.

4.3 Analysing the Trader

The trader's order of the service offers based on a client's constraints was ignored in the presented analyses to examine only the benefit of the load balancing strategies. As in a real situation this order denotes the user's requirements on a service, it cannot be neglected; both, the service quality described by the trader's ordering, and the server load have to be considered. We found that the behaviour of the load balancing strategies is similar to the case without considering the trader's ranking. Weighting the load with 75% and the trader's ranking with 25%, yields hardly any difference to the former measurements; the ranking was influenced mostly by the load situation. In case of reversing the weights, the load was almost neglected, and for load situations of more than around 20% the mean response time increased heavily, as nearly all requests were sent to the server which was valued best by the trader. The best result was achieved with a weighting of 50% for both, load and quality score. In figure 7a, the mean response time for the homogeneous case without different request classes is shown as an example. Yet, weighting the load influence in comparison with the service quality must be a choice of the user.



(a) using a compromise (b) mediation time of the trader

Fig. 7. Trader analysis

In a further measurement, we examined the influence on the trader's service mediation time caused by the load balancer, see figure 7b. When using a caching strategy for obtaining the load values as well as for using UC, the time for service mediation increased only slightly, as most of the load balancer's work can be done independently from the trader. The load scores only have to be copied into the `ScoreTable` and the whole list has to be searched for the minimal value. When using a polling strategy, additionally all load values are obtained from the monitors. This adds a significant overhead to the mediation time, and should be avoided in most cases. By using the dynamic switching between caching and polling described in chapter 3, this overhead can be adapted to a given scenario.

5 Conclusions

In this paper, a combination of a trader and a load balancer was presented. Instead of using the trader's dynamic service properties, a load balancer was added to the trader as an additional component. Thus, the approach is independent of the servers' characteristics, and is flexible to enhance. Several load balancing strategies, and a concept for combining the trader's and the load balancer's results, were implemented on a CORBA platform. The implementation was evaluated in several scenarios.

The optimisation of service selection with respect to the servers' load seems to be a worthwhile enhancement of the trader. The response times of servers offering a service which is available in several places can be significantly reduced. The cost for this advantage is an increased service mediation time, but this overhead is very small. The usage of trader-internal knowledge like the number of mediations of a server is only useful in idealised scenarios. In a heterogeneous environment, it does not help to improve the load situation significantly. For such environments, dynamic strategies are more suitable. A simple strategy like the trader's queue length is best for most situations. The weighting of the load influence in comparison to the service quality must be the user's choice, but an equal consideration of both seems to be the best.

In addition to an evaluation of our trader in a bigger and more realistic scale, the next steps will be to realise a load balancing in a larger system in which trader federations are used. Obtaining load information and the co-ordinated interworking of traders in the load balancing process adds new, additional tasks. Additionally, the load concept has to be enhanced to consider more aspects influencing a server's performance. Furthermore, a usage of the load balancer component for other trader types or existing traders would be interesting. In such a case, the load balancer would need to encapsulate the trader. As in our approach trader and load balancer are largely independent, this approach is a candidate for these studies.

References

1. Dierkes, S.: *Load Balancing with a Fuzzy-Decision Algorithm*. Informatics and Computer Science, Vol. 97, No. 1/2, Elsevier/North-Holland, 1997.
2. Eager, D. L.; Lazowska, E. D.; Zahorjan, J.: *Adaptive Load Sharing in Homogeneous Distributed Systems*. IEEE Transactions on Software Engineering, Vol. 12, No. 5, 1986.
3. Golubski, W.; Lammers, D.; Lippe, W.: *Theoretical and Empirical Results on Dynamic Load Balancing in an Object-Based Distributed Environment*. Proc. 16th International Conference on Distributed Computing Systems, Wanchai, Hong Kong, 1996.
4. Keller, L.: *From Name-Server to the Trader: an Overview about Trading in Distributed Systems* (in German). Praxis der Informationsverarbeitung und Kommunikation, Vol. 16, Saur Verlag, München, 1993.
5. Kovacs, E.; Burger, C.: *MELODY – Management Environment for Large Open Distributed Systems*. Institute for Parallel and Distributed High-Performance Computers, University of Stuttgart, 1995.
6. Linnhoff-Popien, C.; Thißen, D.: *Integrating QoS Restrictions into the Process of Service Selection*. In: Campbell, A.; Nahrstedt, K.: Building QoS into Distributed Systems. Chapman & Hall, 1997.
7. Lipperts, S.; Thißen, D.: *CORBA Wrappers for A-posteriori Management*. Proc. 2nd International Working Conference on Distributed Applications and Interoperable Systems, Helsinki, 1999.
8. Milosevic, Z.; Phillips, M.: *Some new performance considerations in open distributed environments*. IEEE International Conference on Communications, New York, 1993.
9. Neukirchen, H.: *Optimising the Set of Selected Services in a CORBA Trader by Integrating Dynamic Load Balancing* (in German). Diploma thesis at the Department of Computer Science, Informatik IV, Aachen University of Technology, 1999.
10. Schiemann, B.: *A New Approach for Load Balancing in Heterogeneous Distributed Systems*. Proc. Workshop on Trends in Distributed Systems, Aachen, Germany, 1996.
11. Wolisz, A.; Tschammer, V.: *Performance aspects of trading in open distributed systems*. Computer Communications, Vol. 16, Butterworth-Heinemann, 1993.