Design and implementation of a service oriented architecture for WebGIS systems

DIPLOMA THESIS

By

Konstantinos Katakis

Submitted to the Department of Electronic & Computer Engineering in partial fulfillment of the requirements for the Diploma Degree.

Technical University of Crete, Chania

Advisor: Assistant Professor Vasilis Samoladas

Co-advisor: Associate Professor Euripides Petrakis

Co-advisor: Professor Minos Garofalakis

November 2009

Contents

Chapter 1

Introduction

This thesis is based on Isoteia Map Server Portal [3], which is a Web GIS system designed for environmental protection of regions. A geographic information system (GIS) captures, stores, analyzes, manages, and presents data that refers to or is linked to location. In the strictest sense, the term describes any information system that integrates, stores, edits, analyzes, shares, and displays geographic information. In a more generic sense, GIS applications are tools that allow users to create interactive queries (user created searches), analyze spatial information, edit data, maps, and present the results of all these operations.

1.1 Area of interest

This thesis is focused on designing a service-oriented architecture for managing computational and storage resources of IMS. Our intention is to implement a low

scale computational grid capable to serve the demanding requests of a multi-user environment. The implementation of our distributed services is based on CORBA architecture. This decision was made because CORBA standard supports object oriented model data, interoperability between applications written in different programming languages and executing in different computer platforms.

Our intention is to convert the various visualization, analysis and storage services of IMS to CORBA services. This service oriented approach introduces to our system interoperable services, which will enable the extension of IMS system logic and the integration of new functionalities. These services would be adapted or upgraded independently, according to application needs.

1.2 Problem Issue

The main problem of IMS is that it was developed as a monolithic web application with GIS models, MapServer, Map Algebra and Http server executing all together on a single computer. As a matter of fact, a system like this cannot provide stable access to the various IMS portal functionalities and in the same time to execute computationally expensive GIS models, composing and projecting maps. The implementation of a service oriented architecture as described above and the transition from a single server to a low scale grid definitely accomplishes the computational needs of IMS. However, there are yet many questions unsolved. There is a need of a mechanism to control the storage resources of the system. This is very significant in order to avoid data redundancy and need-

less file transfer among the workstations of the grid. Moreover, this mechanism should include a version control system for the files, since they will be spread in different workstations, as well as, a unified file naming system to distinguish files throughout the storage devices of the system. Another issue is the adoption of a load balancing policy to distribute the workload.

1.3 Thesis Contribution

The goal of this thesis is the transition from a monolithic to a service oriented architecture with ultimate end to support the execution of various kinds of GIS models, which are computation intensive and resource demanding due to mass process of geo-data. The fulfillment of these aims will enable our system to assure:

- support of multiple concurrent users
- simultaneous execution of multiple, high CPU consuming programs
- load balancing of the visualization and analysis tasks
- maximum utilization of hardware resources
- reliable file handling mechanism
- integration of new functionalities

This will render a service to the scientific community, which is specialized in the process and the study of geo-data and will contribute to the further improvement of technologies which are focused on GIS issues.

Chapter 2

Related Work

2.1 Isoteia Map Server Portal

WebGIS is becoming prominent in spatial decision support applications, as it allows researchers and stakeholders to benefit from sharing, analyzing and visualizing large, up-to-date geospatial datasets with minimal effort and cost. Initial applications appeared roughly a decade ago and involved centralized dissemination of maps, first static and later dynamic (allowing pan/zoom as well as primitive layer composition). As the foundations of web technology matured, WebGIS applications could provide more sophisticated cartography and spatial visualization features [8], [9], [10], [11]. IMS [3] addresses the integration of open-source, open-standards software and state-of-the-art interactive web technology to develop an interactive web mapping portal for spatial analysis.

It demonstrates that, open-source software offers a level of flexibility, avail-

ability and lowered cost that is typically unavailable with commercial software, while an architecture and design based on open standards ensures system interoperability and data reusability.

The prototype system, IMS, aims at enhancing collaboration and decision making among researchers and stakeholders in environmental decision-making, being highly accessible, and requiring minimal computing expertise. IMS's capability of integrating and merging data layers of different natural variables by a stepwise application of complicated functions enables multi-criteria, multi-temporal, and multi-scale data analysis. Furthermore, IMS enables the cooperation of different task groups sharing a common geospatial data base. Results from one group are made instantly available to parallel working groups in an interactive way.

ISOTEIA Map Server (IMS) is a web-based spatial decision support system. IMS was developed to support spatial data analysis within the ISOTEIA project, which includes the generation of a number of GIS-supported alternative scenarios for specific case studies on strategic environmental assessments in different domains, such as surface water management, ecosystem protection, forest management, industrial siting, irrigation management, water supply optimization, and sustainable agriculture [12]. IMS provides data sharing, visualization of geospatial data, spatial decision support services, and Map Algebra (MA) as a tool for spatial analysis. MA refers to the use of images as variables in normal arithmetic operations [5].

2.1.1 Mapping Portal Architecture

A WebGIS architecture for spatial analysis must enable the agglomeration of disparate computation and storage resources into a unified operational framework. The deployed architecture must provide rich access to spatial data, support multiple concurrent users and allow data to be transformed, composed into maps, and annotated by the user. To support spatial analysis, the system must initiate and manage several concurrent, long-running, computationally demanding analysis processes. Scalability is of paramount importance, but without compromising ease of use.

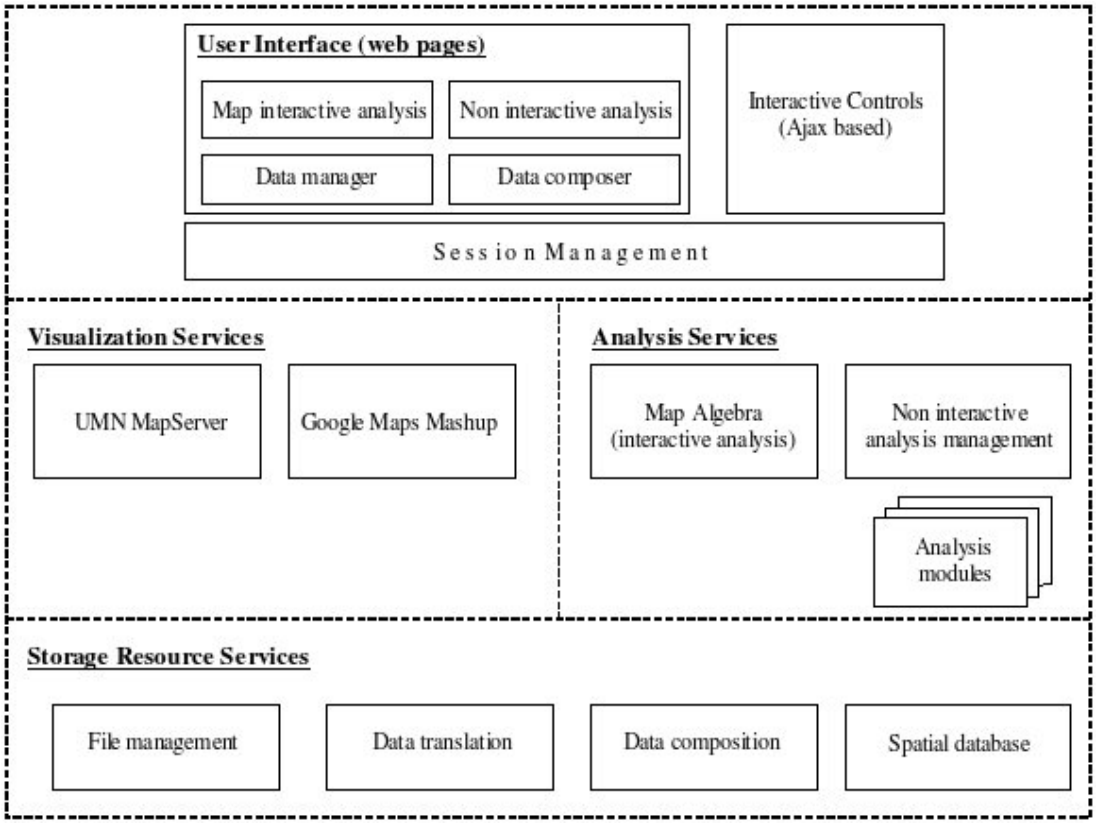

Figure 2.1: IMS architecture showing the main services grouped by layer.

Portal services are distributed into three tiers, namely the user interface tier, the processing tier, and the resource tier. Starting from the bottom, the storage resource tier comprises of two types of storage resources, file-oriented and database-oriented. File-oriented storage can be used both to store spatial data in files, and transform files among different file formats. Database-oriented storage is used for vector spatial data and spatial metadata. The processing tier encapsulates a number of visualization and spatial analysis services. Visualization services are used to render maps to be presented to the user, while analysis services can be initiated by the user to perform simple or demanding computations. Simple computations execute synchronously, i.e., the system returns the result to the user immediately, while demanding computations are performed asynchronously. The user interface tier is responsible for user management, authentication and user interaction. It layers a user interface on top of spatial data rendered by the visualization service using a mashup approach. The actual services are depicted in more detail in Figure 2.1.

2.1.2 User Interface

To compete with desktop applications, web applications must overcome two challenges related to interaction with the user: (a) they must provide a set of graphical controls embedded in the document-oriented web framework, and (b) they must minimize response time, by dividing processing between the browser and the server in order to reduce CPU and network transfer delays.

In IMS, user interaction is provided by a number of simple, intuitive web pages. There are two main web pages, the map interactive analysis page, used for spatial visualization, annotation and simple analysis, and the non-interactive analysis page, used for the configuration, monitoring and control of demanding analysis computations. These interfaces are implemented as AJAX (Asynchronous Javascript And XML) pages [6]. AJAX pages are web pages enhanced by substantial Javascript code executing at the web browser. This code is able to perform Remote Procedure Calls to the server, and modify only parts of the page displayed to the user, without significant delays (such as the delay of fully reloading a web page). The use of AJAX technology offers a level of interactivity and flexibility that approaches desktop applications. For example, using AJAX we have implemented graphical annotation and inspection tools on the spatial data displayed (distance measurement, cropping, feature selection, etc.)

2.1.3 Visualization Services

Visualization of spatial data can be challenging to implement. The portal must be able to synthesize spatial content from different sources into a coherent HTML-based document, on top of which a number of additional inspection and annotation tools can be overlaid. In order to minimize network traffic between browser and server, only the minimum necessary data is transferred. The transferred data is mostly in the form of raster images, with a small amount of additional metadata needed to configure the inspection and annotation controls.

This data is called the spatial component of a web page.

The role of visualization services is to prepare the spatial component for pages displayed to the user. Spatial component can be prepared either by rendering locally stored data (e.g., using UMN MapServer) or possibly by incorporating remotely accessible information (e.g., interfacing to Google Maps). The spatial component is then integrated with user interface components into a rich, AJAX-based HTML page using mashup techniques. The service-oriented architecture of IMS allows easy extension with additional visualization services, as long as these are wrapped by an appropriate service interface, allowing mashup integration with the rest of the system.

2.1.4 Analysis Services

Visualizing prepared spatial data may not be sufficient for sophisticated decision support; instead, it is often required to be able to analyze raw data in ad-hoc ways. IMS supports this functionality via two types of analysis tools: interactive and non-interactive (offline).

For interactive analysis, IMS supports a rich library of Map Algebra (Raster Algebra) operations. Included are thresholding, algebraic fitting, linearization, and histogramming operations. These tools operate synchronously, i.e. each tool is applied on the visible spatial data and the result is displayed to the user immediately. Note that some tools may take a long time (in the order of several seconds to tens of seconds) to operate, and then re-render the output via the

visualization service.

Non-interactive analysis involves long-running computations, possibly operating on massive amounts of data and consisting of several steps. The actual analysis software is external to IMS, running on remote platforms (e.g., high-performance parallel computers, computational grids) and requiring extensive configuration. Such analysis is supported in IMS by an extensible modular architecture, each module corresponding to a different analysis. Modules communicate with IMS via an interface, accessible via scripting extensions that can be used to prepare input data and customize analysis parameters, initiate and monitor the state of processing, and finally, collect the output data and make it available for visualization. Execution of analysis processes is asynchronous: non-interactive analysis jobs are not related to user sessions.

2.1.5 Storage resource Services

Spatial data exchange and management must be able to handle massive datasets, described by incompatible metadata and stored in a multitude of formats. IMS storage services offer effortless data exchange and management. Storage resources are either file systems or spatial databases. The bulk of IMS data is stored in files. However, a spatial database can be used to store (a) metadata related to spatial files, (b) annotations on spatial data entered interactively, and (c) metadata related to particular analysis services (in application-specific database schemas). File-based spatial data is managed through three types of services.

File management services are related to uploading/downloading/copying files, controlling access and organizing into directory hierarchies. Data translation services support transformation between over 40 spatial data formats (essentially those supported by the GDAL and OGR Simple Features Libraries). Finally, data composition services are used to construct assemblies of data, and augment them with the necessary metadata files, for particular visualization and/or analysis services. For example, a number of spatial raster and/or vector files can be composed into a new map (e.g., each file defining a map layer) to be rendered by the visualization service using UMN MapServer.

2.1.6 Implementation Issues

The architecture of IMS was implemented as a monolithic web and application server process, written in Tcl/Tk. This implementation had minimal overhead, but its scalability was limited, as computationally demanding services would not be effectively dispatched to additional computational resources. Performance has been a major consideration in designing the new architecture, since geospatial datasets of landscape units, such as watersheds of tens of km2, are big enough to require extensive computation times. By moving to a Service-Oriented Architecture, it is possible to scale to the required amount of resources that can support system load.

A major concern addressed by the architecture of IMS is extensibility, in order to take advantage of the wealth of open-source geospatial software available

today, as well as publicly available spatial services. To this end, IMS core is implemented on a portable TCL/Tk-based environment and can be deployed in most popular operating systems and hardware platforms. In addition, all major IMS components are implemented by open-source software, including the Google AJAX engine, the Postgres relational database with PostGIS extensions, the MapServer system from the University of Minnesota, the GDAL library for manipulation and translation of spatial data, the NAP system [7] for matrix algebra operations, and others. These are superior-quality components, free of cost and licensing restrictions, continuously supported and upgraded. Thus, IMS deployments can be done cheaply and easily on available hardware, but with the possibility to scale as application needs increase and to evolve as the technology standards change.

2.2 Grid Computing

Grid computing is the act of sharing tasks over multiple computers. Tasks can range from data storage to complex calculations and can be spread over large geographical distances. Unlike conventional networks that focus on communication among devices, grid computing takes advantage of the unused processing cycles of all computers in a network, for solving problems too intensive for any stand-alone machine. Grid computing can provide effective solutions for commercial, academic and personal problems.

These computers join together to create a virtual supercomputer. Computers

networked together can work on the same problems, that traditionally were reserved for supercomputers, and yet this network of computers are more powerful than the super computers built in the seventies and eighties. Modern supercomputers are built on the principles of grid computing, incorporating many smaller computers into a larger whole.

The ideas of the grid (including those from distributed computing, object-oriented programming, and Web services) were brought together by Ian Foster, Carl Kesselman, and Steve Tuecke, widely regarded as the fathers of the grid [13]. They led the effort to create the Globus Toolkit [14] incorporating not just computation management but also storage management, security provisioning, data movement, monitoring, and a toolkit for developing additional services based on the same infrastructure, including agreement negotiation, notification mechanisms, trigger services, and information aggregation. While the Globus Toolkit remains the de facto standard for building grid solutions, a number of other tools have been built that answer some subset of services needed to create an enterprise or global grid.

In fact, grid can be seen as the latest and most complete evolution of more familiar developments such as clusters, the Web, peer-to-peer computing and virtualization technologies.

- Like the Web, grid computing keeps complexity hidden: multiple users enjoy a single, unified experience.

- Unlike the Web, which mainly enables communication, grid computing en-

ables full collaboration toward common goals.

- Like peer-to-peer, grid computing allows users to share files.
- Unlike peer-to-peer, grid computing allows many-to-many sharing not only files but other resources as well.
- Like clusters, grids bring computing resources together.
- Unlike clusters, which need physical proximity and operating homogeneity, grids can be geographically distributed and heterogeneous.
- Like virtualization technologies, grid computing enables the virtualization of IT resources.
- Unlike virtualization technologies, which virtualize a single system, grid computing enables the virtualization of vast and disparate IT resources.

2.3 Load Balancing

CORBA is increasingly popular as distributed object computing middleware for systems with stringent quality of service (QoS) requirements, including scalability and dependability. One way to improve the scalability and dependability of CORBA-based applications is to balance system processing load among multiple server hosts. Load balancing can help improve system scalability by ensuring that client application requests are distributed and processed equitably across a group of servers. Likewise, it can help improve system dependability by adapting

dynamically to system configuration changes that arise from hardware or software failures. Load balancing can be distinguished into three types, Network-based, OS-based and Middleware-based.

Network-based load balancing is provided by IP routers and domain name servers (DNS) that service a pool of host machines. For example, when a client resolves a hostname, the DNS can assign a different IP address to each request dynamically based on current load conditions. The client then contacts the designated backend server, unaware that a different server could be selected for its next DNS resolution. Routers can also be used to bind a TCP flow to any backend server based on the current load conditions and then use that binding for the duration of the flow. High volume Web sites often use network-based load balancing at the network layer (layer 3) and transport layer (layer4). Layer 3 and 4 load balancing use the IP address/hostname and port, respectively, to determine where to forward packets. Load balancing at these layers is somewhat limited, however, by the fact that they do not take into account the content of client requests. Instead, higher-layer mechanisms,such as the so-called layer 5 switching described below,perform load balancing in accordance with the content of requests, such as pathname information within a URL.

OS-based load balancing is provided by distributed operating systems via clustering, load sharing, and process migration [15] mechanisms. Clustering is a cost effective way to achieve high-availability and high-performance by combining many commodity computers to improve overall system processing power.

Processes can then be distributed transparently among computers in the cluster. Clusters generally employ load sharing and process migration [16]. Balancing load across processors, or more generally across network nodes, can be achieved via process migration mechanisms, where the state of a process is transferred between nodes. Transferring process state requires significant platform infrastructure support to handle platform differences between nodes. It may also limit applicability to programming languages based on virtual machines, such as Java.

Middleware-based load balancing is performed in middleware, often on a per-session or per-request basis. For example, layer 5 switching [17] has become a popular technique to determine which Web server should receive a client request. ORB middleware allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware. Moreover, ORBs can determine which client requests to route to which object replicas on which servers. Middleware-based load balancing can be used in conjunction with the specialized network-based and OS-based load balancing mechanisms outlined above. It can also be applied on top of commodity-off-the-shelf (COTS) networks and operating systems, which helps reduce cost. In addition, middleware-based load balancing can provide semantically rich customization hooks to perform load balancing based on a wide range of application-specific load balancing conditions, such as run-time I/O vs. CPU overhead conditions.

2.3.1 Load Balancing Strategies

There are various strategies for designing CORBA load balancing services. These strategies can be classified along the following characteristics.

Client binding granularity

A load balancer binds a client request to a replica each time a load balancing decision is made. Specifically, a clients requests are bound to the replica selected by the load balancer. Client binding can be classified according to its granularity, as follows:

- Per-session Client requests will continue to be forwarded to the same replica for the duration of a session, which is usually defined by the lifetime of the client [18].

- Per-request Each client request will be forwarded to a potentially different replica, for example bound to a replica each time a request is invoked.

- On-demand Client requests can be re-bound to another replica whenever deemed necessary by the load balancer. This design forces a client to send its requests to a different replica than the one it is sending requests to currently.

Balancing policy

When designing a load balancing service it is important to select an appropriate algorithm that decides which replica will process each incoming request. For example, applications where all requests generate nearly identical amounts of load can use a simple round-robin algorithm, while applications where load generated by each request cannot be predicted in advance may require more advanced algorithms. In general, load balancing policies can be classified into the following categories:

- Non-adaptive A load balancer can use non-adaptive policies, such as a simple round-robin algorithm or a randomization algorithm, to select which replica will handle a particular request.

- Adaptive A load balancer can use adaptive policies that utilize run-time information, such as the amount of idle CPU available on each back-end server, to select the replica that will handle a particular request.

2.3.2 Load Balancing Architectures

By combining the strategies described above in various ways, it is possible to create the alternative load balancing architectures described below.

Non-adaptive per-session architectures

One way to design a CORBA load balancer is make to the load balancer select the target replica when a client/server session is first established, i.e. , when a client obtains an object reference to a CORBA objectnamely the replicaand connects to that object. Note that the balancing policy in this architecture is nonadaptive since the client interacts with the same server to which it was directed originally, regardless of that servers load conditions. This architecture is suitable for load balancing policies that implement round-robin or randomized balancing algorithms. Different clients can be directed to different object replicas by either using:

1. A middleware activation daemon, such as a CORBA Implementation Repository [19].

2. A lookup service, such as the CORBA Naming or Trading service.

Load balancing services based on a per-session client binding architecture can satisfy requirements for application transparency, increased system dependability, minimal overhead, and CORBA interoperability. The primary benefit of per session client binding is that it incurs less run-time overhead than the alternative architectures described below. Non-adaptive per-session architectures do not, however, satisfy the requirement to handle dynamic client operation request patterns adaptively. In particular, forwarding is performed only when the client binds to the object, i.e. , when it invokes its first request. Overall system

performance may suffer, therefore, if multiple clients that impose high loads are bound to the same server, even if other servers are less loaded. Unfortunately, non-adaptive per-session architectures have no provisions to reassign their clients to available servers.

Non-adaptive per-request architectures

A non-adaptive per-request architecture shares many characteristics with the non-adaptive per-session architecture. The primary difference is that a client is bound to a replica each time a request is invoked in the non-adaptive per-request architecture, rather than just once during the initial request binding. This architecture has the disadvantage of degrading performance due to increased communication overhead.

Non-adaptive on-demand architectures

Non-adaptive on-demand architectures have the same characteristics as their per-session counterparts described above. However, non-adaptive on-demand architectures allow re-shuffling of client bindings at an arbitrary point in time. Note that run-time information, such as CPU load, is not used to decide when to rebind clients. Instead, clients could be re-bound at regular time intervals, for example.

Adaptive per-session architectures

This architecture is similar to the non-adaptive per-session approach. The primary difference is that an adaptive per-session can use runtime load information to select the replica, thereby alleviating the need to bind new clients to heavily loaded replicas. This strategy only represents a slight improvement, however, since the load generated by clients can change after binding decisions are made. In this situation, the adaptive on-demand architecture offers a clear advantage since it can respond to dynamic changes in client load.

Adaptive per-request architectures

This design introduces a front-end server, which is a proxy [20] that receives all client requests. In this case, the front-end server is the load balancer. The load balancer selects an appropriate back-end server replica in accordance with its load balancing policy and forwards the request to that replica. The front-end server proxy waits for the replicas reply to arrive and then returns it to the client. Informational messages,called load advisories, are sent from the load balancer to replicas when attempting to balance loads. These advisories cause the replicas to either accept requests or redirect them back to the load balancer. The primary benefit of an adaptive request forwarding architecture is its potential for greater scalability and fairness. For example, the front-end server proxy can examine the current load on each replica before selecting the target of each request, which may allow it to distribute load more equitably. Hence, this forwarding architecture

is suitable for use with adaptive load balancing policies. Unfortunately, this architecture can also introduce excessive latency and network overhead because each request is processed by a front-end server. Moreover, two new network messages are introduced:

1. The request from the front-end server to the replica.

2. The corresponding reply from the back-end server (replica) to the front-end server.

Adaptive on-demand architectures

In this architecture clients receive an object reference to the load balancer initially. Using CORBAs mechanisms the load balancer can redirect the initial client request to the appropriate target server replica. CORBA clients will continue to use the new object reference obtained as part of the LOCATION FORWARD message to communicate with this replica directly until they are redirected again or finish their conversation. Unlike the non-adaptive architectures described earlier, adaptive load balancers that forward requests on-demand can monitor replica load continuously. Using this load information and the policies specified by an application, a load balancer can determine how equitably the load is distributed. When load becomes unbalanced, the load balancer can communicate with one or more replicas and request them to use the standard CORBA LOCATION FORWARD mechanism to redirect subsequent clients back to the load balancer. The load balancer will then redirect the client to a less loaded replica. Upon receipt of

a LOCATION FORWARD message, a standard CORBA client ORB re-contacts the load balancer, which then redirects the client transparently to a less heavily loaded replica. Using this architecture, the overall distributed object computing system can

1. recover from unequitable client/replica bindings while
2. amortizing the additional network and processing overhead over multiple requests

This strategy requires minimal changes to the application initialization code and no changes to the object implementations (servants) themselves. The primary drawback with adaptive on-demand architectures is that server replicas must be prepared to receive messages from a load balancer and redirect clients to that load balancer. Although the required changes do not affect application logic, application developers must modify a servers initialization and activation components to respond to the load advisory messages mentioned above.

Chapter 3

Architecture

A GIS system is using data ,which size differs between one another but is commonly huge. This is a problem that will consume time, if we have to transfer data over a local area network or internet, as well as space if we maintain files in more than one storage devices. It is important therefore in the beginning of our design, to have a clear view of the way that the different services in our system could have access on local or remote disks.

The figure 3.1 represents our approach that a service running on a specific machine could have data access in more than one local and remote hard disks and also the system would be capable to add or remove disk links, while a service is executed.

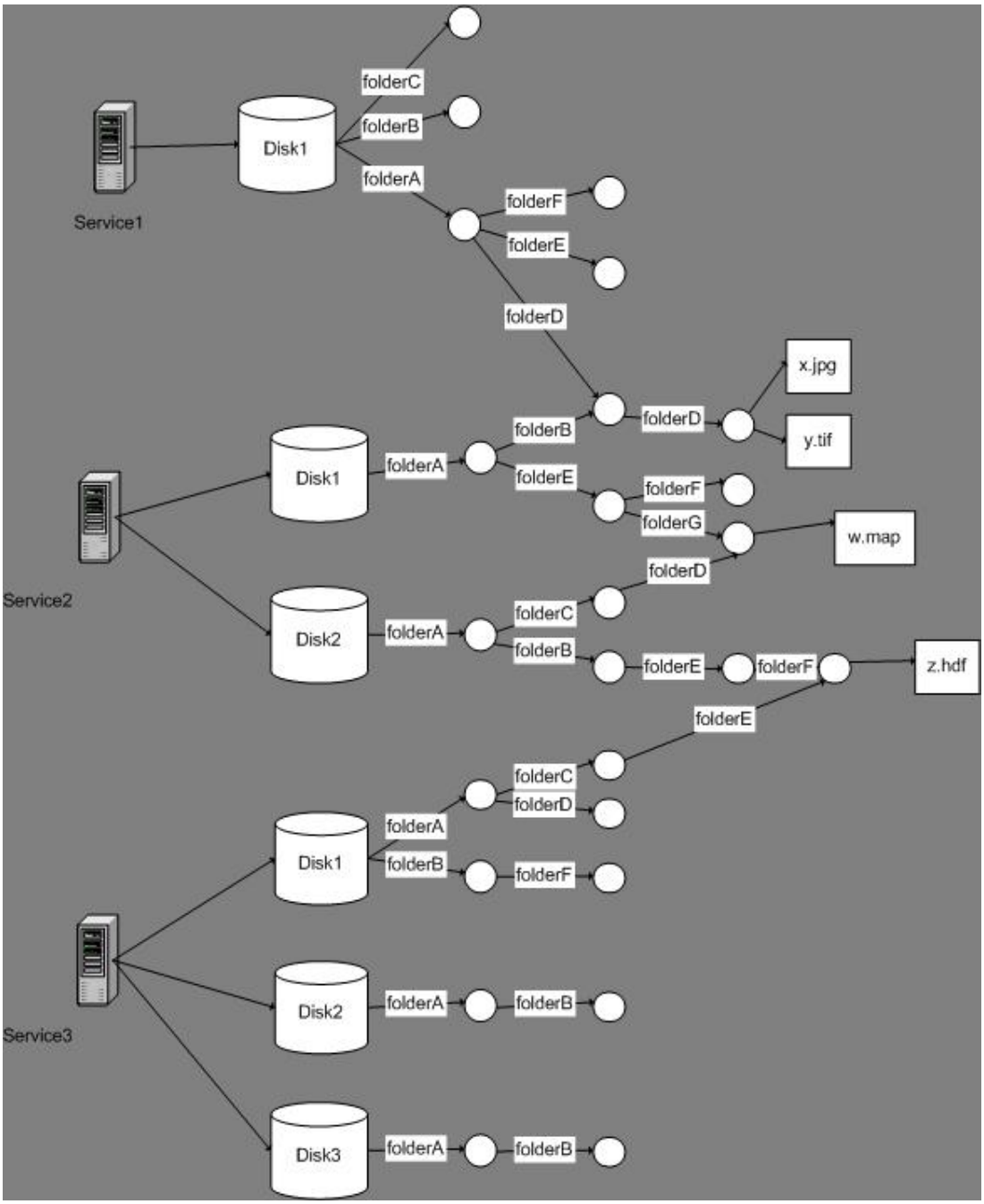

Figure 3.1: Distributed Filesystem

3.1 File Namespace

The services of the proposing architecture would be able to mount and unmount storage resources and each service could maintain several physical disks or even network disks. Moreover, files would be copied from a service's filesystem to another. These characteristics are raising filepath confusions of the system while retrieving files among various storage devices. In order to avoid them we are introducing an unified namespace of files (File Namespace). The designation of files is done by a specific URI scheme. The URI scheme is distinguished between physical and symbolic addresses. To be more specific a physical address has the form : ***ims://LFMName:diskId/filePath***. LFMName identifies a Local File Manager (LFM), it is a name of a service which manages storage resources and its functionality will be analyzed later. Moreover, diskId is a numerical identifier, which is unique for every local or remote storage device. Lastly, filePath indicates the name and optionally the location of a referenced file by specifying the directory containing the file, and other directories that may precede it in the system hierarchy. On the other hand symbolic addresses have the form: ***ims:///filePath***. A symbolic address can refer to another symbolic or to a physical one. Every symbolicDir is an arbitrary directory name or directory path, which is registered by a LFM service and it is unique throughout the system. Furthermore, some examples will be mentioned, based on figure 3.1, to demonstrate the use of file namespace :

Physical addresses locating the same file

- ims://service1LFM:1/a/d/d/x.jpg = ims://service2LFM:1/a/b/d/x.jpg
- ims://service2LFM:1/a/e/g/w.map = ims://service2LFM:2/a/c/d/w.map
- ims://service2LFM:2/a/b/e/f/z.hdf = ims://service3LFM:1/a/c/e/z.hdf

Mapping between global directories

- ims:///symbolic/data = ims://service2LFM:1/a/b/d

Symbolic mapping to physical address

- ims:///symbolic/data/y.tif = ims://service1LFM:1/a/d/d/y.tif

3.2 Service Oriented Architecture

First of all, we illustrate the general outline of our service oriented architecture. The basic components are the LookUpServer, the NapBalance, the LFMServers and the NapServers. All these components were implemented as CORBA services and are written with Python programming language, whereas TclHttpd session is the web server of our system. In figure 3.2 except from the basic services of our architecture, it is also introduced the communication relations among them.

The Object Request Broker we used for our implementation is omniORB which is a robust high performance CORBA ORB for C++ and Python. It is freely available under the terms of the GNU Lesser General Public License (for the libraries), and GNU General Public License (for the tools). omniORB is largely CORBA 2.6 compliant. The services communicate among them and with

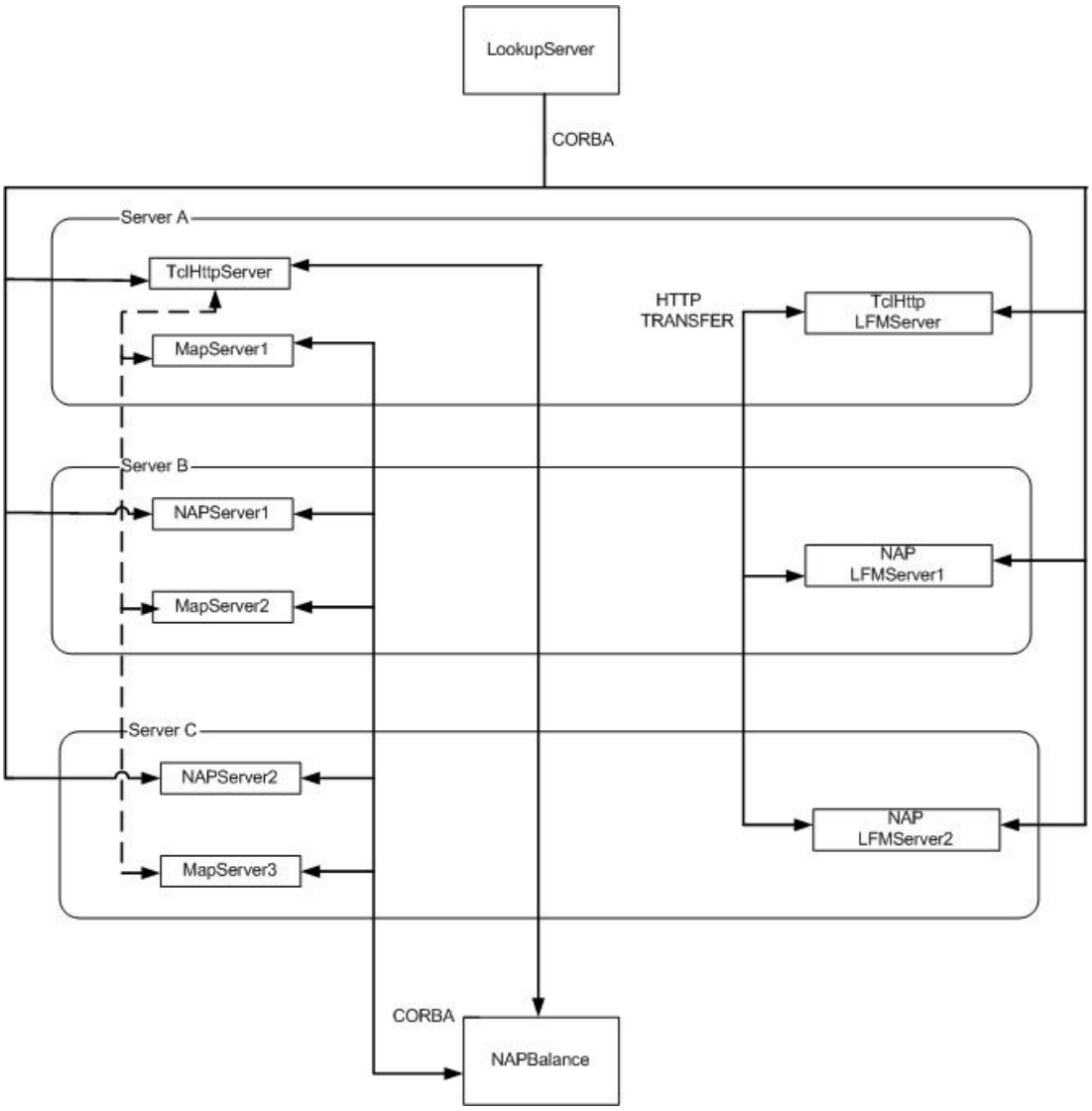

Figure 3.2: Service Oriented Architecture Overview

the clients, using the OMNI Naming Service (omniNames), which is an omniORB implementation of the OMG's COS Naming Service Specification. It offers a way for a client to turn a human-readable name into an object reference, on which the client can subsequently invoke operations in the normal way. The Naming Service stores a set of bindings of names to objects. These bindings can be arranged as an arbitrary directed graph, although they are often arranged in a tree hierarchy.

From the python library we used bsddb module, which provides an interface to the Berkeley DB library. Bsddb objects behave generally like dictionaries. Keys and values must be strings, however, so to use other objects as keys or to store other kinds of objects the user must serialize them somehow, typically using marshal.dumps() or pickle.dumps(). For storing list objects, pickle module was used for serializing. The lookupServer and the LFMServer are using the Btree file format of Berkeley DB, to keep records of information about system's storage resources. According to python's documentation, starting in Python 2.5 Berkeley DB interface should be safe for multithreaded access. Moreover, we make use of python's BaseHTTPServer module for file transfer among LFMs. This module defines two classes for implementing HTTP servers (Web servers). Usually, this module is not used directly, but is used as a basis for building functioning Web servers. For file transfer the HTTPServer was implemented using SimpleHTTPServer module, which defines a request-handler class interface-compatible with BaseHTTPServer.BaseHTTPRequestHandler, that serves files only from a base directory. In our attempt to add extra MapServers to our

system to distribute their C.P.U. and time consumption, we had to take into account that the MapServer program itself consists of only one file, the "mapserv" binary executable. This is a CGI executable, meant to be called and run by HTTP server. Python's CGIHTTPServer module was preferred for this task. This module defines a request-handler class, interface compatible with BaseHTTPServer.BaseHTTPRequestHandler and inherits behavior from SimpleHTTPServer.SimpleHTTPRequestHandler but can also run CGI scripts. It can also run CGI scripts on Unix and Windows systems, and on Mac OS it will only be able to run Python scripts within the same process as itself. BaseHTTPServer.HTTPServer class process requests synchronously, each request must be completed before the next request can be started. This characteristic does not take advantage the bandwidth of a Local Area Network (LAN) or any other network. For instance, if a large number of requests occur simultaneously, many of them will be queued and as a result their transfer time will be increased.. The solution is to create a separate process or thread to handle each request. The ForkingMixIn and ThreadingMixIn mix-in classes can be used to support asynchronous behavior. In our case a threading version of HTTPServer was created.

3.2.1 LookupServer

LookupServer keeps all the information of the system. It keeps record of all the other running services, of the symbolic links and global paths which are used by the LFMs, of the newest version of files that have been transferred and altered

in other storage devices and for every user a list of his personal files which are spread in several machines. The CORBA Interface Definition Language which defines the interface to LookupServer is illustrated below. The type listString is defined by us for input and output arguments.

```
typedef sequence <string> listString;

interface lookupServer{
  //search for a global address or sympolic link
  //and return physical address(es)
  listString searchFile(in string logicalName);

  //add links between global->physical, symbolic->global,
  //symbolic->physical addresses in mapping dictionary
  string addDiskMapEntry(in string logical,in string physical);

  //delete an entry from mapping dictionary
  string delDiskMapEntry(in string logical);

  //return ip address and file path for trasfering a file
  listString getHTTPArgs(in string machineName,in string diskID,in string filePath);

  //obtain for the asking session the current version of the file in uri
  string getFile(in string uri,in string session,in string userName);

  //register in lookupServer database a new file, a copied file or a new version
  //of an already existing file
  void registerFile(in string uri,in string session,in string action,in string userName,in string imgName);

  //delete file from database and promt all the lfmServers
  //to delete it from their scope
  string delFile(in string uri,in string name);

  //keep in local dictionary all the running services
  void registerService(in string name,in string ior);

  //return a pointer to extract from uri file name and file dir
  short splitUri(in string uri);

  //add a directory entry to hold its filelist
  void addDir(in string dirName);
```

```
    //add filename in directory's list
    void addFileList(in string dir,in string fileName);

    //del filename from directory's list
    void delFileList(in string dir,in string fileName);

    //call the appropriate lfm Service
    listString img_list(in string userName,in string type);

    //call LFM to check if a folder exists, if not create it and update local, global mapping
    void dirCheck(in string session,in string path);

};
```

Registration of Services

The lookupServer initializes, when it is activated, an object of type dictionary, which records a mapping between a service's name and it's object reference. Every service ,except http server, is calling lookup server's function registerService(..) to record itself. This dictionary object helps the synchronization of LFMs when a file has to be transferred from one LFM to another via HTTP or when a file has to be deleted from every individual LFM scope. Although we use omniNames to invoke operations among the various servers on our system, we preferred to implement a naming service of our own, for LookupServer's internal use to achieve simplicity and reduction of omniNames calls for resolving object references.

Recording of LFMs mappings

In its initialization stage a LFMServer reads from a file information such as its global name, an IP address where its HTTP server listens and other atomic infor-

mation that separates it from other LFMs. Furthermore, in that initial file is kept the information about storing devices, which are accessible from it, as well as all the address bindings that is using. These address bindings are of type physical to physical, symbolic to symbolic and symbolic to physical. Every LFMServer after reading this contents it sends them to lookupServer to store them at mapping.db .This information aids lookupServer to synchronize the LFMs to reduce file transfer. The functions that manage this information are addDiskMapEntry(..) and delDiskMapEntry(..).

LookupServer Functionality

A file in IMS in order to be able to get accessed, transferred from one LFM to another or deleted has to be registered. Lookup server keeps a database record, in which stores information for every file.

Lookup server utilizes two database files. The information in these database files is handled like Python's dictionaries. The first one is fileRec.db, which has for key the file's URI and for value a list object, with items the LFMName, which belongs to the LFM service that handles the file, and a version identifier. This information aids lookup server to locate, in which LFM scope is stored the newest version of a file. The other one is dirList.db, which has for key a user's name and for value a list object, which items are the URIs of all the files this user is handling. This information prevents lookup server from sending a request to all LFMs asking for a user's files.

The functions registerFile(..), getFile(..), delFile(..) make use of fileRec.db and because they are modifying it, are implemented as monitors. To be more detailed about their functionality and interaction with fileRec.db we illustrate an example of their use. Firstly about registerFile(..), when a new file enters IMS, whether it is uploaded or produced by a system function, a new entry is added to fileRec.db with key the URI of the file and value a list object with elements the LFMName the file belongs and version identifier equal to 1.The functions registerFile(..), getFile(..), delFile(..) make use of fileRec.db and because they are modifying it, are implemented as monitors. To be more detailed about their functionality and interaction with fileRec.db we illustrate an example of their use. Firstly about registerFile(..), when a new file enters IMS, whether it is uploaded or produced by a system function, a new entry is added to fileRec.db with key the URI of the file and value a list object with elements the LFMName the file belongs and version identifier equal to 1. For example, suppose that a file is uploaded in tclLFM's scope and another is created by NapServer1 and stored in its LFM scope. The following information will be stored in fileRec.db.

Table 3.1: fileRec.db Example 1:Insertion of a file

Keys	Values
ims://tclLFM:1/data/upload.jpg	[tclLFM ,1]
ims://nap1LFM:1/user/output.jpg	[nap1LFM ,1]

If later on this file is requested by a NAPServer or any other service of our

architecture and is transferred to its LFM, registerFile(..) is called to record the copy of this file to the requesting LFM scope. If the specified file had no entry at the requesting LFM, a new entry will be added. Otherwise, if the requesting LFM had an outdated version of the file, the version identifier of the entry will be updated. As shown in the following example the newest version of a file is transferred to the requesting LFM.

Table 3.2: fileRec.db Example 2:Copying of file

Keys	Values		
ims://tclLFM:1/data/upload.jpg	[tclLFM ,1]	[nap1LFM ,1]	
ims://nap1LFM:1/user/output.jpg	[nap1LFM ,1]	[nap2LFM ,1]	
ims://tclLFM1:/data/file.jpg	[tclLFM ,1]	[nap1LFM ,2]	[nap2LFM ,2]

If finally a NAPServer modifies a file through one of its functions, registerFile(..) is called to increase the version identifier of the file to the corresponding LFM.

Table 3.3: fileRec.db Example 3:Modification of a file

Keys	Values	
ims://tclLFM:1/data/upload.jpg	[tclLFM ,1]	[nap1LFM ,2]
ims://nap1LFM:1/user/output.jpg	[nap1LFM ,1]	[nap2LFM ,2]

In case a NAPServer is calling the getFile(..) function requesting a file the following actions will take place. First of all, the function will retrieve from the database the LFMName in which lies the newest version of the asked file.

Secondly, it will examine if the LFMName that was retrieved serves the calling NAPServer. If the correspondence exists, it will respond NAPServer that has access to the file. In a different case, it will prompt NAPServer's LFM to get the file supplying it with the file's URI and information about the LFM where the file is stored. Finally, the LFM will follow a procedure, which is analyzed later, and will answer to the lookupServer's getFile(..) with a boolean value, if access to the file was accomplished. This answer is then forwarded to the calling NAPServer.

A user has the option of deleting a file that he owns. When this action is taking place the delFile(..) is called. Its purpose is to make a list of all the LFMs that possess this file and send them a request for deleting it.

A user of a web portal like IMS is keeping and processing many files and is very useful for him to choose the file he wants from a drop down list instead of typing a filename. In our service oriented architecture the files of a user is possible to be spread in many storage devices, which are handled by different LFMs. The functions addUserList(..), delUserList(..), userList(..) make use of dirList.db and because they are modifying it they are also implemented as monitors. Their purpose is to manage a list with the URIs of the files of every user. This structure will prevent LookupServer from sending a request to all LFMServers asking them for a user's files and then joining and returning a list of URIs. The result will be better time performance, less network traffic and less CPU cost for the services of our architecture. To be more comprehensive, addUserList(..) is called only

when a file is uploaded or created by a NAP function and adds a file name to the user's list. The list isn't updated when a file is copied or modified. Function delUserList(..) removes a file name from a user's list in case he has deleted the corresponding file. Finally function userList(..) returns a list with the URIs of the files of a specific user. Below we illustrate an example of dirList.db's contents, where for simplicity's sake, we use simple filenames instead of displaying lists of URIs.

Table 3.4: dirList.db Example

Username	Filenames					
user1	file1.hdf	file2.hdf	out1.hdf	out2.jpg	results.jpg	file.shp
user2	map.hdf	map1.hdf	out1.hdf	out2.jpg	upload.hdf	modified.hdf
user3	data.hdf	data1.hdf	out1.hdf	out2.jpg	modified.hdf	file.hdf

3.2.2 LFMServer

The abbreviation LFM stands for Local File Manager. In the design of our system every service that handles files needs an associated LFM. The purpose of LFM is to maintain a list with all the global and symbolic address mappings, which it handles. Moreover, when a file's URI, of another LFM scope, is given from lookupServer, it has to check whether can obtain access through its global address mappings. Finally, when access to a file from another LFM scope cannot be achieved through address mappings, it has to send a request to the

HTTPServer of the other LFM in order to transfer the file to its own scope. The Interface Definition Language of LFMServer is illustrated below.

```
interface LFMServer{

  //calls lookupServer.searchFile and handles response
  listString searchFile(in listString fileList);

  //transfers a file with http
  string transferFile(in string machineName,in string fileName,in string username);

  //first is calling searchFile and then if is necesary transferFile
  string getFile(in listString lista,in string username);

  //add an entry for a link between topical and remote disks and for new topical disks
  //to mapping dictionary
  string addDiskMapping(in string localPath,in string remotePath);

  //delete an entry from mapping dictionary
  string delDiskMapping(in string localPath);

  //check if a folder exists and if not create it and update local and global mapping
  void dirCheck(in string path);

  //return a pointer to extract from uri file name and file dir
  short splitUri(in string uri);

};
```

LFM Initialization

The first function that is called when LFMServer is started, is reading from a file all the necessary information that LFM needs in order to be a part of the system. The contents of initFile are a unique name which distinguishes the LFMServer in the system and is stored in class attribute machineName. The name of LFMServer is commonly associated with the service that is accommodating. Other values of initFile are the IP address and the port number, which

HTTPServer is listening and are also stored in class attributes. The rest contents of initFile are the global and symbolic address mappings that the service is using, which are stored at mapping.db database file. After reading each of the previous values LFMServer calls LookupServer's function addDiskMapEntry(..) in order to store them at LookupServer's mappping.db. Secondly, we start a daemon thread with target function httpServer() and args LFMServer's IP and portNumber. The function httpServer() implements a HTTPServer for file transfer. From the Python library we used the necessary classes and modules, which were mentioned at the technologies that were used, to assemble a threaded HTTPServer. Finally, LFMServer is calling LookupServer's function registerService() to record its object reference.

LFM Functionality

As shown in figure 3.2 a LFMServer communicates only with LookupService with CORBA calls and with the other LFMServers via HTTP. It has not communication with the various services, whose file handling serves. In the analysis of lookupServer's function getFile(..) we had mentioned that if a NAPServer requests a file and lookupServer verifies that the latest version of that file is not stored in NAPServer's LFM scope, then lookupServer will call LFMService's function getFile(..) which serves the NAPServer. The function getFile(..) has two phases, firstly it calls LFMService's function searchFile(..) to check through address mappings if access to the file can be achieved. If this approach fails,

then function transferFile(..) is called to transfer the file through HTTP. At the end of function's getFile(..) execution, it responds to lookupServer whether file access was successed or not. The function searchFile(..) make use of LFM's address mappings in order to gain access to a file, which is stored in another's LFM scope. First of all, it parses the URI of the requested file, to obtain informations about the LFM that handles it, its diskID and its file path. In the second place it checks a list with all the LFM's address mappings, to figure out if there is a binding with desired LFM and diskID. Finally, it compares the file paths and if there is a correlation a final check is made using Python's built-in function os.access(..), which can test the existence and our permissions to the file. If function searchFile(..) fails to acquire access to the file, the control is obtained by function transferFile(..). This function establish a connection with the HTTPServer of the other LFMServer, then makes a request for the specific file and since the file is transferred the function stores it with the same name in the local scope. When the transportation is finished the fileRec.db of LookupServer is updated by calling its function registerFile(..).

It is already mentioned that LFMServer keeps its personal information in file mapping.db, which is created at the initialization stage. The contents of this database file are handled by methods addDiskMapping(..) and delDiskMapping(..). These methods provide the functionality to LFMServer to append or remove information about its address mappings. Whenever one of this methods are called to commit a change, LookupServer is subsequently updated in order

to have a balance in the system.

3.2.3 NapBalance

On our designing approach of the distributed system load balancing is very significant, because it can help improve system scalability by ensuring that client application requests are distributed and processed equitably across a group of servers. Likewise, it can help improve system dependability by adapting dynamically to system configuration changes that arise from hardware or software failures.

Load balancing architecture

In our implementation we adopted the logic of an adaptive per-request architecture for load balancing. This design introduces a front-end server, NapBalance, which is a proxy that receives all client requests. In this case the front-end server is the load balancer. Our system consists also of multiple back-end servers, NapService replicas, which are identical to each other and are processing requests that clients send over the network. The load balancer selects an appropriate back-end server in accordance with its load balancing policy and forwards the request to that server. The front-end server proxy waits for the servers reply to arrive and then returns it to the client. The primary benefit of an adaptive request forwarding architecture is its potential for greater scalability and fairness. For example, the front-end server proxy can examine the current load on each server

before selecting the target of each request, which may allow it to distribute load more equitably. Hence, this forwarding architecture is suitable for use with adaptive load balancing policies. Unfortunately, this architecture can also introduce excessive latency and network overhead because each request is processed by a front-end server. Moreover, two new network messages are introduced:

- The request from the front-end server to the back-end server
- The corresponding reply from the back-end server to the front-end server.

The load balancing policy that NapBalance adopts is based on the CPU load percentage of the NapServers. More simply, for every new client request the NapBalance obtains from a list the NapServer object with the less CPU consumption and forwards the request to it.

Moreover, NapBalance has additional duty to distribute the computational load of the MapServer. All the operations which are related to maps are deputed to MapServer. It can run as a CGI program or via Mapscript which supports several programming languages. In the initial design of IMS Portal the executable CGI file was located in the cgi-bin directory of TclHttpd server, which was handling MapServer as an ordinary CGI script in order to illustrate the requested map result. Because of the fact that accordingly to the data size that are given as input and the procedure that will be asked to be executed by the MapServer, it will maybe consume considerable CPU time and will delay in returning the requesting results. As an effect this delay will be displaced to the rest operations

of the web portal and eventually to the user-client. In this case was followed a non-adaptive per-session architecture design. Note that the balancing policy is non-adaptive, specifically we use round-robin scheduling, since the client interacts with the same server to which it was directed originally, regardless of that server's load conditions. By the term per-session we mean that since a MapServer composes a map after a client request, it will serve all the other requests on this specific map such as zoom in or out, surface measurement, layer addition or subtraction and the rest operations that are offered through map tool box. The Interface Definition Language of NapBalance is illustrated below.

```
interface napBalance{

  //register a cgi-bin/mapserv link
  void registerMapserv(in string mapserv);

  //return a cgi-bin/mapserv link
  string getMapServer();

  //get the consumed time of NAP services
  void napRusage();

  //announce the new cpu load
  void setNAPBalance(in string NAP_Id,in float cpuLoad);

  //keep in local dictionary all the running NAPServers by registering their names-objects
  void registerService(in string name,in string ior);

  //delete from local dictionary the specified NapServer
  void deleteService(in string name);

  //NAP functions are called through napBalanceServer after choosing the less busy server
  void ral_unary_rel(in string user ,in string source ,in string imgname ,in string img_func ,in string value);

  void ral_unary_functions(in string user,in string imgsource,in string imgname,in string img_func);

  string ral_binary(in string user,in string source1,in string source2,in string imgname,in string img_func);
```

```
  void ral_fuzzy_logic(in string user,in string imgsource,in string imgname,in string img_func);
  listString gen_img(in string user,in string imgsource,in string class);
  string img_info(in string user,in string imgsource);
};
```

NapBalance Initialization

At the initialization stage of NapBalance we set some service variables of great importance to the functionality of it. First of all, we initialize variable *services* which type is dictionary. This variable keeps NapServer's id as a key and its object reference as a value for every NapServer that is up and running. Furthermore, we set variable *napBalance* of dictionary type. The purpose of this variable is to store a pair of values for every active NapService. These values are the NapServer's id and the CPU percentage that its consuming. Another variable that we set is *mapservers*, which is of type list. The contents of this list are the URLs of the active MapServers, which are of type http://ip:port/path-to/mapserv-executable. The first locator that is added to the list *mapservers*, at the initialization stage, is the default Mapserver, which is called through TclHttpd server. Lastly, we set the variable *mapPtr* which is of type integer and is used as pointer to perform the round-robin scheduling of the registered MapServers. The first two are assisting NapBalance in the load balancing of the NapServer requests, while the last two aid NapBalance to distribute the load of the requests for MapServer.

NapBalance Functionality

The functions of NapBalance service are distinguished into three different groups according to their aim. The functions registerService(..), setNapBalance(..) are focused on implementing the balancing policy for the NapServer's calls. While registerMapserv(..) and getMapServer(..) are responsible for the MapServer's load distribution. Finally, there is a group of functions where each one of them correlate to a NapServer's function and their purpose is to be the intervener between all clients requests and the NapServer.

The function registerService(..) has two input arguments of string type. The first one is named NapId, which is a name that identifies uniquely a NapServer in the system. The other one is named ior, where IOR stands for Interoperable Object Reference, and from it the object reference of the NapServer can be constructed. This function is called by every NapServer after its initialization. In other words, we implement a local naming service in order to achieve simplicity and speediness by reducing the number of calls to omniORB's Naming Service. All the given input data to this function are stored on the dictionary variable *services*. In the second place, function setNapBalance(..) has also two input arguments. The first one is named NapId and contains the same information as described in the previous function. The second one is named cpuLoad and its type is float. This variable is carrying the percentage of the CPU consumption of the correlated NapServer. The information that is given to this function is stored on the dictionary variable *napBalance*.

The functions which perform the MapServer's load balancing are described below. To begin with, the function registerMapServ(..) takes one argument of string type and adds it in the list *mapservers*. To be more precise, the input argument is a locator to a web server, which has been started and handles MapServer's CGI. The web server that was chosen is Python's CGIHTTPServer. The other function is getMapServer(). This function takes no input arguments and is called to select one of the running MapServers following a round-robin policy as we have described.

Finally, there is six functions that are mediators between client requests and NapServes's functions. Their purpose is to select from the dictionary *napBalance* the NapServer's object with the lowest CPU consumption and forward to it the client's request. In case that the NapServer's function is returning values, the NapBalance's function has to wait for them and respond them to the client.

3.2.4 NapServer

NapServer contains all the raster functionality of IMS. These functions were originally placed at directory custom of TclHttpd server, where according to server organization there you put your own custom code and functions are automatically loaded by the server on startup. In these functions the basic role is hold by the Tcl package NAP. The abbreviation NAP stands for N-dimensional Array Processor, so it is obvious that this package is specialized for processing data in the form of n-dimensional arrays. NAP was developed using C and Tcl

programming languages. The first problem that it was faced, was the fact that NapServer was intended to be written in Python and NAP did not have any bindings with Python. However, the coupling with NAP package was achieved through Python's Tkinter module, which is a Python interface to Tcl/Tk. The second problem arise when we realized that NAP was not designed to be executed in multi-threaded environments. While omniORB is designed from the ground up to be fully multi-threaded. The solution was given by enclosing the blocks of NAP commands with a lock, which was not released until all the commands were finished. This modification resulted to a serialized execution of commands within a server and did not let us to take fully advantage of the characteristics of omniORB. The Interface Definition Language of NapServer is illustrated below.

```
interface NAP{

  //returns NAP consumed time
  void getRusage();

  //get global directory URI from input
  short getDir(in string fileName);

  //NAP functions
  void ral_unary_rel(in string user ,in string source ,in string imgname ,in string img_func ,in string value);

  void ral_unary_functions(in string user,in string imgsource,in string imgname,in string img_func);

  string ral_binary(in string user,in string source1,in string source2,in string imgname,in string img_func);

  void ral_fuzzy_logic(in string user,in string imgsource,in string imgname,in string img_func);

  listString gen_img(in string user,in string imgsource,in string class);

  string img_info(in string user,in string imgsource);

};
```

NapServer Initialization

At the initialization stage NapServer in order to make itself functional in the system, is registering firstly at the lookupServer and then to the NapBalance. The only matter that is left, before it begins receiving requests, is to start informing NapBalance about the CPU percentage that its consuming. This task is deputed to a daemon thread that runs a function which makes the appropriate checks and accordingly informs NapBalance. The function that control CPU consumption make use of Linux command ps, which gives informations of the current processes. Python gives the capability, through module commands, to execute a terminal command. First and foremost, the cpuLoad function informs the NapBalance of NapServer's CPU consumption. Then it enters to an infinite loop and whenever the CPU percentage is altered, it informs NapBalance.

NapServer Functionality

In this section is not going to be explained the raster functionality that preexisted and is not part of this work. However, additional control, that was encapsulated to take advantage of architecture functionalities, will be described. All of the six functions have to examine at the beginning of their execution whether they have access to the files that are given for process. This is achieved with a call to LookupServer and then will follow the control and transfer, if needed, procedure that is analyzed in sections of LookupServer and LFMServer. Since LookupServer responds positive, the NapServer's function can continue its command flow. At

the end of the function's execution the lookupServer will be informed if an already existed file was altered or if a new file was created to register it and keep the cohesion of the system. NapServer whether is acquiring a file via HTTP transfer or modifying it with one of its functions, informs lookupServer using its method registerFile(..). The distinction of the acts of copying and modifying is accomplished by an argument that is passed to registerFile(..) and refers to the type of registration.

Another point that is significant to mention is the functionality that is added to function gen_img(). The load balancing of the MapServer is actually taking place here. During the execution of gen_img() two files are created. The first one is isoteia.map and contains information about the dimensions of the map the layers that would be illustrated etc. The other one is init.js and contains descriptive information as isoteia.map and also defines the locator to MapServer CGI executable that will handle the map. Besides this information init.js is passing the right arguments and calling the Javascript functions that are composing the map. At this execution stage, where init.js is created, NapBalance is requested for an appropriate locator to serve the visualization part. At the end of the execution these files are returned to the TclHttpd server.

Another aspect of NapServer's functionality that is significant to mention is the exception handling in case a NapServer is out of the system. In this case it is very important that NapBalance will stop forwarding requests to it. To be more specific, if a hardware of software error occurs an exception is raised and

NapBalance's deleteService(..) method is called, which removes NapServer from the list of available servers.

In order to make our architecture more comprehensible a few examples will be illustrated to observe the sequence of function calls.

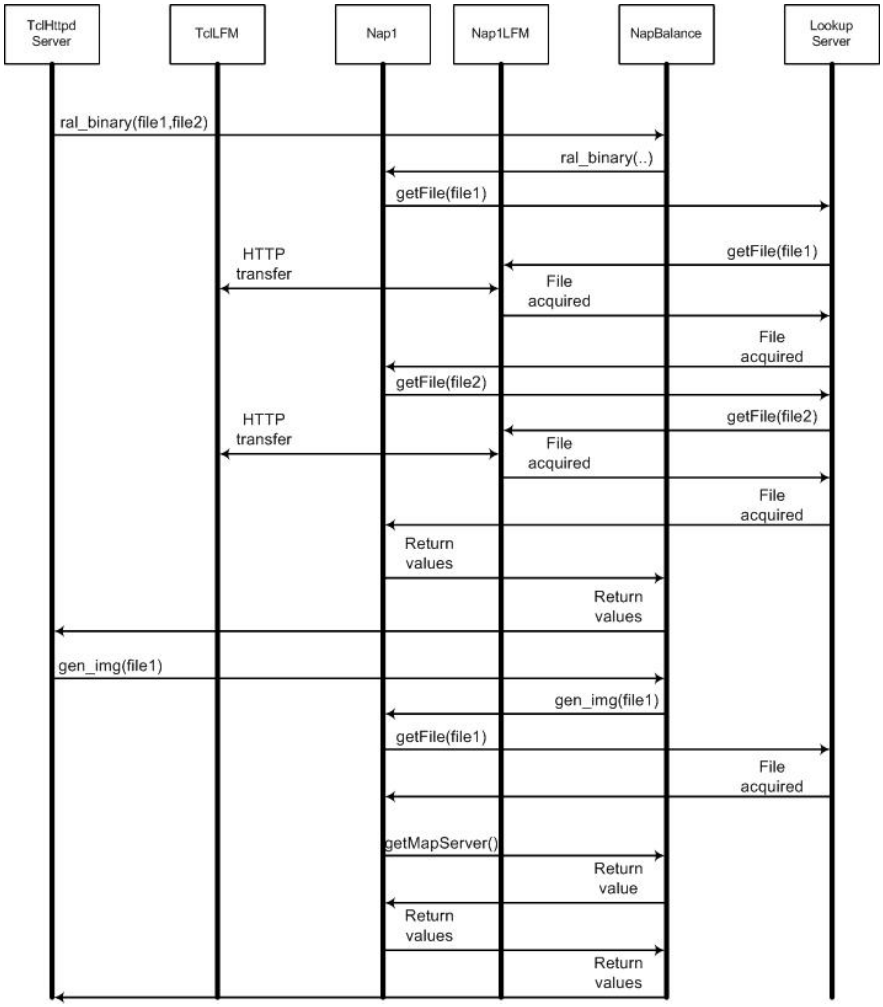

Figure 3.3: Call Flow Example 1

In Figure 3.3 man can observe the function call sequence that is followed after a client request. To be more descriptive, the client requests function ral_binary(file1,file2) which receives two filenames as input to process them. The TclHttpd server for-

wards the request to NapBalance which redirects it to Nap1 server. Then Nap1 make a call to lookupServer to assure file1 access. In the next step lookupServer realizes that Nap1LFM does not have the latest version of the requested file in its scope and informs it to acquire it. The file is transferred via HTTP and a positive response is sent from Nap1LFM to lookupServer and finally to Nap1. Exactly the same procedure is followed and for file2. When Nap1 server will finish its process the returning values, if any, will be passed to NapBalance, then to TclHttpd server and finally to the client. Afterwards there is a second client request for gen_img(file1). This time file1 is already acquired and there is no need of transferring it.

The example of Figure 3.4 is representing two almost simultaneous requests. NapBalance is distributing the load by forwarding each request to a different NapServer. The file transfer between Nap1LFM and Nap2LFM makes clear that a file can be positioned everywhere in the system.

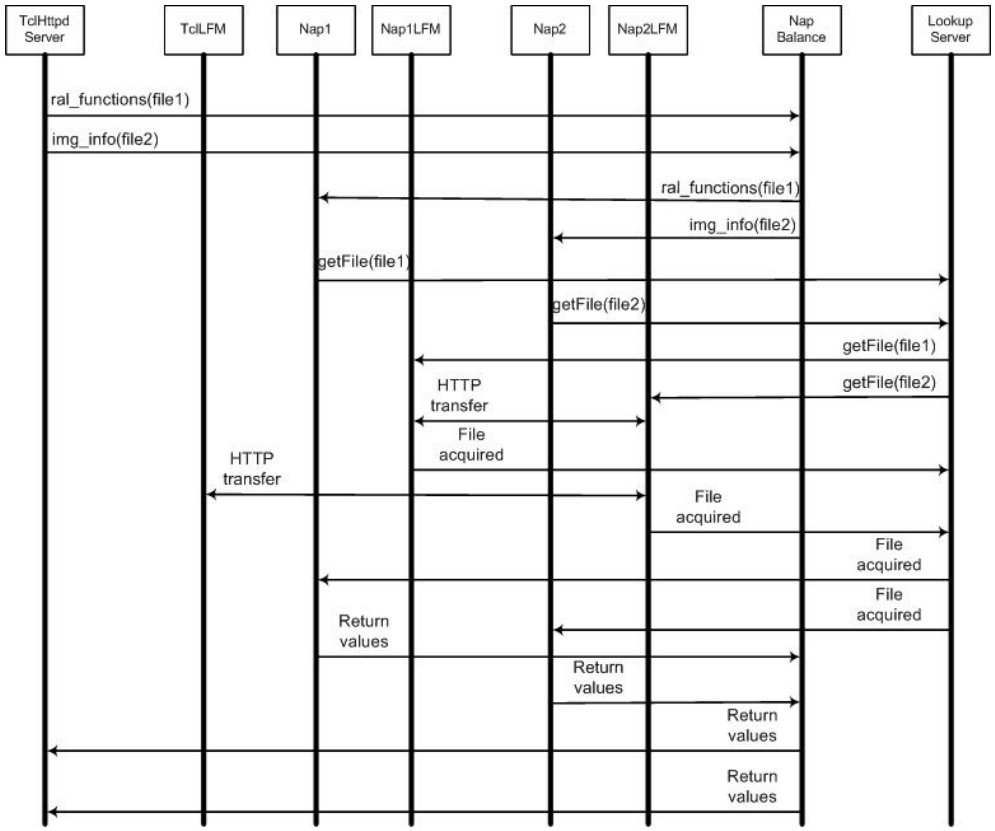

Figure 3.4: Call Flow Example 2

Chapter 4

Performance

Since the implementation of the services of our service-oriented architecture was finished and the correctness of the results was confirmed, there was a need of a benchmark scenario to test if the overall goals of our architecture were accomplished. This section describes the design and results of several experiments that were performed to measure the benefits of our architecture. Benchmarks performed for this measurement were run using ten 1799.977 MHz AMD Hammer Family processor workstations with 450584 kB RAM, all running Ubuntu 8.04 LTS Hardy Heron with Linux kernel version 2.6.24-16-generic. The processing and the plots of the received data was done with the assistance of program Octave. All workstations are connected through a 100 Mbps ethernet switch. The aim of the benchmark is to simulate a scenario where multiple users are using concurrently the raster functionalities of the IMS portal.

4.1 Benchmark Description

Firstly, we create thirteen treads to execute in parallelism their tasks. Each one of these threads is representing a different user and is invoking thirty nine NapServer's methods. In total 507 invocations to NapServer's methods are performed during the execution of a benchmark. The sequence of methods that each thread is invoking is selected in a random way, in order to avoid that the same methods with the same input file arguments will be called in a serialized way by every thread. The selection is made from a list which includes a pair of entries for every NapServer's function.

In order to have a clear view of the performance of our architecture, one to eight NapServers are used to run benchmark's scenario to extract conclusions. While a benchmark is executed, different data are recorded using system functions. These data consist of benchmark's total execution time, NapServer's execution time and the size of the transmitted files. The benchmark is executed ten times for every different number of NapServers to achieve accuracy of our results.

4.2 Benchmark Results

In the following graphic representations are illustrated the recorded data which were obtained through several Benchmark's executions. In figure 4.1 is displayed Benchmark's mean execution time graphic and in figure 4.2 Benchmark's speedup,

where speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm and is defined by the following formula $S_p = \frac{T_1}{T_p}$. It is obvious that using more than four NapServers is not improving significantly the performance of our system. The reason that the performance of the system is limited up to this point, can be accounted to the rise of network calls between the various servers and to the increase of process time of the synchronizing servers of the system. First of all, adding NapServers to the system increases the LookupServer's workload. It is receiving requests in a higher frequency rate and it has to assure that NapServers have access to their desired files, whether evolving their LFMServers or not. Moreover, the workload of NapBalance is increasing, due to the rise of the selection time between the running NapServers before sending them the request and the increase of the notifications of the NapServers about their CPU load. Furthermore, the file transfer through the network is raising, since more NapServers are utilizing data.

In figure 4.3 is displayed NapServer's speedup. This graphic representation is showing clearly that NapBalance is distributing equally the workload among the running NapServers. Figure 4.4 is illustrating NapServer's mean execution time and its standard deviation.

The figures 4.5 and 4.6 are illustrating the mean amount of transferred data in MB and the mean number of transferred files respectively.

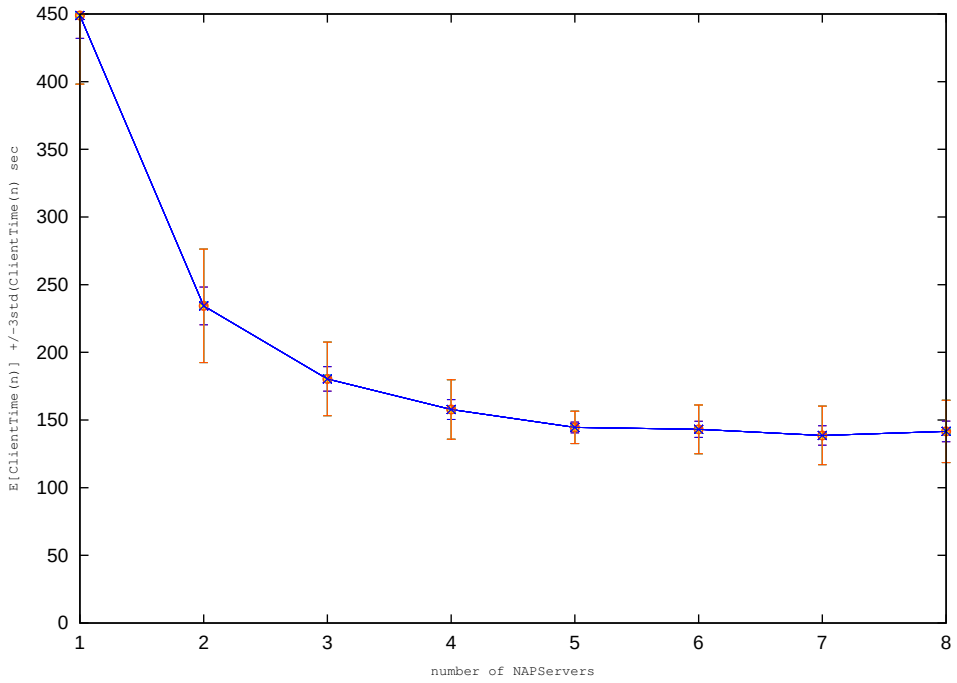

Figure 4.1: Benchmark's mean execution time +/- 3*σ

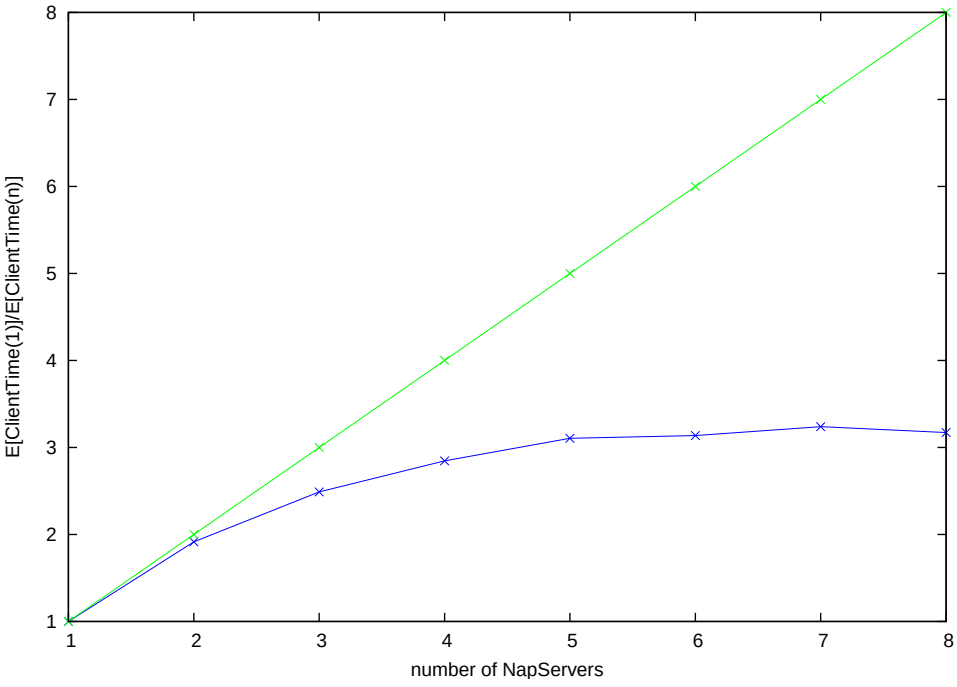

Figure 4.2: Benchmark's speedup

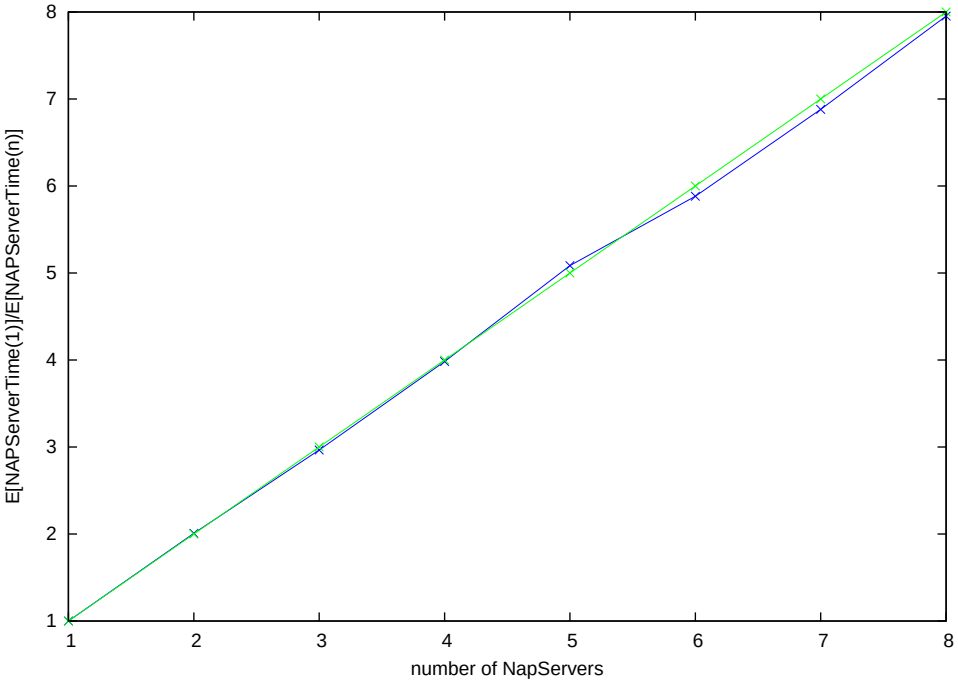

Figure 4.3: NapServer's speedup

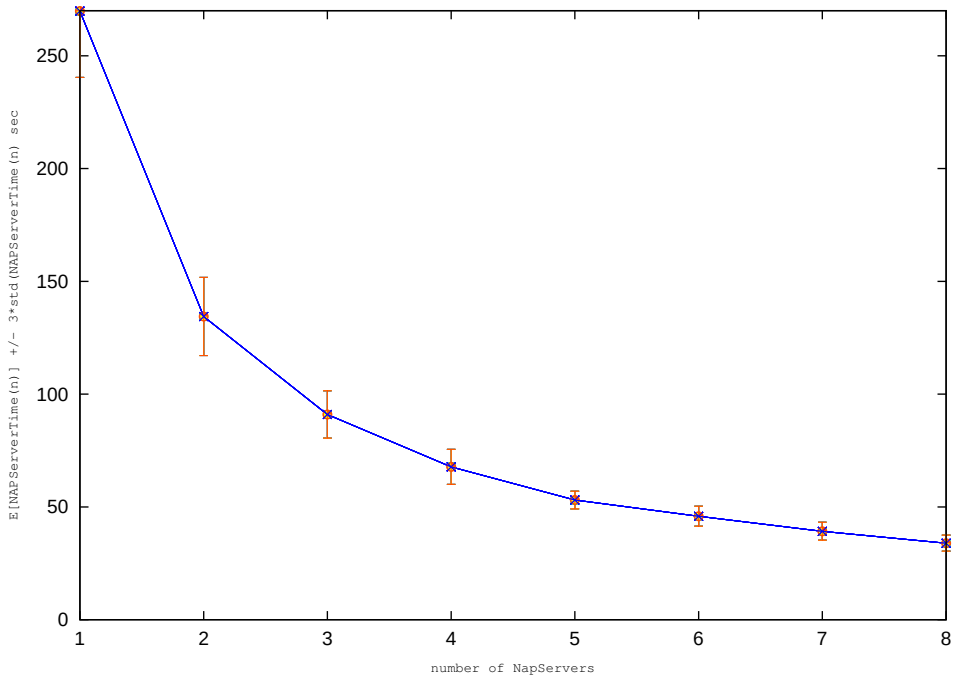

Figure 4.4: NapServer's mean execution time +/- 3*σ

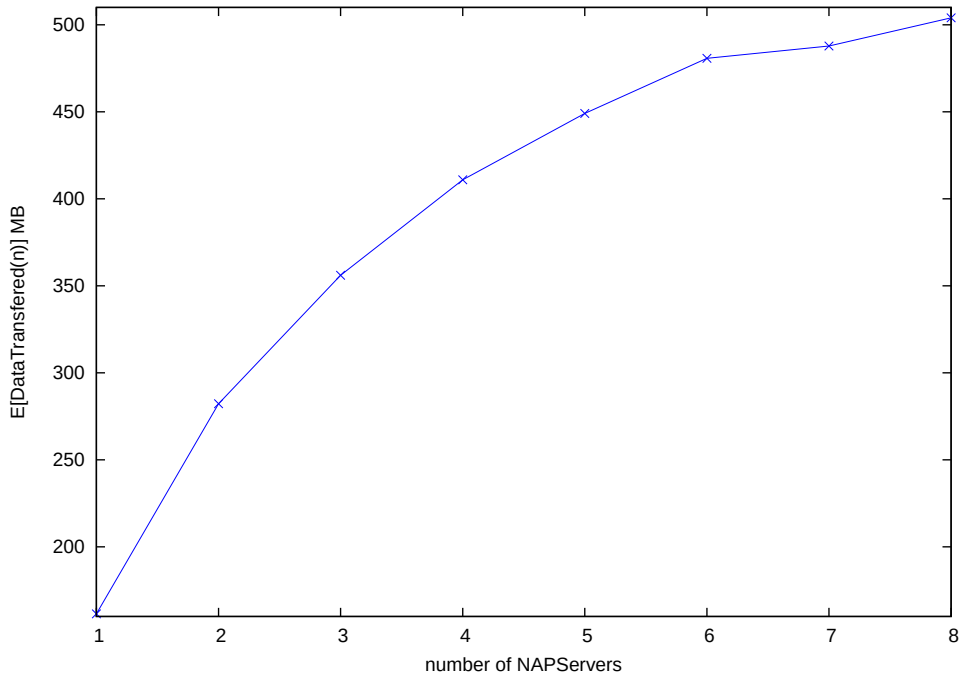

Figure 4.5: Mean amount of transfered data in MB, per number of NapServers

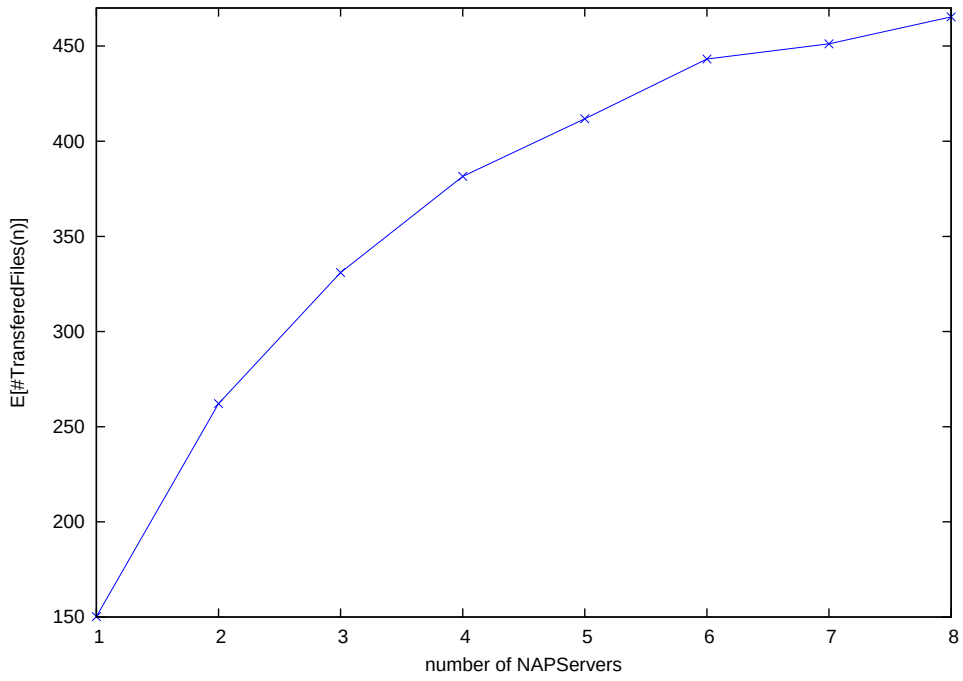

Figure 4.6: Mean number of transfered files, per number of NapServers

Chapter 5

Conclusions

In conclusion, the implemented service oriented architecture fulfilled its goals. It managed to transform IMS from a monolithic system to a low scale grid, by adjusting its functionalities to CORBA services. The control mechanism of system's storage resources adds scalability and enables integration of new GIS functionalities to the system. The load balancing policy offers better utilization of hardware resources and system dependability by adapting dynamically to system, configuration changes that arise from hardware or software failures.

There are a few matters which can be proposed for future work. First of all, it would be better to adopt an adaptive on demand load balancing policy. Using CORBAs mechanisms the load balancer can redirect the initial client request, which is queued waiting for execution to the firstly selected server, to the appropriate target server replica according to the CPU consumption policy. Unlike the adaptive per request architecture that we implement, adaptive load balancers

that forward requests on-demand can monitor replica load continuously. Using this load information and the policies specified by an application, a load balancer can determine more equitably the load distribution. Moreover, even in our service oriented architecture a user request may last considerable time. This situation for a portal's user, waiting in front of a frozen screen, is not the preferred. There is a need for a mechanism to permit the user to continue with other activities in the portal or even log out from it and not to wait until his tasks are completed. At the point where a user's request execution is completed, a notification mechanism will inform him, with a screen message if he is still logged in or via e-mail. Finally, another important issue is the maintenance of the outdated files. Since LookupServer manipulates the version control of the files, it would be useful to remove all the future unused files. This would lead to more available system's storage resources and reduced computational process for LookupServer.

Bibliography

[1] John K. Ousterhout, "Tcl and the Tk Toolkit", Addison Wesley

[2] Brent B. Welch, Ken Jones, Jeffrey Hobbs, "Practical Programming in Tcl and Tk", Prentice Hall 2003

[3] Salah Juba, Vasilis Samoladas, Nikos Boretos, Ioannis Manakos and Christos Karydas, IMS: a Web-based Map Server for Spatial Decision Support, Neural, Parallel and Scientific Computations, 15(2), 2007, 207–220

[4] Ossama Othman, Carlos ORyan, Douglas C. Schmidt, "An Efficient Adaptive Load Balancing Service for CORBA", Distributed Systems Engineering Journals Online March 2000

[5] Eastman J.R., 2003. IDRISI Kilimanjaro Tutorial

[6] Garrett J.J., 2005. Ajax: A New Approach to Web Applications

[7] Davis H., 2002. The NAP (N-Dimensional Array Processor) extension to Tcl, Proc. 9th Annual Tcl/Tk Conference

[8] Lu C.T., Y. Kou, H. Wang, S. Shekhar, P. Zhang and R. Liu, 2003. Two web-based spatial data visualization and mining systems: Mapcube & Mapview. Proc. Next-Generation Geospatial Information Systems (NG2I'03)

[9] MacEachren A.M., 2001. Cartography and GIS: Extending collaborative tools to support virtual teams. Progress in Human Geography, 25(3), 431-444

[10] Mitchell T., 2005. Web Mapping Illustrated. ISBN: 0596008651 OReilly

[11] Kraak M.J. and A. Brown, 2001. Web Cartography development and prospects. New York: Taylor and Francis Inc

[12] Karydas, C.G., G.C.Zalidis, L.Kouskouna, E.Feoli, N.G.Silleos, I.Z.Gitas, 2006. European Environmental Legal Framework and Decision Support towards Sustainable Development: ISOTEIA model. OTEM Summer School Proceedings, Baia Mare, Romania

[13] Fathers of the grid http://magazine.uchicago.edu/0404/features/index.shtml

[14] Globus Toolkit http://www.globus.org/toolkit/

[15] G. Coulouris, J. Dollimore, and T. Kindberg, Distributed Systems: Concepts and Design. Harlow, England: Pearson Education Limited, 2001

[16] F. Douglis and J. Ousterhout, Process Migration in the Sprite Operating System, in Proceedings of the 7th International Conference on Distributed Computing Systems, (Berlin, WestGermany), pp. 1825, IEEE, Sept. 1987

[17] E. Johnson and ArrowPoint Communications, A Comparative Analysis of Web Switching Architectures. http://www.arrowpoint.com/solutions/white papers/ws archv6.html, 1998

[18] N. Pryce, Abstract Session, in Pattern Languages of Program Design (B. Foote, N. Harrison, and H. Rohnert, eds.), Reading, MA: Addison-Wesley, 1999

[19] Adiron, LLC, et al., Portable Interceptor Working Draft Joint Revised Submission. Object Management Group, OMG Document orbos/99-10-01 ed., October 1999

[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture - A System of Patterns. Wiley and Sons, 1996