Int J Digit Libr (2000) 3: 19–35

INTERNATIONAL JOURNAL ON
Digital Libraries

A framework for performance monitoring, load balancing, adaptive timeouts and quality of service in digital libraries

Sarantos Kapidakis[1], **Sotirios Terzis**[2], **Jakka Sairamesh**[3]

[1] Institute of Computer Science, FORTH, Heraklion, Crete, GR 71110, Greece;
E-mail: sarantos@ics.forth.gr
[2] DSG, Computer Science Department, Trinity College, University of Dublin, Dublin 2, Ireland;
E-mail: Sotirios.Terzis@cs.tcd.ie
[3] IBM T.J. Watson Research Center, 30 Sawmill, Hawthorne, New York, NY 10532, USA;
E-mail: jramesh@watson.ibm.com

Received: 18 December 1998/Revised: 1 June 1999

Abstract. In this paper, we investigate the issues of performance management in large scale, autonomous and federated digital library systems, performing the tasks of indexing, searching and retrieval of information objects. We have defined a management architecture and performance framework for measuring and monitoring the behavior of digital libraries as they operate. Our architecture and mechanisms are easily applicable to other digital library systems of similar flavor and architecture. We implemented this architecture over a testbed of Dienst servers using real data and workload from the operational NCSTRL system. We have defined the relevant parameters for investigating the performance of the servers and have developed visualization tools to monitor the parameters. In addition, our performance framework provides mechanisms for load-balancing search requests in a network of digital library servers. We have demonstrated this by building a testbed for investigating a few novel load balancing policies. Given that network delays and outages are unpredictable over the Internet, we have developed new adaptive mechanisms to detect timeouts and provide quality of service.

Key words: Performance management – Load balancing – Adaptive timeouts – Quality of service – Dienst servers – NCSTRL

1 Introduction

The rapid advances in computer and networking technology in recent years has provided worldwide access to a huge volume of information and services for an increasingly larger number of people. *Digital libraries* [5] recently have emerged to offer a structured way of organizing, indexing, searching, and retrieving information. They are architectures for the provision of information access and management services for information repositories consisting of various information objects such as text, audio, video, and image. Currently, there is research being conducted around the world on digital libraries and their associated issues, architectures, and mechanisms [3, 5, 6, 9, 13, 14, 17, 24, 32, 39] or solutions to specific issues. Their wide-spread and increased popularity will influence the design of future information systems.

Digital library systems are characterized by the huge volume of information they store (e.g., the Alexandria system [11] stores gigabytes of information as satellite images and maps), by the wide distribution of their nodes (e.g., the NCSTRL-Dienst [18] system has more than 100 nodes in over 15 countries covering most of the USA and Europe) and by the fact that they often integrate existing collections of information over the *World Wide Web* (e.g., the Infobus [25] that connects systems like Altavista, the Alexandra system [11] and the university of Michigan digital library system [2]). These characteristics make those systems especially changeable and have a profound influence on their performance. As a result, mechanisms for the dynamic adaptation of the system to the constantly changing operation environment are required. In this article we investigate the issues of performance management and monitoring in large scale, autonomous and federated digital library systems.

In the development of digital library systems various models have been deployed. For example, Stanford University's [25] view of digital libraries is as a shared information bus that connects various information sources. On the other hand, Michigan University's [2] view is of a collection of collaborating agents. The common denominator in all these different views is that a digital library system consists of a number of servers, spread over the

MS ID: JODL013
10 March 2000 23:53 CET

Internet, that interact with each in order to service user requests. In processing a request a server might invoke a number of external programs. All the communication between the servers is done with the use of the World Wide Web's protocol, *HTTP*. This is the system model for our investigation.

Our primary goal is to define an architecture for monitoring and measuring the performance of digital libraries. This architecture should be based on the model presented above and should allow us to monitor the system's behavior as it operates, since it will be the base for the dynamic behavior of adaptation mechanisms, such as load balancing, dynamic timeout adaptation and support for quality of service searching and retrieval [15, 33]. A secondary but equally important goal is that the monitoring process should be applicable to any digital library system. This implies that the architecture should (a) impose minimal overhead on the system's performance and (b) require minimal changes to the system's code.

We have developed an architecture and framework for the monitoring and measuring of digital libraries. We implemented this architecture over a testbed of Dienst servers using NCSTRL data. We defined the relevant parameters for investigating the performance of the servers.

We conducted a performance study on our testbed using some special visualization tools, which we implemented and tested. The performance study was based on the tracking of a search request by monitoring the system and led to the proposal of some modifications on the design and implementation of the Dienst system. In order to demonstrate the performance architecture for dynamic behavior adaptation, we designed and implemented some load balancing strategies for distributed searching over a network of servers. According to these strategies a server, using local observations, can forward a request to the *least loaded*[1] server from the pool of those servers where the requested information is available (replicated). We built a testbed to demonstrate the extensibility of our architecture, investigate load balancing policies and dynamic timeout adaptation during distributed searching.

In Sect. 2 we present our architecture for performance analysis and interesting application areas: performance monitoring, load balancing, dynamic adaptation of timeouts and quality of service. We also discuss architectural implementation issues and the supporting visualization tools we have made. In Sect. 3 we describe the Dienst system, analyze its operation, present our extensions to it to take advantage of performance monitoring, and explain how we carried out performance monitoring on Dienst. In Sect. 4 we present the results from using our implementation, with emphasis on the load balancing results, and conclude in Sect. 5. Appendix A gives details, and a mathematical formulation, of the Dienst request processing performance model.

2 Performance framework for monitoring and load balancing

Considerable work has been done in the area of performance management and monitoring in distributed systems [10, 12, 16, 20, 23, 28, 29, 31, 34]. Besides this research work, a series of commercial products for distributed systems performance management is also available [1, 4, 22, 26, 27, 30]. Of particular interest in our case is the work in online performance monitoring [34], since our goal is to design a performance monitor that (a) will not interfere with the system's operation (external monitor) and (b) will monitor the system in operation. Additionally, performance monitors deploy various semantic models in their operation according to [12] and can be classified based on approaches like *program profiling* (e.g., Parasight [1]), *event based* (e.g., Pablo [28, 29]), or *information modeling based* (e.g., the performance monitor described by Snodgrass in [31]). In our case the most appropriate approach seems to be the *event based* since according to our system model the actions of interest are component invocations and message exchange. Although past research work has addressed most of the main issues in performance management, the use of a current commercial product is forbidden because they (a) are proprietary and (b) are limited only to some hardware and software platforms.[2]

In this section, we discuss the performance monitor architecture. The performance monitor measures a number of performance parameters and provides access to the measurements. These parameters are defined through a performance analysis of the library system. With the use of these measurements we can implement a series of policies that improve the performance of the systems both at the administrative and user level. These policies are load balancing, dynamic adaptation of timeouts, and quality of service.

2.1 Performance monitoring in digital libraries

We defined the minimum server functionality to support a management architecture for monitoring and measuring.

We developed a simple client-server architecture model for performance management of the digital library. This model is based on ideas from SNMP-based performance management in networks and distributed systems.

[1] The least loaded server could be based on the average of the response times for search requests sent to that server. The response time takes into account the processing load at the server and network delays. More complex functions can be chosen, but this is beyond the scope of this paper.

[2] The development of the Universal Measurement Architecture *UMA* [37], an X/Open standard (implementation available by Amdahl [38]), although dealing with the problem of hardware and software incompatibilities, does not deal with the problem of proprietary technology.

We developed mechanisms for measuring the delays in the various components of the digital library system. We have broken down the performance monitoring architecture into five main elements, which are stated as follows:

- *Performance parameters:* we define and name the performance parameters (of various software modules and components) for the various tasks performed by the server when servicing a user request. Each performance monitor keeps a list of well-defined parameters, which it updates based on every request it processes.
- *Measurement system:* we measure and store the performance parameters, by developing a measurement system for updating and storage. We developed mechanisms to measure the variables for every request generated. A measurement process (daemon) updates the performance variables and stores their current values, averages and variances in a database.
- *Protocol:* we define a protocol to retrieve the performance parameters. The parameter database is managed by a process, called database manager process (DMP), which returns the variables and values in the database.
- *Visualization tools:* we use tools to visualize the performance parameters during the operation of the digital library system.
- *Messaging:* we extended the digital library protocol in order to retrieve and report the performance parameters during searching. We developed a simple message protocol to monitor and debug the server.

Only the first and last components are digital library specific and must be designed and implemented differently for each system. The rest are generic enough for most digital library systems.

A performance variable is associated with each task (or function) that we wish to measure. For example, we measure the time spent by a server while searching its local database.

In order to define the performance parameters, we first analyze the digital library system to find and clearly define all procedures of interest, which we call *components*. Although different components may overlap, we must specifically locate their starting and ending times. Figure 1 indicates a possible relation for some components and their starting and ending times.

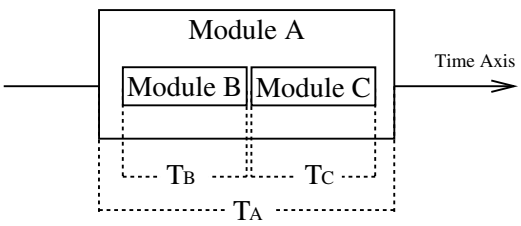

Fig. 1. Example relation of components ($T_A > T_B + T_C$)

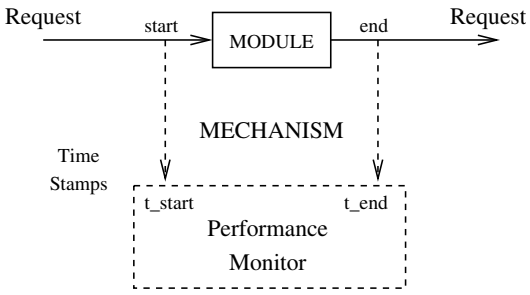

Fig. 2. Messages passed by components, to the performance monitor

The performance monitor captures the time spent by the server while performing the various tasks of a user request. The entry and exit time of components are recorded, as illustrated in Fig. 2.

Each server sends two messages to the performance monitor: the first message when the task begins, and the second just before it finishes. This is done for every task we measure. The messages are sent through a socket to the performance monitor. *These are the only changes that are needed on the server.* This way, the changes to the code of the digital library are minimal, and do not compromise its functionality and complexity. Additionally, the digital library performs as minimal additional work as possible, and does not sacrifice its performance, as only some extra messages are sent, and separate processes take care of the rest of the procedure. Finally, the digital library does not depend on the existence or the operation of the performance monitor.

In our architecture, the performance monitor can accept many connections, for retrieving performance parameters of the digital library system. Two alternative interfaces are provided for this purpose. One for direct requests to the performance monitor, and one for requests through the digital library system, as seen in Fig. 3.

This way, others, like system administrators and users interested in the system performance, as well as the digital library itself, can ask for the performance of the system. Using the appropriate requests, we can retrieve specific or all performance variables from the performance monitor.

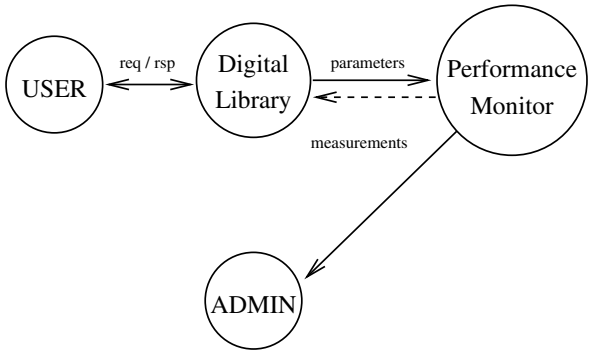

Fig. 3. Interaction with the performance monitor

Using this architecture, we can also build a self-adapting digital library system, a system where the digital library, just like any other process, queries its own performance, as seen in Fig. 3, and can take decisions that depend on past performance. Now, the digital library can adapt to the performance conditions (e.g., the timeouts), and can explore new alternatives (e.g., load balancing).

2.2 Load balancing

In creating and maintaining a large distributed digital library system with many servers on the Internet, the problems of replication of indexes and objects and load balancing need to be addressed. Especially when the digital libraries contain multimedia objects, which, while being searched and retrieved by users, will use up many system (local, remote and network) resources. A measure of the system load is very useful to route requests to the idle (or least loaded) servers that contain the relevant objects. We expect that servers will *replicate* information objects or indexes of information objects in order to utilize network resources efficiently.

By the term load balancing we mean the dynamic routing of requests to the servers of the library system. Since digital libraries are naturally dynamic systems static routing of requests is deemed to be inefficient. The performance monitor with the measurements it produces allows the library systems to estimate the response time of the various servers. So, it can use these estimates for dynamic routing of the requests. The goal is the minimization of the total response time. There are a number of different mechanisms for the retrieval of the measurements:

- *Polling:* the library system periodically sends a message to all the servers of the system, measuring their response time. These messages are either ping messages or pseudo requests. Polling is not widely used due to its high communication cost (a lot of ping messages).
- *Probing:* the library system periodically sends probing requests to all the servers and they reply with their average processing time and their current status. The disadvantage of this method is that all the servers have to be contacted which dramatically increases the response time of the system.
- *Name services:* every server advertises its response times in special naming servers. In this case we have a combination of the above approaches since the servers periodically send data to the name servers and they retrieve these data whenever they need an update. The only difference is that the whole process is done through a name server. This approach is particularly appropriate for large scale systems. In these systems the name server could be distributed in order to minimize the cost for contacting them.

Load balancing is possible only when replication is used. The replication can be based on political and/or technical reasons or, in the best case, on past performance statistics from server load and network conditions.

Considerable work [7, 8, 35, 40] has been done in developing algorithms for load balancing jobs or transactions or queries in large computer systems, but very few [21] have investigated issues in designing and implementing mechanisms for monitoring performance and load balancing jobs in distributed computer systems spread across a vast network such as the Internet.

Some advantages of load balancing are that there is no need for a priori network knowledge, especially when nodes are setup for the first time, and it can lead to better performance, with dynamic system reconfiguration, as the system always adapts to the changing environment. Also, it leads to a simpler system structure, where there is no need for explicit specialized roles such as Merged Index Servers and Backup Index Servers, and provides better reliability when servers are down, as others are automatically the next choice.

To avoid lengthy computations on the digital library server, the digital library protocol has to be further extended to get precalculated results, like the ordered list of servers to query.

Some disadvantages and problems of load balancing is that the network conditions change continuously, and performance data based on non current information are partially useful. In a few circumstances, there may be no previous history and the decision will be almost random. Finally, there may be unpredictable response time arising from factors that are not easily detected, such as from a dependency on query complexity.

Our performance monitoring system can also be used for load balancing: since the performance measurements are known to the performance monitor, the digital library server can ask for these measurements and decide where to send its requests. The load balancing may refer to requests that access indexes (mostly search queries) and/or to requests that access objects (objects retrieval). There can be advantages in both cases: the use of indexes is more dense, while objects may be bigger. The mechanisms for load balancing are the same for both cases. We provide the mechanisms for load balancing, a good testbed for developing, debugging and evaluating policies.

2.3 Dynamic adaptation of timeouts

A big problem in distributed systems is the synchronization of the distributed processes, due to the high diversity of system heterogeneity and internet bandwidth. When a process waits for another process on a different host to finish, it never knows how long it should wait for, as the process may still be working, or may be unable to return its result. Unnecessary waiting results in degradation of performance, and a good estimation of the amount of time to wait, the *timeout period*, can improve the response time with minimal (or the desired degree of) information

lost. In most systems, the timeout periods are predefined constants.

We also propose novel timeout adaptation mechanisms which set timeouts for distributed searching in a dynamic fashion by estimating the round-trip delays to various digital library servers and by defining appropriate waiting time intervals. The estimations are based on the response time history for each server that the performance monitor keeps. The timeout period is set to be: $min\{TO_{\text{user}}, max\{T_{\text{est}_i}\} + T_{\text{SI}}\}$, where TO_{user} is the upper limit for the timeout period set by the user, T_{est_i} is the response time estimate for server i, and T_{SI} is a safety interval. The purpose of the safety interval is to maintain the probability that server i will respond in time above a user-defined limit. So, if we know the mean response time and the response time variance for server i then the Tchebisev inequality ($P\big[\mid x - E(x) \mid > c\big] \leq \sigma^2(x)/c^2$, where x is the current response time, $E(x)$ and $\sigma(x)$ are the current mean and standard deviation of the response time and c is a user defined constant) can provide us with the required safety interval.

For the smooth operation of parallel and distributed searching mechanisms, timeouts of search requests to various servers need to be addressed. These are crucial to the operation of the digital library system under network or server failures. The values of the timeouts play an important role in the distributed search response time. For example, short timeouts will make the servicing of complicated queries or the use of "remote" servers impossible. On the other hand, too long timeouts will deteriorate the response time of the system and its utilization (the system will spend a lot of time waiting for servers that are unreachable).

2.4 Quality of service provisioning

The data included in digital libraries are in various formats and every format usually has a different quality of presentation (e.g., higher analysis images, different document formats). Also, for the retrieval of the library objects various sets of servers could be asked (e.g., only the closest servers) and various search methods could be used (e.g., keyword search or full-text search). In any case the retrieval is a tradeoff between speed and quality. It would be good for the user to be able to specify the level of quality he/she wants for his/her requests. The use of the performance monitor can provide the user with estimates on consumption of resources for the various levels of quality. So, the user can specify the level of quality he/she wants and the systems could find out if it can guarantee that level of quality based on the performance estimates it has. The library system could even implement a negotiation mechanism for the discovery of a set of servers that could provide the requested quality of service. Finally, since the system monitors resource consumption it could support charging mechanisms too.

Quality of service guarantees can only be given when we have an estimate of the current network performance. Thus, our architecture is a prerequisite for that. As a first step, we only show the expected performance to the users by estimating the expected performance of the available operations (like the transfer time of files) from past history. For this functionality, we also used an external tool that monitors all TCP/IP packets from the local network to all destinations, keeps statistics and provides available bandwidth information, on request.

Our mechanisms can be used for full quality of service support, according to the desired levels of service. To support many levels of quality of service, the past and current performance is used and many possible scenarios can be evaluated.

2.5 Implementation issues and supporting tools

The way that the performance monitor communicates with the rest of the world, does not specify its internal structure and its evolution. In its simpler form, it could be a single-threaded process. A system with two manager processes, which, technically, can be different processes or just threads, is more functional:the first process captures the messages, time-stamps them and passes them to the second process through an open pipe; the second process computes the difference between the time-stamps and updates the corresponding variables. It also computes the mean and variance. This way the first process is always unloaded and can process requests instantly, so that the added time-stamps are accurate. Of course, if the digital library system has the ability to satisfactorily provide accurate time-stamps with negligible performance cost, the first process can be eliminated.

As the second manager process may have to make heavy computations, and during this time is unable to process requests, a third manager process, called Database Manager Processes (DMP), that manages the variable database, ensures that performance responses are given instantly, as seen in Fig. 4.

The second manager process still makes all its heavy calculations, and when new results are available, it sends

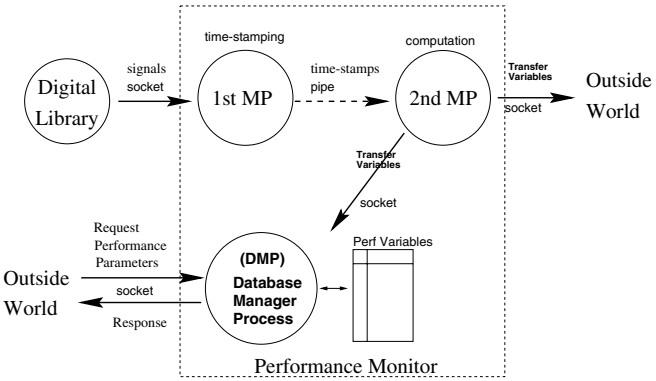

Fig. 4. Internal structure of the performance monitor

the updates to all processes connected it, such as the DMP, which is then responsible for answering requests from the outside world.

Performance tools can be easily connected now: tools that poll for new values can connect to the DMP, and tools that passively wait for updates when they are available can connect to the second manager process.

2.5.1 Visualization of performance parameters

The performance parameters are useful to administrators, to monitor the performance of the network and their systems, to detect problems and to make appropriate decisions to improve the performance. Also, the performance parameters are useful to users, to see the performance of their system and adjust their actions or expectations. In any case, visualization tools are needed to present the performance behavior to humans.

There are many different ways that users can see the performance parameters, such as:

– Using the performance log files directly. Our performance monitor keeps logs, if configured to do so, in many files in html format with links between them (like from one digital library component to another, following the execution path), so that users can use them offline to process the parameters and see the flow of information and to follow process or parameters relations, using the links. Although this is very helpful for statistics and post-mortem debugging, this was not the main goal of our monitor.
– Using a WWW browser and performance monitoring requests, users can see current values of the performance parameters. The performance monitor accepts http requests for performance and formats and sends the reply in html format, so that the user can use his/her familiar WWW browser to utilize the monitor. The replies have a strict structure, so that they are easily parsable by programs, too.
– Using a graphics tool users can see the parameters as they change. We built a graphical visualization tool, as shown in Fig. 5, that can connect at the performance monitor and obtain the current system performance and display it. In this tool, every request creates a new set of points. The tool has a Java-based user interface, can selectively display some of the performance variables as they change in real-time (i.e., new requests are coming), and can also read the performance monitor log files and display the past values of the performance variables, while appending the new requests to the picture.

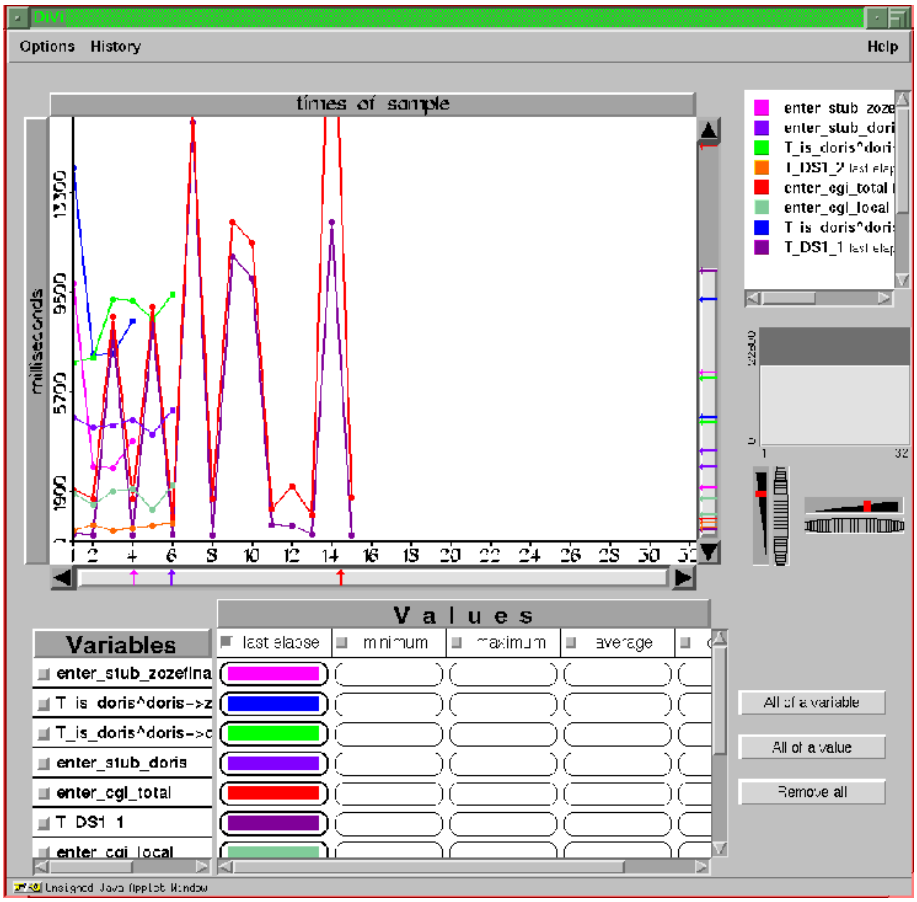

Fig. 5. Our performance visualization tool

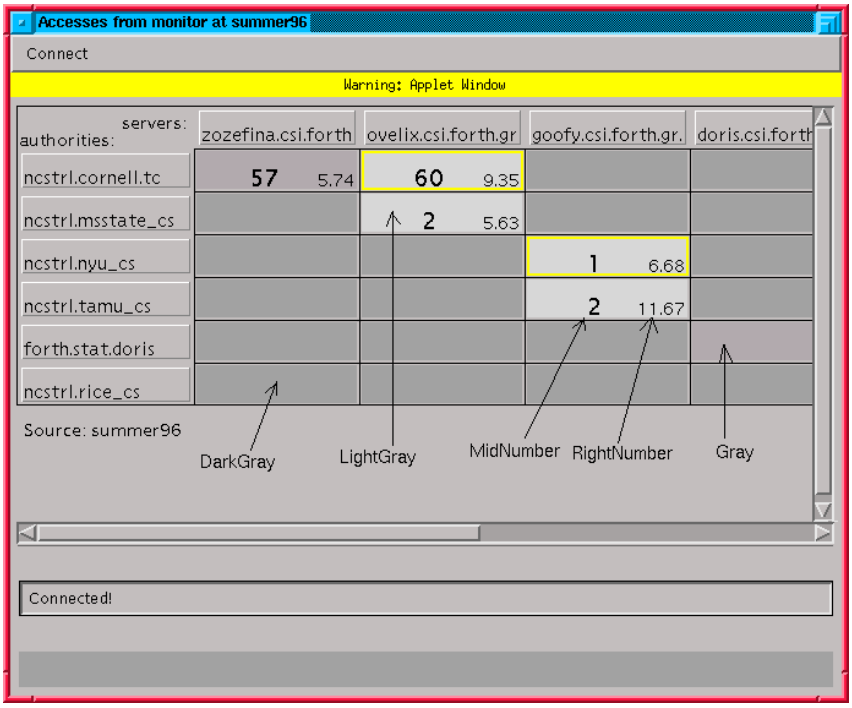

Fig. 6. Load balancing visualization tool

– Using other (most probably interactive) custom tools or agents. Since our system is open, many such tools can be made and connect to the performance monitor (possibly at the same time, too).

In order to visualize the load balancing status and effectiveness, we also made a load balancing monitor tool, as shown in Fig. 6.

3 Using the performance monitor on Dienst

In order to test our ideas, we need to implement them on a real, working system. We applied our ideas to the NCSTRL [19] based digital library system, which consists of Dienst servers distributed across the Internet. Dienst was our system of choice because it is used in NCSTRL,[3] and connects sites all over the world, providing a good natural testbed for distributed testing on digital libraries. Each Dienst server manages a collection of computer science technical reports (documents) owned by organizations such as computer science departments and computer science research institutions.

3.1 Description of Dienst system

We applied our performance management architecture for measuring and monitoring the operation of the NCSTRL-based digital library system. We conducted our experiments over a testbed of Dienst [19] servers; Dienst uses the WWW protocols (mainly HTTP) for searching and presentation. These servers manage three basic library services: (a) repositories of multi-format technical reports; (b) indexes of the technical reports collection and search engines for these indexes; (c) distributed search and retrieval.

Dienst is a digital library system that provides transparent distributed search and document access and each node of the system consists of a database that contains the available objects (reports), a WWW server that handles all incoming Dienst requests, Dienst CGI stubs that are called by the WWW server, and a Dienst server that is called by the CGI stubs.

The operation of a Dienst server is as follows: a user submits a keyword search query[4] (request) to one of the Dienst servers. The Dienst server initiates a search request to the other Dienst servers, responses are collected, and the user is presented with the search results.

We defined the performance variables which capture the delays experienced by each user request as it propagates through the components (modules) of the digital library system. The components, for example, include the delays in WWW interface to the digital library (DL) system, local digital library server processing, network delays and remote digital library server processing of the user requests.

To overcome bad network connectivity and delays, Dienst divides the servers into regions and replicates the indexes for reliability. It uses *Backup Index Servers*, when a Dienst server fails to respond in time and additionally

[3] NCSTRL stands for Networked Computer Science Technical Report Library.

[4] In this paper, request, search request, query, all mean the same.

it uses (statically assigned) *Regional Meta Servers* and *Merged Index Servers* (Regional Index Servers), which keep a replica of the indexes of the other regions.

A query within a region is first directed to the Dienst servers in the region and the MIS (Merged Index Server). The combined results are collected and submitted to the user. The user then chooses documents, and obtains them via the URLs supplied by the search results. There are several factors that affect the overall response time of the search request, like the processing capacity of each Dienst server, the load (current number of active user requests) generated at the Dienst servers, the network delays between the Dienst servers, the (local and/or remote) processing time of a search request as a function of its type (there are many types of queries: simple keyword to more complex multiple keyword based searching), and local searching time and the size of the index file at each Dienst server and the MIS.

Each MIS acts a front-end index server to the rest of the regions in the world. Due to the replication of indexes by the MIS of each region, user queries could be submitted by the Dienst servers of a region to other MISs as well. To improve scalability and performance, an entirely new architecture is needed, without such strong statically assigned roles, based on performance facts and not speculations.

3.2 Analysis of Dienst operation

As seen in Fig. 7, the user communicates directly with one Dienst server, and when he/she issues a Dienst request, this server decides where to forward the split subqueries. Each server is responsible for searching its own local Database.

Each Dienst request goes through many stages, as seen in Appendix A, where most of them are trivial components, but may introduce significant overhead.

According to our performance architecture, our model for performance monitoring on Dienst is simple as illustrated in Fig. 8. The performance requests also use the native Dienst model: they can be sent to a Dienst server,

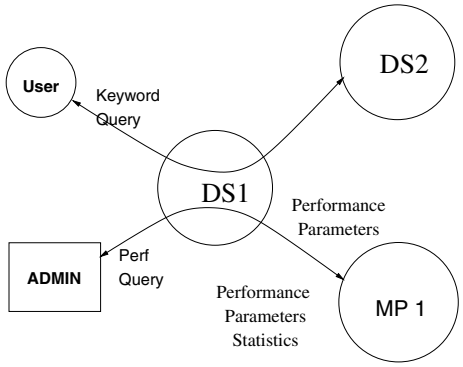

Fig. 7. The user view of the route of a query

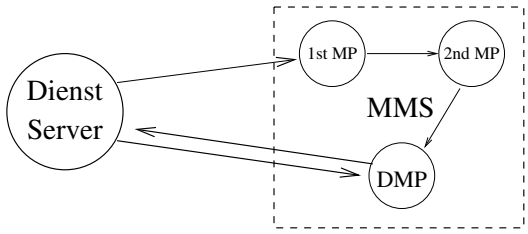

Fig. 8. The Dienst performance management system

and this server decides where to forward the split subqueries. Each server is also responsible for communicating with its own performance monitor to retrieve performance parameters.

By using our performance monitoring system on Dienst, we studied the Dienst protocol and program and located inefficiencies. We also studied the log files, to better explore the flow of information. We propose improvements, such as a new protocol for server communication and timeouts. We extended Dienst architecture to perform load balancing and proceeded to an implementation, which was used for experimentation and to obtain example results.

In order to have a user-friendly way to access the performance requests and to be able to ask the Dienst server itself for them, we extended the Dienst protocol by adding new Dienst performance requests:

`Dienst/Stat/2.1/Print-Local-Parameters`
for retrieval of all parameters local to this host only and

`Dienst/Stat/2.1/Print-Parameters`
for retrieval of all parameters of all hosts.

These requests just call the appropriate request from the performance monitor and redirect their output.

The performance requests can be asked for, like all other requests, directly on the WWW or from links in the Dienst interface. Users can continue using original Dienst requests only, and the performance monitor is invisible to them. The requests return simple html tables, as in Fig. 11, that can be shown directly to the user, or parsed by programs. More complex requests can always be answered directly by the performance monitor.

3.3 Distributed search timeout adaptation

The variance of the response times, as seen in Fig. 9, indicates that timeout selection affects performance. Figure 9 shows the distribution of the response times for 100 requests to 100 distinct servers. Some servers did not respond and have no response time in the picture. The horizontal lines indicate possible timeout settings. The higher the timeout setting is, the longer the user has to wait for getting the final answer to his/her query. For these timeout setting, we can see the number of servers that would not have been able to respond in time.

The Dienst system in the current implementation uses TO^{remote} (the local or remote database search timeout),

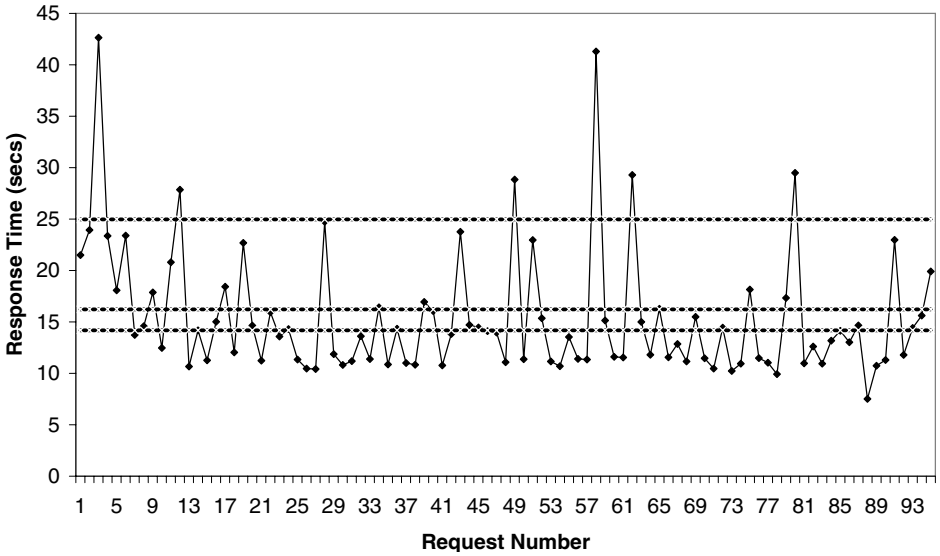

Fig. 9. Response time distribution and timeout cut-off

TO^{search} (the total search timeout) and TO^{backup} search (see Fig. 10). The values of all three are set at configuration time. The use of static values for the timeouts complicates the timeout problem because of the dynamic nature of the system.

As a solution to the timeout problem, we propose dynamic adaptation of timeouts, based on the history the performance monitor keeps for the response time of each server as was described in Sect. 2.3.

By abolishing the use of the timeout for the search of the local index database (TO^{remote}), this timeout will be ignored and replaced by the total search timeout (TO^{search})in the remote request message. The remote server will estimate the probability of processing the request locally before the timeout expires, using performance statistics. If the estimated probability is low then the server can notify for its inability to service the request.

The above extensions to the protocol result in some changes in the use of timeouts. Since the local server knows quite early if the remote server is alive or not and the goal of the total search timeout is to avoid blocking because the remote server is down or unreachable, then we have more information for the remote server's condition and we can adjust the timeout value. Thus, we can give a bigger safety interval T_{ESI} (Extended Safety Interval), for the servers that follow the extended protocol because quite soon either it will not respond and we could proceed with forwarding the request to another server, or it will tell us it is alive and give it more time to process the request. A new timeout for the alive messages is needed.

The above extensions abolish the notion of Backup Index Server. Thus, the procedure alive message timeout and total search timeout procedures can be expanded

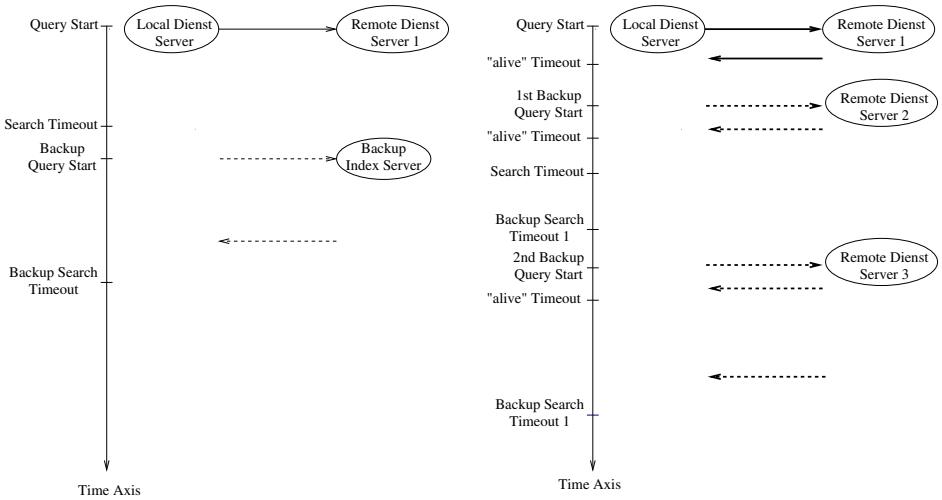

Fig. 10. Current and new timeout mechanism

to all the appropriate servers, until either we get an answer to the request or we exhaust the time of TO^{user}. On the other hand the notion of the backup search timeout ($TO^{\text{backupsearch}}$) could be maintained, set as:

$$min[TO^{\text{user}}, T_{1st_{\text{est}}} + T_{\text{BSI}}] \,. \quad (1)$$

$T_{1st_{\text{est}}} = max[T_{\text{est}^i}]$ of formula and T_{BSI} is the Backup Safety Interval.

For example, the Local Dienst Server sends a request to a Remote Dienst Server. The remote server times out, and the request is sent to the Backup Index Server and a reply comes back.

For example (see Fig. 10, the local Dienst server sends a request to remote Dienst server 1. The remote server 1 replies "I'm alive and can't service the request". The local server forwards the request to remote Dienst server 2. The remote server 2 replies "I'm alive and can service the request". The remote server 2 times out and the local server forwards the request to remote Dienst server 3. The remote server 3 replies "I'm alive and can service the request" and after some time returns the reply to the request. Note that the request is forwarded to remote server 3 before the search timeout expires and the absence of specialized Backup Index Servers.

3.4 Load balancing

We have neither developed efficient policies nor built a specific policy into our system. We did try an indicative policy that gave acceptable results. We did this in order to demonstrate the functionality of our system. Good load balancing policies may depend on geographical data, connectivity, reliability requirements, data distribution and replication, user requirements and even on legal issues and individual cooperation and deals. We have provided the testbed to experiment with such algorithms.

We have built the mechanisms, but in order to test them we also need policies. Load balancing can refer to retrieval of either indexes or data. In our policy implementation, we only performed load balancing on indexes. The algorithms for dynamic routing are based on estimates of the response time of alternative server choices that the performance monitor provides. Their goal is the minimization of the total response time. In particular, suppose that we want to route a request for the publishers a_i where $i = 1, \ldots, N$ and that $\text{ER}_{jk}^{a_1,\ldots,a_{N_k}}$ is the estimated response time for server S_k for a request from server S_j for the publishers a_i, $i = 1, \ldots, N_k$ and M the number of servers. Then the problem of choosing the appropriate servers for the requests to all the publisher is as follows:

- Choose: $\text{S}_k^{a_i}, k = 1, \ldots, M$
- For all items $a_i, i = 1, \ldots, N$
- Where $min_{\forall(\text{S}_1,\ldots,\text{S}_M)|\text{S}_k\in(\text{S}_1,\ldots,\text{S}_K)} max_{k=1,\ldots,M} \text{ER}_{jk}^{a_1,\ldots,a_{N_k}}$

Choose a number of servers and a distribution of publishers to those servers that minimize the total estimated response time. The estimated total response time is the maximum estimated response time of all the chosen servers. This algorithm gives the optimal server choice but it requires computing the estimate for all the combinations of servers and publisher distributions. This means that its complexity grows exponentially with the number of available choices. We chose to implement a sub-optimal algorithm but with significantly better complexity as follows:

- Choose: $\text{S}_k^{a_i}, k = 1, \ldots, M$
- For all items $a_i, i = 1, \ldots, N$
- Where: $min_{\forall \text{S}_k} \text{ER}_{jk}^{a_i}$

For each publisher choose the server with the minimum estimated response time.

4 Experimental results

Here we describe a session where we experimented with monitoring the performance of three Dienst servers (DS1, DS2 and DS3), and interpret the results obtained. We defined a few variables and monitored them. Dienst requests (keyword based) were sent to one server (DS1) and the performance variables were monitored for each request and the statistics computed. Figure 11, displays an instance of the output of the performance monitor, as will be explained later:

The performance variables capture delays in the following components: (a) local search response time; (b) local index database processing time; (c) remote processing time of the index database; (d) remote search response time (includes network delay+remote processing time).

For each performance variable, the following are observed: `Calls`: number of requests (measurement points); `RSP`: current value of the variables (last request response time); `Min`: minimum value of the variable; `Max`: maximum values of the variable; `Total`: aggregate of the measurements; `Mean`: total divided by the number of requests; `Std Dev`: standard deviation of variable values.

In Fig. 11, 5 requests were generated at Dienst server 1. The requests were routed to the remote Dienst servers 2 and 3 for a search in their index files. The request is also routed locally for a search in the local index file (of Dienst server 1).

The parameters for Dienst server 2 and 3 are simply the remote search response times for both servers. There are two variables per remote server. T_{DS1_2} is the time taken to perform a search in the index file for the user request. The average time is 0.632 seconds. T_{nds} is the time taken for the overall operation before the reply is sent back to Dienst server 1, which initiated the request (for an explanation of what the parameters represent see Appendix A).

Print Parameters

Performance Parameters of Dienst Server 3

Parameter	Calls	RSP	Min	Max	Total	Mean	Std Dev
$T_{DS\ 1_2}$	5	0.617	0.574	0.677	3.162	0.632	0.039
T_{nds}^{local}	5	1.219	1.219	2.138	8.959	1.791	0.426

Performance Parameters of Dienst Server 2

Parameter	Calls	RSP	Min	Max	Total	Mean	Std Dev
$T_{DS\ 1_2}$	5	0.42	0.406	0.477	2.197	0.439	0.03
T_{nds}^{local}	5	0.796	0.69	1.682	6.456	1.291	0.502

Performance Parameters of Dienst Server 1

Parameter	Calls	RSP	Min	Max	Total	Mean	Std Dev
$T_{DS\ 1_1}$	5	15.16	15.16	17.776	80.873	16.174	0.964
$T_{DS\ 1_2}$	5	0.608	0.608	1.016	3.758	0.751	0.159
$T_{is_{Server\ 3}}^{Server\ 1->Server\ 3}$	5	11.527	11.527	13.968	62.964	12.592	0.879
$T_{is_{Server\ 2}}^{Server\ 1->Server\ 2}$	5	9.782	8.784	11.97	50.698	10.139	1.156
$T_{is_{Server\ 1}}^{Server\ 1->Server\ 1}$	5	10.553	2.062	11.793	36.841	7.368	4.716
T_{nds}^{local}	5	1.295	1.053	2.367	7.859	1.571	0.581
T_{nds}^{total}	5	15.735	15.735	18.085	84.807	16.961	0.888
$T_{is_{Server\ 3}}$	5	5.424	5.424	7.651	32.408	6.481	0.804
$T_{is_{Server\ 2}}$	5	3.289	2.989	5.032	19.182	3.836	0.785
$T_{is_{Server\ 1}}$	5	4.507	3.987	5.976	23.343	4.668	0.805

Fig. 11. Web-based user interface to monitor the performance variables

Network delay between two Dienst servers can be computed by simply subtracting the remote request response time between two servers (stored in the remote database) from the remote request processing time (stored in the local database).

4.1 Locating performance inefficiencies via monitoring

Performance monitoring was used in profiling Dienst search queries and for studying the effect of parameters on their response time. The parameters studied are divided into two categories: (a) those that depend on the configuration of the Dienst server (e.g., index database size); (b) those that depend on the machine that hosts the server (e.g., memory size, cpu speed, etc) and the network delay (latency) to reach the server.

The testbed consists of four Dienst servers:

1. a Sun4c workstation with 48 MB memory and SunOS 4.1.2
2. an Alpha workstation with 32 MB memory and Digital Unix V3.2
3. an Alpha workstation with 64 MB memory and Digital Unix V3.2
4. a Sun4c workstation with 16 MB memory and SunOS 4.1.3.

The two Sun machines have the same index database (4644 words). The first Alpha machine indexes 10 979

words and the second 20 222 words. The load was heavy and artificially created. The load is generated by a perl script which submits keyword queries to the one of the Dienst servers. Even from our testing measurements, it became apparent that the performance of the Dienst version 4.1.4, that we tested, is seriously affected by:

– *Machine load (and other machine properties).* The first observation was that the high resource consumption of the current Dienst implementation imposes a limit on the rate of requests it can service. This request rate limit depends on the machine configuration (mostly the memory), and thus, factors that affect them, like the index database size and the complexity of the queries.
We can see the difference of request processing time over machine memory size:

Sun4c, Memory Size in MB	16	48
Request Processing Time	4.657 s	0.638 s

Also, a request that searches only one database needed 1.16 seconds when the database was on a different machine that was easily accessed (on the same local network) and 2.95 seconds when the call was to the same machine (to the local database), even though the databases had the same content and there was no network latency involved: the overhead that was introduced by creating a new process on the non idle machine is high.
The continuous forks of Dienst (one for every request to the Dienst server) and the fact that it is written in a scripting language, perl, significantly affects the system overhead.
– *Data Base size.* It seems that the decision of Dienst to hold the whole index database in memory, creating huge processes and swapping them out of the memory seriously affects its performance. In the following table we see the size of the Dienst process and a typical request processing time as a function of the size of the index:

Words in Index DB	2, 327	5, 326	10, 797	77, 105
MB of Dienst process	9.5	9.7	10.6	13.6
Request processing time	1.5 s	2.2 s	4.9 s	6.4 s

– *Request complexity.* Different types of queries need different processing times. Dienst seems to optimize queries up to 2 keywords, sacrificing performance on the other, more rare cases.
– *Network delays.* Typically, the most significant part of the request's time is the network latency, and is non flexible. The exact percentage of the Network Latency to the total request servicing time (as shown in the table below) depends on other factors, and mainly the index database size. It takes an index database almost seven times bigger in order to make the network latency part non-dominant in request servicing time.

Words indexed	10, 979	77, 105
Network Latency ÷ Total execution	70%	35%

The percentage of the Network Latency depends, also, on the network "distance" of the server initiating the request and the server servicing it. For example the use of a remote server instead of a local, added about 12 more seconds of the Network Latency time, as measured at that time.
– *Timeout setup.* The values of the different timeouts on the Dienst protocol also affect its performance. A long timeout value not only increases delays, but also may let Dienst processes compute results that nobody is waiting for.

4.2 Load balancing

Using this simple performance architecture in our testbed, we measured the delays in various components of the digital library system, such as local processing of the user request, network round-trip delays when the request is sent to the remote sites for processing, and remote site processing delays. Each server keeps statistics such as averages and variances of each performance variable that is defined. For the sake of simplicity, these variables are common for all kinds of queries. In general, simple queries need very small computational time when compared to the more complex multiple-keyword or form-based query. Keeping track of variables for each kind of request is computationally intensive.

Here we describe some implementation details in our load balancing experiments. We set up 8 servers, divided into 4 sets, each set having 2 servers that have identical content. Also, there is partial overlap between the content of the different sets of servers. The server sets and locations[5] were:

UoCrete-Greece	FORTH-Greece_1
UoColumbia-USA	FORTH-Greece_2
FORTH-Greece_3	FORTH-Greece_4
GMD-Germany	CNR-Italy

Here are a few simplifications that were used by our policy: When a digital library node holds data for many

[5] The four FORTH servers are pseudo-remote. This means that they add a random delay to each response so that their response time is similar to the really remote servers.

publishers, we assume that all questions to the publishers are independent. When a server can provide multiple publishers the performance of the server depends on all the publishers in question and the total load assigned to the node. For more detailed results, see [36].

In Fig. 12 we present two different experiments. In both experiments we show the distribution of the same set of requests[6] testbed. It is important to notice that changes in network and machine load are depicted in the distribution of the requests. For example, looking at the pair of the Italy and Germany servers, which are identical, we see that in the first experiment the network latency and thus the response times of both servers were almost the same, as the requests were distributed almost evenly between the two. On the contrary, in the second experiment the network latency to the server in Italy was significantly higher than that for Germany. Thus, the response times of the Germany server were constantly lower, so the requests were almost all routed to the server in Germany. A similar case is with the pair of the CSI 3 and CSI 4 servers, but in this case the difference is not due to network latency but because of other machine load that was introduced.

The exact point where the change in the environment (network and machine) and thus to the response time takes place can be better seen by analyzing the request distribution between the pairs of identical servers: the plateau in Fig. 13 represents the periods where the respective server was *not preferred*. So, by looking at the pair CSI3 – CSI4, we see that in the beginning both servers are equally preferred, and at some point in the middle the CSI 4 server was mostly preferred, due to bursts of load that were introduced to CSI 3. When this external (to the digital library system) activity resumed both servers became equally preferred again. On the other hand, the concurrent plateau in the Germany-Italy pair show a period during which both servers were unavailable.

Concluding, our experiments confirm that most requests are sent to the machine that is faster to access. Also, on machines with similar access time, a possible external load on a machine can change the target of most requests.

5 Conclusions

In this paper, we have provided a management architecture and performance framework for measuring and monitoring digital library systems. For the performance framework, we defined relevant performance parameters which captured the time spent in the various phases of a search request as it propagates through the system. We used these parameters to perform load balancing so as to improve the overall request response time. In addition, we designed algorithms for estimating and adapting to timeouts in distributed searches. Our mechanism design for the performance enhancement is fairly general and can be applied to similar digital library systems where indexing and searching is distributed.

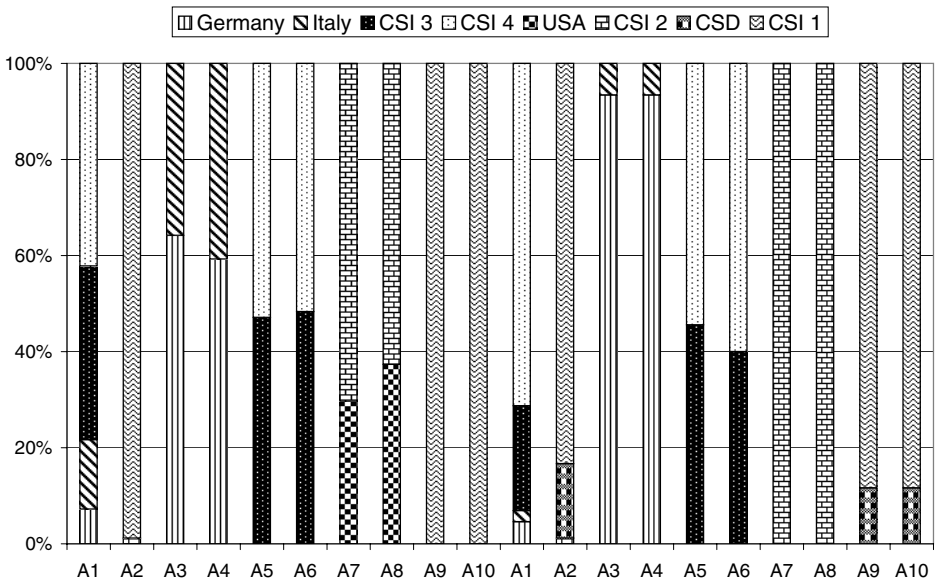

Fig. 12. Load balancing experiment results

[6] The set used consisted of 50 requests and was submitted twice. The requests were selected such that they simulate the Cornell University Dienst server load. More details about the experimental procedure can be found in [36].

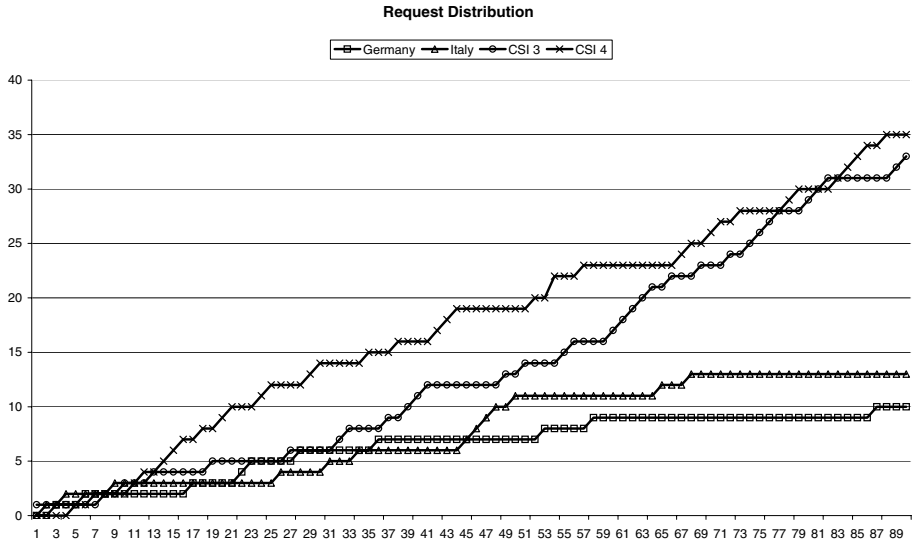

Fig. 13. Analysis of request distribution between pairs of identical servers

Our performance monitoring framework can be used in digital library systems to improve on the design, dynamic behavior, scaling, adaptive mechanisms and quality of service provisioning. Our architecture and mechanisms capture the time spent by the various tasks in a digital library server and helps in harnessing this information for load balancing user queries to servers. The emphasis in this paper is on mechanisms that can apply to a wide variety of digital library systems. We used an operational Dienst-based distributed digital library system as our testbed for performance monitoring. We gathered load data and other information from this system for our load balancing and adaptive timeout management.

Our future work is to design mechanisms for user-level analysis (session, account, access patterns), to examine compatibility with the Universal Measurement Architecture and to perform more experiments and find better policies for load balancing and timeouts. Finally, we would like to experiment with more complex performance enhancing policies that take into account empirical observations and can do some forecasting of load based on time of day. Our goal is also to explore more realistic data distributions for load balancing. Using these mechanisms, sophisticated policies can be exercised.

Appendix : Performance model for Dienst request processing

Each Dienst request goes through many stages, as seen in Fig. 14, where most of them are trivial components, but may introduce significant overhead.

The following is a mathematical model of the above operation of the Dienst servers in serving a query. The model is necessary to understand what we would like to measure and where, and what are the relationships between the modules that we measure. We model the time spent in "important" components of the Dienst server, as seen in Fig. 15. Each module has a time tag associated, which is nothing but the variable. The model is illustrated below, in a step by step example. For more information on the mathematical model, see [36].

Submit request modules

Our mathematical model also takes into consideration the maximum timeout periods involved. With this model we can better understand the procedures and the relationships between the components, prove properties about components and their execution, express performance results precisely, and trace problems.

Please note that the variable T_X^Y implies that T is the time spent in a module, where X is the subscript to denote a module and Y is the superscript to denote an interaction or a function.

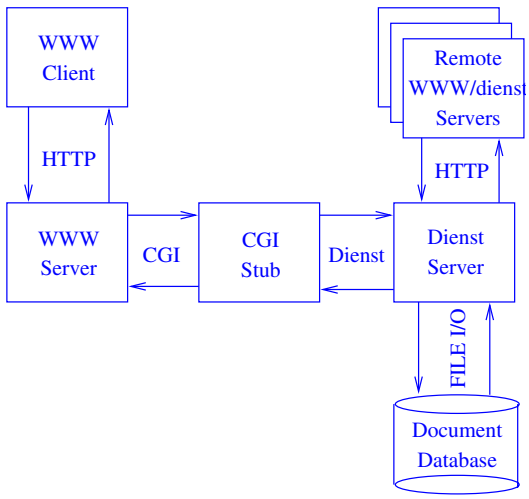

Fig. 14. The system view of the stages of a query

– T_{nds}: The time spent by the nph-dienst-stub (*nds*) to process the Dienst request sent by the browser. This is the time spent in submitting the request to $DS1_0$ and present the response in html format to the browser. Note that the Web server and the Dienst server are on different machines.
– $T_{nds} = T^{\text{socket}}_{nds->DS1_0} + T^{\text{fork}_1}_{DS1_0} + T^{\text{first}}_{DS1_1}$ (or $T^{\text{last}}_{DS1_1}$)
  – $T^{\text{socket}}_{nds->DS1_0}$: Time spent in the socket when nph-dienst-stub passes the request to $DS1_0$.
  – $T^{\text{fork}_1}_{DS1_0}$: Time $DS1_0$ needs to fork $DS1_1$ that will service the request.
  – $T^{\text{first}}_{DS1_1}$: Time taken by $DS1_1$ to submit the search and send the first results to the *nds* module. Note that $DS1_1$ sends the results to *nds* as they arrive from any of the *is* processes.
  – $T^{\text{last}}_{DS1_1}$: Time taken by $DS1_1$ to submit and complete the parallel search and send the whole result page (final result, or URL).

Begin parallel search modules

$T^{\text{first}}_{DS1_1} = T^{\text{fork}_{is}}_{DS1_1} + \min\{TO^{\text{search}}, \max\{T^{A->B}_{is_1}, T^{C->D}_{is_2}, T^{E->F}_{is_3}\}\}$

– $T^{\text{fork}_{is}}_{DS1_1}$: The time $DS1_1$ needs to fork all indexer_stubs: is_1, is_2 and is_3.
– TO^{search}: The total search timeout.
– $T^{A->B}_{is_1}$: The time taken by is_1 to send the query to $DS1_0$, receive the results and forward them to $DS1_1$. Similarly for the rest of the variables.

According to the probability of timeout, we have three cases:

– If none of the index servers respond, then the variables $T^{A->B}_{is_1}$, $T^{C->D}_{is_2}$, $T^{E->F}_{is_3}$ become infinity, and then $T^{\text{first}}_{DS1_1} = T^{\text{fork}_{is}}_{DS1_1} + TO^{\text{search}}$.
– If one of $T^{X->Y}_{is_j} > TO^{\text{search}}$ for $j = 1, 2, 3$, then the backup server is contacted, and the analysis continues (see [36]).
– If all of the servers respond before the time-out, then $T^{\text{first}}_{DS1_1} = T^{\text{fork}_{is}}_{DS1_1} + \max\{T^{A->B}_{is_1}, T^{C->D}_{is_2}, T^{E->F}_{is_3}\}$.

Processing request in local site

– $T^{A->B}_{is_1} = \min\{TO^{\text{search}}, (T_{is_1} + T^{\text{socket}}_{is_1->DS1_1})\}$
  – $T^{\text{socket}}_{is_1->DS1_1}$: The socket time to transfer results from is_1 to $DS1_1$.
– $T_{is_1} = T_{\text{Http}} + T^{\text{local}}_{nds}$
  – T^{local}_{nds}: The time *nds* (local) takes to send the query to $DS1_0$ and get the response in html format.
  – $T_{\text{Http}} = T^{\text{req}}_{\text{Http}} + T^{\text{rsp}}_{\text{Http}}$: The HTTP request and response time.
– $T^{\text{local}}_{nds} = T^{\text{socket}}_{nds->DS1_0} + T^{\text{fork}_2}_{DS1_0} + T_{DS1_2} + T^{\text{socket}}_{DS1_2->nds}$
– $T_{DS1_2} = \min\{T_{DS1_2}, TO^{\text{remote}}\}$
  – T_{DS1_2}: The time the second forked Dienst server needs to process a local query.
  – TO^{remote}: The local or remote database search timeout.

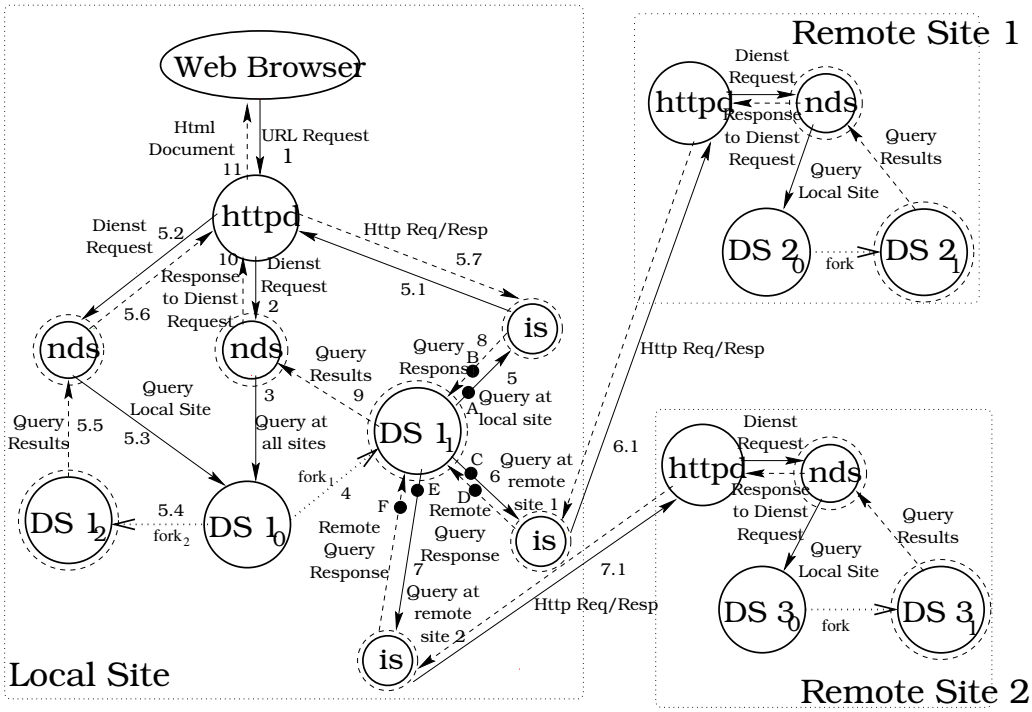

Fig. 15. The processes involved in a query processing

The analysis of $T_{is_2}^{C->D}$ and $T_{is_3}^{E->F}$ that follows, is similar to the analysis of $T_{is_1}^{A->B}$ above.

Processing request at remote site 1

- $T_{is_2}^{C->D} = \min\{TO^{\text{search}}, (T_{is_2} + T_{is_2->\text{DS1}_1}^{\text{socket}})\}$
- $T_{is_2} = T_{\text{Http}} + T_{nds}^{\text{remote1}}$
- $T_{\text{Http}} = T_{\text{Http}}^{\text{req}} + T_{\text{Http}}^{\text{rsp}}$
- $T_{nds}^{\text{remote1}} = T_{nds->\text{DS2}_0}^{\text{socket}} + T_{\text{DS2}_0}^{\text{fork}} + T_{\text{DS2}_1} + T_{\text{DS2}_1->nds}^{\text{socket}}$
- $T_{\text{DS2}_1} = \min\{T_{\text{DS2}_1}, TO^{\text{remote}}\}$

Processing request at remote site 2

- $T_{is_3}^{E->F} = \min\{TO^{\text{search}}, (T_{is_3} + T_{is_3->\text{DS1}_1}^{\text{socket}})\}$
- $T_{is_3} = T_{\text{Http}} + T_{nds}^{\text{remote2}}$
- $T_{\text{Http}} = T_{\text{Http}}^{\text{req}} + T_{\text{Http}}^{\text{rsp}}$
- $T_{nds}^{\text{remote2}} = T_{nds->\text{DS3}_0}^{\text{socket}} + T_{\text{DS3}_0}^{\text{fork}} + T_{\text{DS3}_1} + T_{\text{DS3}_1->nds}^{\text{socket}}$
- $T_{\text{DS3}_1} = \min\{T_{\text{DS3}_1}, TO^{\text{remote}}\}$

The same notation is used in the user interface shown in Fig. 11. For example, the parameters for Dienst server 1 represent the following:

- Response time of request from Dienst server 1 forwarded to Dienst server 3 ($T_{is_{\text{server3}}}^{\text{Server1->Server3}}$) is an average of 12.592 seconds for the 5 requests generated. This time is the sum of the round-trip network delay and the remote processing time of the request. Similarly for remote Dienst server 2 and the local search response time.
- Remote request response time for Dienst server 3 is represented by $T_{is_{\text{server3}}}$. The mean is 6.481 seconds. This includes the remote request processing time and the operating system overheads. Similarly for Dienst server 2 and 1 (local).
- Remote request processing time for Dienst server 3 is shown in the first table of the figure (T_{nds}^{local}). This is the time taken to search the index database at the remote server T_{DS1_2} and some overhead.
- Total request response time, T_{nds}, is the sum of all delays in collecting the responses. This is 16.961 seconds for the 5 requests that were generated.

References

1. Aral, Z., Gertner, I.: Non-intrusive and interactive profiling in Parasight. Proc. ACM/SIGPLAN PPEALS 21–30, 1988
2. Atkins, D.E.: The University of Michigan Digital Library Project: The Testbed. D-Lib Magazine 2(7), 1996; available at http://www.dlib.org/dlib/july96/07atkins.html
3. Birmingham, W.: An agent-based architecture for digital libraries. D-Lib Magazine 1(1), 1995; available at http://www.dlib.org/dlib/July95/07birmingham.html
4. Continuous Profiling Infrastructure.: Product Description, Digital Equipment Corp., 1997
5. Digital Libraries. Special issue, Communications ACM 38(4): 23–96, 1995
6. Fenske, D.E., Dunn, J.W.: The VARIATIONS Project at Indiana University's Music Library. D-Lib Magazine 2(6), 1996; available at http://www.dlib.org/dlib/june96/variations/06fenske.html
7. Ferguson, D., Georgiadis, L., Nikolaou, C.: Satisfying response time goals in transaction processing. Proceedings of the 2nd International Conf. on Parallel and Distributed Information Systems, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 138–147
8. Ferguson, D., Sairamesh, J., Cieslak, R.: Black holes, sacrificial lambs, and a robust approach to transaction routing. Proc. Int. Conf. on Parallel and Distributed Computing, 1995
9. Fox, E.A., Eaton, J.L., McMilan, G., Kipp, N.A., Weiss, L., Arce, E., Guyer, S.: National Digital Library of theses and dissertations: a scalable and sustainable approach to unlock university resources; D-Lib Magazine 2(8), 1996; available at http://www.dlib.org/dlib/september96/theses/09fox.html
10. Gu, W., Eisenhauer, G., Kraemer, E., Schwan, K., Stasko, J., Vetter, J.: Falcon: On-line monitoring and steering of large-scale parallel programs. GIT-CC-94-21, College of Computing, Georgia Institute of Technology
11. Frew, J., Freeston, M., Kemp, R.B., Simpson, J., Smith, T., Wells, A., Zeng, Q.: The Alexandria Digital Library Testbed. D-Lib Magazine 2(7), 1996; available at http://www.dlib.org/dlib/july96/alexandria/07frew.html
12. Gu, W., Vetter, J., Schwan, K.: An annotated bibliography of interactive program steering. SIGPLAN Notices (ACM Special interest group on programming languages) 29(9):140–148, 1994
13. Harum, S.L., Mischo, W.H., Schatz, B.R.: Federating repositories of scientific literature. D-Lib Magazine 2(7), 1996; available at http://www.dlib.org/dlib/july96/07harum.html
14. Hearst, M.A.: Research in support of digital libraries at Xerox PARC. D-Lib Magazine 2(5), 1996; available at http://www.dlib.org.dlib/may96/05hearst.html
15. Kapidakis, S., Sairamesh, J., Terzis, S.: A management architecture for measuring and monitoring the behavior of digital libraries. Proc. 2nd European Conf. on Research and Advanced Technology for Digital Libraries, Crete, Greece, September 1998, pp. 95–114
16. Kilpatrick, C., Schwan, K.: ChaosMON-application-specific monitoring and display of performance information for parallel and distributed systems. Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, May 1991, pp. 57–67
17. Klavans, J.L.: New center at Columbia University for digital library research. D-Lib Magazine 2(3), 1996; available at http://www.dlib.org/dlib/march96/klavans/03klavans.html
18. Lagoze, C., Davis, J.: Dienst: an architecture for distributed documents libriaries. Communications ACM 38(4):47, 1995
19. Lagoze, C., Shaw, E., Davis, J.R., Krafft, D.B.: Dienst: Implementation Reference Manual, TR95-1514. Cornell University, May 5th, 1995
20. Lange, F., Kroeger, R., Gergeleit, M.: JEWEL: design and implementation of a distributed measurement system. IEEE Transactions on Parallel and Distributed Systems 3(6):657–671, 1992
21. Litzkow, M., Livny, M., Mutka, M.W.: Condor - a hunter of idle workstations. Proc. 8th Int. Conf. of Distributed Computing Systems, June 1988, pp. 104–111
22. Managing Application Performance with TME10. Technical White Paper, Tivoli Systems, 1996
23. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.H., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The paradyn parallel performance measurement tools. IEEE Computer 28(11):37–46, 1995
24. Nikolaou, C., Kapidakis, S., Georgianakis, G.: Towards a paneuropean scientific digital library. TR96-0167, Institute of Computer Science – FORTH, May 1996
25. Paepcke, A.: Summary of Stanford's Digital Library Testbed Design and Status. D-Lib Magazine 2(7), 1996; available at http://www.dlib.org/dlib/july96/stanford/07paepcke.html
26. Perfstat.: Product description, Instrumental Inc., 1996
27. POLYCENTER Performance Advisor for Unix.: Product Description, Computer Associates, Digital Equipment Corp., 1996

28. Reed, D.A., Olson, R.D., Aydt, R.A., Madhyastha, T., Birkett, T., Jensen, D.W., Nazief, B.A.A., Totty, B.K.: Scalable performance environments for parallel systems. In: Proc. 6th Distributed Memory Computing Conference, IEEE Computer Society Press, 1991, pp. 562–569
29. Reed, D.A., Noe, R.J., Shields, K.A., Schwartz, B.W.: An overview of the Pablo Performance Analysis Environment. Dept. Computer Science, University of Illinois, Urbana, IL, November 1992
30. Resource Management Facility.: IBM Corp., June 1995
31. Snodgrass, R.: A relational approach to monitoring complex systems. ACM Transaction on Computer Systems 6(2):157–195, 1988
32. Rusbridge, C.: The UK Electronic Libraries Programme. D-Lib Magazine 1(6), 1995; available at http://www.dlib.org/dlib/december95/briefings/12uk.html
33. Sairamesh, J., Kapidakis, S., Nikolaou, C.: Architectures for QoS based retrieval in digital libraries. Workshop on Networked Information Retrieval, SIGIR '96
34. Schroeder, B.A.: On-Line monitoring: a tutorial, IEEE Computer 28(6):72–78, 1995
35. Tantawi, A.N., Towsley, D.: Optimal static load balancing in distributed computer systems. J ACM 32(2):445–465, 1985
36. Terzis, S.: Performance monitoring in digital library systems. Master thesis, TR97-0210, Institute of Computer Science – FORTH, October 1997
37. UMA Technical Information.: Performance Management Working Group, March 1994
38. A+UMA Performance Data Manager Technical Overview, Amdahl Corp., October 1996
39. Witten, I.H., Cunningham, S.J., Apperley, M.D.: The New Zealand Digital Library Project. D-Lib Magazine 2(10), 1996; available at http://www.dlib.org/dlib/november96/newzealand/11witten.html
40. Yu, P.S., Leff, A., Lee, Y.H.: On robust transaction routing and load sharing. ACM Transactions on Database Systems 16(3):476–512, 1991