COMPONENT-BASED DIGITAL LIBRARY SCALABILITY USING CLUSTERS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

By
Muammar Zamir Omar
December 2011

Supervised by
Hussein Suleman

Abstract

Digital Libraries (DLs) are systems to manage information or data. They range from monolithic systems to loosely coupled component-based ones. DLs provide the services to manage, retrieve and access this information. Where they have fallen short is providing methods to manage huge volumes of information quickly and effectively. While the services provided by these systems work correctly, the time it takes to provide a response is unacceptable. In many cases, this is due to the underlying architecture of the DL system and other factors which influence resources available to the system owners for upgrades or maintenance.

This dissertation documents an alternate approach to the normal one of using more powerful machines to overcome the problem. Instead, a cluster of computers is used to provide increased performance. It presents a DL system in which loosely-coupled components can migrate and replicate across machines in order to meet the demands of the system. These components provide user services such as searching and browsing. Over time, the system adjusts itself automatically to provide better service times as the number of incoming requests increase. These adjustments include migrating components or services to machines with more resources and replicating those which are being queried constantly. The architecture introduced is one which can be created from most component-based DLs and is easily replicated.

Initial analysis and evaluation indicate that this system provides better performance under conditions of heavy load while maintaining good response times under minimal loads. This approach has thus proven to be a viable one for addressing performance degradation in an experimental environment and is ready for testing in a live environment.

Acknowledgments

This work would not have been possible without my supervisor, Dr Hussein Suleman. I would like to thank him for all the support, guidance, understanding and encouragement he has given me throughout my studies. He has not only served as a supervisor but also as a friend.

To my parents, I wish to convey my greatest thanks for affording me the opportunity to complete my post-graduate studies. They have been a constant support within my life since my undergraduate studies.

Thank you to all my fellow Masters students who have helped me along my journey. They have made my time much more pleasant and provided a good atmosphere for both academia and fun.

Thanks to the members of the Advanced Information Management (AIM) lab as well as the newly formed High Performance Computing (HPC) lab.

Additionally, I would like to thank Riaad Adams, Cecilia Augustine and Hema Jeaven for taking time to provide me with their assistance.

I would like to thank my wife for keeping me motivated and constantly ensuring that I am moving forward.

Finally, I would like to thank Allah (SWT) for allowing me to accomplish this.

.

Table of Contents

List of Tables

List of Figures

Chapter 1

1. Introduction

The Web has enabled us to make use of information all across the world. Whether this information is academic, commercial or any other type, it can be obtained using the Web. Most of this information needs to be managed somehow and many institutions make use of Digital Library (DL) systems to do this.

A major focus has been on which services can be provided to utilize this data to its full potential. These include services to manage, retrieve and access this information. This focus has not shifted much recently but the overwhelming volumes of data being received and created have added other aspects to utilizing this data: how quickly the data can be retrieved, as well as how quickly services can act on this data and provide results.

Where systems have fallen short is in providing methods to manage huge volumes of information quickly and effectively. While the services provided by these systems work correctly, the time it takes to provide a response is unacceptable.

1.1 Digital Libraries

DL systems have evolved from monolithic systems to be more robust component-based systems we find today. What all of them provide, though, is services which can be utilized over the Internet. These Web services access their underlying data and provide users with the ability to search, browse, discover and manage information.

While the performance of these systems decreases as the underlying data increases, we have not seen much focus on improving this aspect of these systems. While data size is the main reason for the degradation in performance, the architecture of the DL systems are to be blamed as well. Not much support for scalability has been built into these systems. The temporary solution, which is probably the most prominent, is to make use of faster machines. While this is a solution, it is only suffices until the underlying data increases again and the same problem recurs.

With the monolithic systems, we have seen performance degradation due to its design deficiencies. Many of these systems built from scratch and are tailor made to a specific community or organization. These types of systems are often hard to modify and incorporation of new modules or algorithms usually requires significant work. Maintenance costs are generally higher due to its internal design problems as well [1]. Additionally, copyrights on digital material limit the content freely available and DL systems harvest the metadata from various other DL systems. As this content is searched by the users, it becomes necessary to use complex algorithms for information retrieval. These factors are likely to affect performance and system scalability on DL systems [2].

With the increased availability of high speed networking, grid technologies provide a cheap way to gain performance. This has been used in conjunction with DL systems in order to overcome some of these problems. In developing countries, such as South Africa, however these networks are not easily accessible or available and alternative solutions are needed.

This research is aimed at tackling this problem by using a DL system with components that can migrate and replicate across machines. This dynamic system, coupled with a cluster of off-the-shelf machines should provide an alternative solution to more expensive and less scalable existing architectures.

1.2 Motivation

This research is motivated by the following factors:

- Not many DL systems provide support for scalability based on incoming requests. As users require specific content, the DL system should adjust automatically and be able to provide additional services which meet the user's requirements as it evolves.

- Many of the current systems have core components which rely on one another to complete requests and therefore have a one-to-one or many-to-one relationship. As a result, when a core service or component has gone down, all services relying on it cannot function. Generally, failovers exist but they again provide a single point of failure.

- With increased needs for a particular service, the administrator is required to make additional resources and services available. In a traditional DL environment these requirements may be difficult to satisfy.

- Since most DL systems are utilized by non-profit entities, methods for overcoming performance issues should be inexpensive, easily replicated and robust.

The above factors highlight the need for a DL system which is flexible, adaptable and low cost. It should reduce the maintenance load on the DL system owner and easily allow additional services to be added or introduced [1]. As user's interests vary, so should the system in order to maximize the resources available to provide the best possible service [2].

1.3 Aims

The aims of this research are stated below.

- To develop a flexible, reliable and scalable DL system. This system should meet users' needs without manual intervention.

— To provide evidence that this type of system handles increasing computational loads by utilising the migration and replication features effectively.
— To ensure that the need for a particular service is met by making more resources available.

1.4 Methodology

A component-based DL system was modified to create an experimental test bed. This included introducing additional components to track and monitor the performance of the modified components. The modified and additional components were then tested to ensure that they worked correctly with one another.

This experimental system along with the original one was distributed across a cluster of machines. These two systems were then tested using various component layouts and load conditions. Additional testing was done on the experimental system to determine the impact of the modifications.

1.5 Dissertation Organisation

Chapter 1 – introduces the dissertation contents, including the research motivation, its aims and the methodology used. It discusses the problems with some of the current DL systems and highlights need for a low cost solution to DL scalability.

Chapter 2 – documents the background to this research. It discusses other technologies that are related to the research topic. This includes a review of current DL systems, mobile agents, cluster and grid technologies as well as routing and registry services.

Chapter 3 – is composed of the design of the research system. This includes the design decisions made, the design issues that needed to be overcome in order to evolve the original system and the final experimental systems design.

Chapter 4 – presents the evaluation methodology including the results obtained. It details the tests performed and documents the differences in performance between the two systems.

Chapter 5 – concludes the dissertation and documents possible future research. This section covers further improvements which can be made to the experimental system.

Chapter 2

2. Background

This section documents the development of Digital Libraries, including the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH). Its concentrates on how these systems have evolved in order to provide better services and availability to their users. This allows us to get a good coverage of the current systems available as well as the services they provide and their design architectures. This is followed by a brief discussion of cluster and grid technologies as they potentially provide a cheap and efficient solution to managing large amounts of data which are dealt with in digital libraries. Mobile agents, which are similar to mobile or dynamic services, are discussed to highlight the benefits associated with them. Lastly, this section is concluded with a brief overview of registry and routing systems which are used to locate or track resources or services.

2.1. Digital Libraries

Digital libraries (DLs) have been given various definitions. In order to clarify our understanding of what a DL is the following definition has been selected:

> "A collection of digital objects, including text, video, and audio, along with methods for access and retrieval, and for selection, organization and maintenance of the collection" [1]

Initially the information stored in DLs was dependant on the group of individuals depositing the digital objects. Thus a DL system used by one group of people differed from another one that was managed and maintained by another. These DL systems would use different interfaces, different metadata formats and different interoperability protocols and posed two major problems. The first is that it was becoming harder for users to discover resources and the second is that there was no machine-based method of sharing metadata [4]. In order to overcome these problems the OAI-PMH was developed (and is discussed in more detail in the following subsection). This enabled different DL systems to exchange metadata on the content they contained, thereby enabling all users of different DLs to discover the resources available. Since then focus has shifted to the services provided by the DL systems as well as the different types of systems available. Numerous DL systems have recently become available. With the increase in digital content, performance of these systems has become one of the focal points of research. The following sections document some DL systems and discuss how they work as well as methods they use to maintain system performance.

2.1.1 Open Archives Initiative (OAI)

The OAI promotes and develops interoperability solutions in order to facilitate dissemination of content [5]. Their OAI-PMH provides a low-cost mechanism for repositories to share information or metadata records with one another.

The OAI framework identifies two main classes of participant. These are Data Providers and Service Providers. Data Providers are responsible for the deposit and publishing of digital objects into the repository. They also expose the metadata of the digital objects inside the repository. This metadata is encoded in XML using the unqualified Dublin Core (DC) metadata element set [5]. Service Providers or "harvesters", as they are more commonly known, harvest metadata from Data Providers. This metadata is then used to provide value-added services such as searching, browsing and peer-reviewing. It is important to note that one organization may offer both the data for harvesting and the end-user services. Data and Service Providers make use of the OAI-PMH in order to interact and exchange information with one another.

The OAI-PMH is based on the HyperText Transfer Protocol (HTTP) and uses the GET and POST mechanisms. OAI-PMH requests have a fixed structure which consists of a *base-url* and *keywords arguments*. The *base-url* includes the Internet host and the port of the server acting as the repository and the *keyword arguments* consists of a list of key-value pairs. The responses to all OAI-PMH requests are encoded in XML and conform to a specific XML schema. This allows responses and requests to be machine verifiable. The following table lists the six verbs used to facilitate the sharing of metadata between different repositories.

Verb	Description
GetRecord	Used to retrieve a single record from an item in the repository. This requires a key or identifier of the record as well as the format of the metadata included in the record.
Identify	Used to retrieve information about the repository. The response must contain a human readable name for the repository, a base URL and the email address of the administrator of the repository. The OAI-PMH protocol version supported by the repository is also required.
ListIdentifiers	Retrieves a list of record identifiers that can be harvested from the repository.
ListMetadataFormats	Used to determine the metadata formats supported by the repository.
ListRecords	Allows the retrieval of records and is used when harvesting from a repository.
ListSets	Used to retrieve the set structure of the repository.

Table 1: Verbs used in the OAI-PMH

2.1.2 Open Digital Library (ODL)

Component-based systems have proven to be more extensible and maintainable than the traditional monolithic systems. Therefore, digital libraries have also been componentised for the same benefits.

The Open Digital Library suite consists solely of components, each one performing a unique task or service. An extended version of the OAI-PMH (XPMH) is used by the components to communicate with one another [6]. While certain components are completely independent, others rely on additional components, for example, to store data. Users can download specific components in order to customise their digital library to provide specific services. Using the ODL architecture, different components can run on different machines and still work together as a single unit.

Each component of the digital library is essentially a Web-service and the reference implementations are written in Perl. This scripting language allows components to be platform independent and thus portable to all machines with a Perl interpreter. Components can be queried directly, using the verbs in the XPMH, or indirectly if an internal interface is provided. Individual ODL protocols for components are also specified as each component supports specific functionality [7].

In a typical scenario, such as searching, the user accesses a search interface in a Web browser. The user then types in his or her search request and waits for the response. The user's request is then sent to the search component. The search component checks its indices for any matching items. Metadata for the matching items is then requested from the repository component and forwarded to the interface, via the search component, for the user to view.

The following table contains a list of some ODL components available.

Component Name	Functionality	Interface Protocol
DBUnion	Merges together metadata from different sources.	ODL-Union
IRDB	Search Engine.	ODL-Search
DBBrowse	Facilitates browsing through metadata based on values of particular metadata fields.	ODL-Browse
WhatsNew	To track and obtain a sample of recent entries.	ODL-Recent
Box	Dumb archive supporting submit and retrieve functionality.	ODL-Submit
Thread	Engine for discussion forums, guest books and resource annotation.	ODL-Annotate
Suggest	System to make suggestions based on collaborative filtering.	ODL-Recommend
DBRate	Manages the submission and access to ratings of individual resources.	ODL-Rate

DBReview	Peer review workflow manager.	ODL-Review

Table 2: List of ODL Components, descriptions and protocols [6]

The ODL architecture provides a platform that may be extended for dynamic services given its distributed design.

2.1.3 OpenDLib

OpenDLib consists of an open network of federated services or components [8]. The overall functionality of the system is broken into a set of well-defined interacting services with functions for the coordination of tasks. Services may be centralized, replicated or distributed and thus the flow of communication between services is dynamic and dependant on service instances and load conditions. Communication between the various services is done using the OpenDLib protocol (OLP) and this is an open protocol in which requests are embedded in HTTP requests. All structured requests and responses are XML-based.

OpenDLib has classified their services into three classes, namely architectural infrastructure (AI), basic utility (BU) and application (AP) services.

AI services are responsible for the management of the DL and cater for the evolutionary aspects of the DL. These include supporting services like controllers, registries, security, etc. These services assume a communication protocol but it is not clear whether this is the OLP or not. Any other service that wants to use an AI service must implement the OLP in order to communicate with it.

BU services are basic DL support services which are used by AI services and possibly AP services. There are two of these services in the current release. The User Registry, which maintains information about users and interest groups, is the first. The second service is the Collection Service which provides a virtual view of the organization of the DL content space. An Accounting Service as well as a Rights Manager Service are to be included in future releases.

AP services form the backbone of the DL and implement the conventional services. The list of AP services can be seen in the table below.

Service	Description
Repository	Stores and disseminates documents that conform to the DoMDL model. Can represent structured, multilingual and multimedia documents.
Multimedia Storage	Supports the storage, real-time streaming and download delivery of stored video manifestations of a document. Includes disseminations of documents as shots and frames.
Library Management	Allows the submission, withdrawal and replacement of documents. Supports multiple metadata formats.
Index	Accepts queries and returns documents matching the queries.

Query Mediator	Dispatches queries to appropriate Index Service instances.
Browse	Supports the construction of indices for browsing as well as the actual browsing of the generated indices.
User Interface	Allows for human interaction with the application services and their protocols.

Table 3: Application Services of OpenDLib [8]

OpenDLib services periodically invoke protocol requests in order to become aware of the current state of instances. Based on the response, the service may use a particular instance or opt to use alternative ones available. As a result of this, services are more complex to develop. In order to alleviate this problem, publicly available Perl packages have been released by OpenDLib. This set includes packages to prepare service requests according to the OLP and packages that implement service-specific protocols.

2.1.4 DSpace

DSpace is an open source institutional repository system for research and educational material. It is freely available and was developed by MIT Libraries and Hewlett-Packard in a span of two years [9]. The system was developed primarily in Java and runs on Windows and Unix-like systems. It also makes use of other open source systems, namely:

- PostgresSQL,
- Jakarta Tomcat Java Servlet Container,
- and Apache HTTPD server.

DSpace has been designed to operate as a central service for an institution. Smaller groups within the institution, such as schools, labs, departments, etc., called communities, each have their own separate work areas within the system. Members of each community may deposit content into the system directly via the Web user interface. This content is reviewed before being included in the main repository.

Support for the OAI-PMH is available and the main services provided by the system are shown in the table below [10].

Service	Description
Searching	Allows the user to search the DSpace content using keywords. Searching can be done on the entire collection or on specific communities.
Browsing	Allows the user to browse the DSpace content by title, item issue date and authors.
Subscriptions	Users can subscribe to certain collections and are informed when new items are added to these collections.
Submission	Allows users to deposit content directly or in batch jobs.
Web Interface	Allows users to make use of DSpace via their Web browsers. All the above-mentioned services are accessible via this interface. Online help and community and collection home pages are reachable as well.

Table 4: Services provided by DSpace

DSpace was developed with the idea of a centrally located repository based on an open source toolkit. The focus has been on using and creating components, which are freely available, and thus the entire system can be installed for free [11]. Less focus has been placed on high scalability and dynamic services. A single point of failure also exists since DSpace is centrally managed and not a distributed system [10].

2.1.5 Greenstone Digital Library Software

Greenstone was developed by the New Zealand Digital Library (NZDL) project and provides full-text searching and browsing of indices based on different metadata types [3].

Indices for searching can be built from different parts of the document or metadata. Collections may have an index of full documents, an index of sections, paragraphs, titles, etc. Each of these can then be searched [1]. Browsing structures are derived from the metadata. Both the searching and browsing are built from instructions in the configuration file. This file controls the building and serving of the collection. The searching and browsing indices are generated automatically and can be rebuilt at regular intervals to stay up to date.

Documents and material can be updated without the system going offline. When material is added, documents are converted into the Greenstone Markup Language (GML) and include any metadata associated with the document. Each collection in the system has a separate directory and has five subdirectories. The *import* subdirectory contains the original raw materials or documents, GML files created from this are placed inside the *archives* subdirectory and the final collection as it is served to users is in the *index* subdirectory. A *building* subdirectory is used during the building process and any supporting files are stored in the *etc* subdirectory.

While Greenstone has not listed the file types supported by the system, they have made room for Plugins and Classifiers [3]. These are modules of code that can be slotted in to enhance the system's capabilities. Plugins are used to parse documents, extracting text and metadata to be indexed. Each plugin is required to specify three things, namely the file formats it can use, how they are parsed and whether the plugin is recursive or not. Classifiers work on GML and control how metadata is created into browsable structures. Classifiers are required to specify an initialisation routine, how they classify individual documents and the final browsable structure they generate.

Greenstone does provide access to distributed collections and version 3 of the software is component-based [12]. While performance gains are obtained by the new component-based software, no mention is made of service migration and replication.

2.1.6 The Flexible Extensible Digital Object and Repository Architecture (Fedora)

Fedora started out as a research project of Carl Lagoze and Sandy Payette. In 1999, the University of Virginia Library's research and development group discovered a paper about Fedora [13]. They developed a prototype and this provided strong evidence that the Fedora architecture could be the foundation for a practical, scalable digital library system.

In 2001, the University of Virginia Library and Cornell University collaborative development team received a grant that enabled the development of a sophisticated digital object repository system based on Fedora. Virginia and Cornell joined forces to build this robust implementation of the Fedora architecture with a full array of management utilities to support it. Fedora 1.0 was released in 2003, and in 2004, the Andrew W. Mellon Foundation awarded the project an additional $1.4 million grant to continue refining and building on Fedora's functionality.

Fedora is composed of two fundamental entities, namely: the digital object and the repository [14]. The Fedora digital object encapsulates both the content (i.e., data and metadata) and behaviors (i.e., services) associated with it. The repository is responsible for providing both access and management services for the digital objects.

The basic components of the fedora digital object are [15]:

- PID: a persistent, unique identifier for the object
- Object Properties: a set of descriptive properties which enable the repository to manage and track the object.
- Datastreams: the component in an object that represents MIME-typed content item. An object can have multiple Datastreams and can be either data or metadata. This content can either be stored internally in the Fedora repository, or stored remotely. Every object has one Dublin Core metadata datastream by default..

The fedora repository supplies a variety of features and can be seen below [16].

Service	Description	Status
Repository Service	A service that enables the creation, management, storage, access, and reuse of digital objects.	Yes
OAI Provider Service	A configurable OAI Provider service for harvesting metadata out of a Fedora repository via OAI-PMH.	Yes
Directory Ingest Service	A service to ingest a hierarchical directory of files into a Fedora repository.	Yes
Search Service	A configurable search service that can index any datastream or dissemination of Fedora digital objects.	Yes
Workflow and Orchestration Service	No description currently available.	Under Specification

Preservation Integrity Service	No description currently available.	Under Specification
Preservation Monitoring and Alerting Services	No description currently available.	Under Specification
Event Notification Service (Messaging)	No description currently available.	No
Persistent Identifier Resolution Service	No description currently available.	No
Object Reuse and Exchange	an interface to a Fedora repository to facilitate cross-repository interoperability	Under Specification

Table 5: List of current and future services for Fedora

Fedora has been shown to be flexible and scalable but requires specific knowledge in order to be installed and implemented correctly [17].

2.1.7 The National Science Digital Library (NSDL)

The NSDL is possibly the largest digital library ever constructed. It was created by the National Science Foundation to provide organized access to high quality resources and tools which support innovations in teaching and learning at all levels of science, technology, engineering, and mathematics education [18].

The NSDL Architecture consists of multiple components that interact with one another via web service interfaces and protocols. The following table shows the list of services currently supported by the NSDL [19].

Service	**Description**
NSDL Data Repository (NDR)	Represents resources as digital objects, and associates with them multiple metadata records from different sources. It represents the organizations and individuals that provide metadata or selected resources, and relates them to the appropriate metadata and resources.
Metadata Harvesting	Metadata, in both raw and normalized forms, can be provided for harvesting by NSDL via the Open Archives Initiative (OAI) protocol or the NDR API.
Search Service	Provides fundamental capabilities for locating resources and collections within the library.
Access Management Service	Access management provides a widely varying set of requirements for the users of the library; it also provides rights management, from the providers of intellectual property, i.e. the NSDL collections, and the providers of other services to the library. Anonymous user access is allowed but some materials are restricted.
Main User Interface	Users of the NSDL access collections and services through portals.

Archive Service	A basic requirement of national libraries is stewardship of the materials assembled within those libraries. The archive services retrieve materials represented in the metadata repository from public sites and archives both the metadata and content for future retrieval.
iVia	iVia provides focused crawling, an automated mechanism for including resources in the NSDL collection.
User Help	Text-based user help is available to users through NSDL.org.

Table 6: Services currently supported by NSDL

There are also plans to create two additional services. These are the Expert Voices service and the On Ramp service. These are focused on broader education and communication goals.

The NSDL does provide multiple services at various end-points. They have noted some problems with this though [20]. First, NSDL does not in promise 'seamless' interfaces as users move from one service to another; there may be variations in the look and feel of individual services. Second, because NSDL does not maintain notions about multiple copies of resources, it cannot guide end-users toward the most appropriate or recent copy of a resource, based on such factors as end-user access rights, cost, institutional preferences or fastest delivery.

While the NSDL architecture does allow for multiple services, there have been comments on data reliability and scalability [21].

2.1.8 Digital Repository Infrastructure Vision for European Research (DRIVER)

Driver is a multinational initiative which is co-founded by the European Commission. It aims to make all research publications openly available via institutional repositories. The current DRIVER network consists of digital repositories containing research and other scholarly articles across ten European countries [22][23].

In June 2008, the DRIVER consortium release version 1.0 of the DRIVER NETWORK EVOLUTION TOOLKIT (D-NET) under the Open Source Apache License. This proved to be a successful basis for a distributed service-orientated architecture. It enabled enhanced interoperability, collection building, provides tools for repository managers and functionality such as search and recommendation [24].

The D-NET toolkit allows services to be deployed dynamically, shared and combined to form new applications. A D-NET installation is run and maintained by a responsible organisation (RO). It is used by various participating organizations (POs), it is

responsible for admitting new POs and supporting them in construction and configuration of their applications. The D-NET infrastructure can be seen as 3 main areas and are listed below [25].

Area	Description
Data Management	Provides the services for harvesting, cleaning, storing, transforming, indexing and searching metadata records from OAI-PMH Repositories. They have been designed in a generic way so that POs can configure them to match specific requirements.
End User Functionality	Provides the services to form applications. These applications include portals, record collection management, recommendation systems, etc.
Enabling Area	Provides the services associated with the run-time of the D-NET installation. These include service registration, discovery, organization, authentication and authorization as well as subscription and notification.

Table 7: D-NET services listed by specific area [26]

DRIVER has successfully shown that integration and interoperability of multiple digital repositories. It has also highlighted the cost benefits of linking multiple repositories instead of using the single monolithic approach [25]. The project does not however deal with the performance of individual repositories as they would still be the responsibility of the RO.

2.1.9 A Digital Library Infrastructure on Grid Enabled Technology (DILIGENT)

DILIGENT is a digital library system which integrates Grid and DL technologies. It was a European Union – Information Society Technologies funded project aimed at allowing various communities to build up specialised digital libraries on demand in order to be utilized. It allows users to access shared knowledge in a secure, consistent and cost effective way [27].

DILIGENT makes use of Distributed Information Retrieval (DIR) strategies in order to service user requests and it has a Service Orientated Architecture. This allows the creation of independent services which can work together regard less of deployment location. It acts as a service which manages the DL requests and the available resources. Virtual organisations (VOs) are used to group together users and resources of a DL in a specific way. This ensures that there is a clear definition of what is shared and by whom for a given period of time. As a result, they can be created dynamically to satisfy transient specific needs by allocating and providing resources as required [28][29].

Users of the DILIGENT system can be classified as consumers, providers or both. Providers make resources available and consumers build their own DLs using the

available resources. This 'new' content, created by the consumer, can thus be shared again and made available to other consumers.

The final version DILIGENT was released as gCube and is being developed and maintained by the gCube Technical Committee [30]. It has been utilized successfully in marine biology, environmental monitoring and aquaculture.

gCube services are broken into 2 main categories and can be seen below:

Service Group	Service	Description
Information Management	Content Management	Monitors changes that occur on the stored information.
	Metadata Management	Supports the management of metadata objects and metadata collections.
	Annotation Management	Used to append a written explanation, illustration or comment about the information object.
	Content Import	Allows the importing of external resources including the description and their logical resources.
	Data Transformation	Fulfils the role of converting data (in XML form) from one input schema to another.
	Data Analysis	Performs data processing and mining in order to provide data in a common format for use by other services.
Information Retrieval	Search Framework	Provides all the services included in performing the execution of the search.
	Index Management	Provides services which manage the creation and maintenance of indices on the digital content.
	Distributed Information Retrieval	Used to retrieve information across various resources as well as being responsible for content ranking, selection and merging of relevant content.

Table 8: Services in the gCube system including their function

While gCube has been proven to work well and provide users with on demand research environments, it relies heavily on the Grid architecture and information sharing.

2.1.10 Summary

This section provides an overview of some of the DL systems available, as well as the services they provide. These range from centrally managed to distributed component-based systems. Many of the systems discussed are moving away from the centrally

managed design and aiming towards a more distributed and flexible system. The distributed systems provide the potential to perform better than the centrally managed ones since they can run across multiple machines. These systems may run on machines geographically far apart or on a group of machines locally and provides a good base for scalability and reliability. The use of multiple machines to provide enhanced performance is common in the cluster and grid computing community. A cluster or grid provides a cheap and easy way to obtain more processing power and is thus used in this work to improve scalability.

2.2 Grid Computing

The term *Grid* is used to describe a group of resources distributed over wide-area networks (WAN) that can support large-scale distributed applications [31]. These WANs typically rely on high speed networks in order to link multiple resources to one another.

Conceptually, the grid allows the networked distributed computing resources to be seen as a single system that shares these resources, and enables useful applications to be implemented. The various benefits achieved through grid computing are [32]:

- exploiting underutilized resources;
- providing parallel CPU capacity;
- supporting collaboration and the establishment of virtual organizations;
- enabling access to additional resources;
- providing (dynamic) resource balancing;
- enhancing reliability; and
- supporting the management of distributed IT infrastructures.

While grid computing provides a very neat and easy solution to large, complex and resource intensive applications, its reliance on high speed networks is a major problem in developing countries where network resources and internet access are expensive, minimal or none-existent.

2.3 Cluster Computing

Cluster computing is best characterized as combining a number of off-the-shelf computers and resources, using both hardware and software, in order to work as a single machine [33].

Over the past 10 years cluster computing has grown dramatically and there are a wide variety of software and hardware choices when creating a cluster today. Currently clusters are classified according to their functionality [34]. These are:

- Failover clusters
  This is the most widely used type of cluster today. Emphasis is placed on achieving high availability with minimum or no downtime if possible.

- Scalable high performance clusters
  These clusters are referred to as parallel or high performance computing clusters. They provide scalability, high-performance and high availability.
- Application clusters
  Application clusters provide high availability and scalability. Each node runs an instance of the application server and thus clients can connect to any of the server instances.
- Network Load balancing clusters
  This type of cluster distributes incoming requests among the multiple nodes it contains. Every node can handle requests for the same content or application. These clusters provide high scalability.
- Global Clusters
  This cluster is formed when two or more clusters, located remotely from one another, are connected to one another.

Cluster software can be categorized into cluster-aware and cluster-unaware applications. Cluster-aware applications have been built or designed specifically for use in a cluster environment. These applications know about other nodes in the system and may communicate with them if necessary. Cluster-unaware applications, on the other hand, do not know whether they are running on a cluster or single node. As a result of this, some additional software may be necessary to set up the cluster.

Most of the focus in cluster computing has moved away from the hardware issues and is currently more concerned with developing efficient and usable cluster software [34].

2.3.1 Cluster Computing Software

There is a wide variety of cluster software available for both management of the cluster as well has developing applications to run on them. This section will focus on four of them, namely: Parallel Virtual Machine (PVM), Message Passing Interface (MPI), Open Source Cluster Application Resources (OSCAR) and Openmosix. PVM and MPI are message passing systems which are commonly used on clusters. OSCAR and Openmosix, on the other hand, are software built for managing clusters.

PVM allows a heterogeneous group of computer systems, connected together via a network, to be viewed as a single parallel virtual machine [35]. It provides a uniform framework in which parallel programs can be developed using existing hardware and is independent of any particular language [35]. All message routing, data conversion and task scheduling across the different platforms are done transparently [36]. Users write their applications as collections of cooperating tasks and these tasks access PVM resources through standard interface routines. These routines allow initiation and termination of tasks as well as communication and synchronization between the tasks. At any point in the execution of a concurrent application, any task may start or stop other tasks as well as add or remove computers from the virtual machine. PVM is composed of two parts. The first is a daemon, called *pvmd3*, which must exist on all of the computers that make up the virtual machine. The second part is a library of PVM interface routines.

MPI is a standard for expressing parallelism through message passing [37]. MPI is portable and like PVM can work on heterogeneous machines [38]. MPI allows the user to run M processes on N processors. The number of processes may be less than, greater than or equal to the number of processes. Performance is influenced by the ratio of processes to processors [39]. MPI consists of library routines, a header file and a runtime environment. The runtime environment and the library routines are required to execute an MPI program. The header file consists of MPI-specific definitions and function prototypes, as well as definitions for macros, special constants and data types used by MPI [40].

OSCAR is a software system designed for building, installing and maintaining a Linux cluster [41]. OSCAR clusters consist of Servers, a gateway, nodes and a network. The servers are computers providing the cluster services. All the computers doing the actual processing of requests are called nodes. The gateway connects the external network to the internal one of the cluster. All requests must pass through the gateway for security purposes. The software consists of a Linux utility tool for installing Linux over a network, OpenSSH to handle security and a cluster management package called C3. PVM and an extended version of MPI are included for development of software that runs on the cluster. The final software component, the Portable Batch System, provides load balancing and ensures that the cluster works efficiently [42].

Openmosix, a UNIX-like kernel extension for single-system image clustering is freely available [43]. Openmosix consists of two parts: A Pre-emptive Process Migration (PPM) system and a set of algorithms for adaptive resource sharing. Both are implemented at a kernel level and are completely transparent at an application level. The PPM is the main tool for resource management and can migrate any process, at any time, to any available node. The resource managers are the load balancing and memory ushering algorithms [44]. The load balancing algorithm is decentralized and ensures that the load is spread equally across the nodes. Memory ushering ensures that nodes have enough free memory for processes. The system migrates processes if a node starts paging excessively, due to lack of free memory. While Openmosix provides built-in migration, memory management and load balancing, its drawback is that it is limited to operating systems using UNIX-like kernels.

Applications developed specifically for use on a cluster are required to be highly configurable. Minimal or no user interaction is required during task scheduling, message passing, migration and replication. These should occur automatically and somewhat autonomously based on a pre-defined configuration. Thus the use of the entire cluster remains hidden to the user. A similarity can be seen in artificial intelligent (AI) systems in which they act and react to the environment based on external information and an initial configuration. Migrating and replicating processes or objects have much in common with mobile agents which also move around from one machine to another.

2.4 Artificial Intelligence – Agents

Agents have been around since the late 1980's and have a great impact on our lives today. An agent can be defined differently depending on the perspective one is looking from. For simplicity and coherence, the following definition of an agent is used:

*"An **autonomous agent** is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future"* [45].

While there are a variety of agents in use today, the most commonly used is intelligent agents. These agents assist users, act on their behalf and learn in order to improve their performance [46]. A variety of these intelligent agents can be found online [47] and provide services like:

- Searching,
- News collecting,
- Shopping,
- Knowledge management,
- Personal assistants, etc.

These tasks are completed without the user's interaction apart from setting their preferences or making any adjustments to the way the agent is currently working. Some agents remain on the user's machine while others function from the Web. There are also those agents that move around and these are discussed in the following subsection.

2.4.1 Mobile Agents

Mobile Agents are software systems that are capable of moving from one machine to another. They provide numerous advantages that are listed in the table below [48].

Advantage	Benefits
Reduce network load	When large volumes of data, stored on remote hosts, need to be processed the agent can move to the location instead of transferring the data over the network.
Overcome network latency	In real-time systems, processes need to respond to changes in their environment immediately. Mobile agents can be dispatched from the central controller and execute locally and thus overcome network latencies.
Encapsulate protocols	In distributed environments various hosts may implement protocols to manage incoming and outgoing data. Protocols change based on new requirements. This need for change has become a problem in some cases. Mobile agents, on the other hand, can move to remote hosts to establish "channels" based on proprietary protocols.
Execute asynchronously and	In systems where network or Internet connections are limited, the agent can be dispatched and become an independent process. This

autonomously	agent can operate autonomously and be collected by the system when it is connected again.
Adapt dynamically	Mobile agents can sense their environment and thus act and make optimal decisions as the need arises.
Naturally heterogeneous	Mobile agents are computer and transport layer independent and thus provide optimal conditions for heterogeneous system integration.
Robust and fault-tolerant	Mobile agents' ability to react dynamically in critical situations makes it easier to build robust and fault-tolerant systems.

Table 9: Advantages of Mobile Agents

While the flexibility and usefulness of agents are clearly of use there are various security risks that exist with mobile agents. These arise from the agent as they have the ability to act on behalf of person, are not necessarily from a trusted source or location and can move from one system to another without user intervention [49]. These special traits make traditional security methods and assumptions invalid as agents do not adhere to them [50].

As a result of the potential security risks most of the focus has shifted to creating secure agents.

2.5 Registry and Routing Services

Registry and routing systems provide methods to track or locate resources. In most systems, resources are being created and removed at all times. The system proposed in this dissertation will utilize resource migration and replication and thus some sort of registry system is required.

2.5.1 Domain Name System (DNS)

The main function of the DNS is to provide resources (e.g., Web Pages, email addresses) with human readable names so that they can be usable in different hosts, networks, protocols, Internets and administrative organizations [51]. To get an understanding of how the DNS works a basic knowledge of domain names is required.

A domain name is used when using the Internet or sending an email. The URL http://www.howstuffworks.com contains the domain name howstuffworks.com. The email address iknow@howstuffworks.com contains it as well. These URLs are easier to remember than their corresponding IP addresses. For example, when someone types http://www.howstuffworks.com into a Web browser, the computer they are working on retrieves the corresponding IP address (216.183.103.150) in order to locate and retrieve the Web page [52]. This conversion from the human-readable address to the IP address is provided by the DNS.

The DNS has three main components [53], namely:

- Domain Name Space and Resource Records
  These provide specifications for a tree structured name space as well as the data associated with those names. Each node and leaf of the tree names a specific set of information or resources and query operations can be used to extract specific information from a set.
- Name Servers
  Name Servers are server programs which hold information about the domain tree's structure and set information. These servers may cache structure or set information, but in general a particular name server has complete information about a subset of the domain space. It will also have a list of pointers to other name servers that can be used to retrieve information from any part of the domain tree. Name servers know the parts of the domain tree for which they have complete information and are said to be an Authority for these parts of the name space.
- Resolvers
  Resolvers extract information from name servers in response to client requests. They must be able to access at least one name server and answer requests directly or indirectly using referrals to other name servers.

Because of the immense size of the database to manage resources, the DNS is maintained in a distributed manner with local caching in order to improve performance. Information is also replicated on various name servers.

Putting all the above together, after typing in the URL into the Web browser, it is passed to the local resolver on the machine. The resolver then takes this domain name and queries a name server to retrieve the IP address associated with the domain name. If the name server has the address it will send it to the resolver from where it is returned to the browse, which may then retrieve the resource (webpage). If the name server does not have the IP address it will refer the resolver to another name server which has more information about the domain. This process may continue until a name server with the IP address is located or the domain name is considered invalid.

Dynamic DNS also exists in order to maintain IP addresses that change several times a day. This system involves the client sending the new IP address to its DNS server and the DNS server, in turn, updating the necessary information. The only major drawback with this method is that there exists latency between when you update your IP and when the change takes effect on the Web [54] . This latency ranges from minutes to several hours. Thus, dynamic services would only be accessible once the update is complete and this could leave the service unavailable for large periods of time.

The DNS system itself has been proven to work both efficiently and effectively [55]. Attacks on the system in order to take down the Internet have failed and experts have concluded that a successful one would take about eight to nine hours of constant bombardment of requests.

2.5.2 Persistent Uniform Resource Locators (PURLs)

PURLs provide a service which is aimed at preservation of URLs. With the Internet growing dramatically we have URLs changing constantly based on system updates [56]. These changes and updates often leave Internet users in limbo as they do not know the updated URL. The PURL system provides a solution to this problem and its implementation is discussed below.

A PURL is a URL which points to an intermediate resolution service instead of the actual address or URL of the webpage being requested. The resolution service associates an actual URL with each PURL. A client using a PURL would be send back the actual URL, from the resolution service, and would complete the request as normally done using the DNS. What follows is a diagram of a client request using the PURL system.

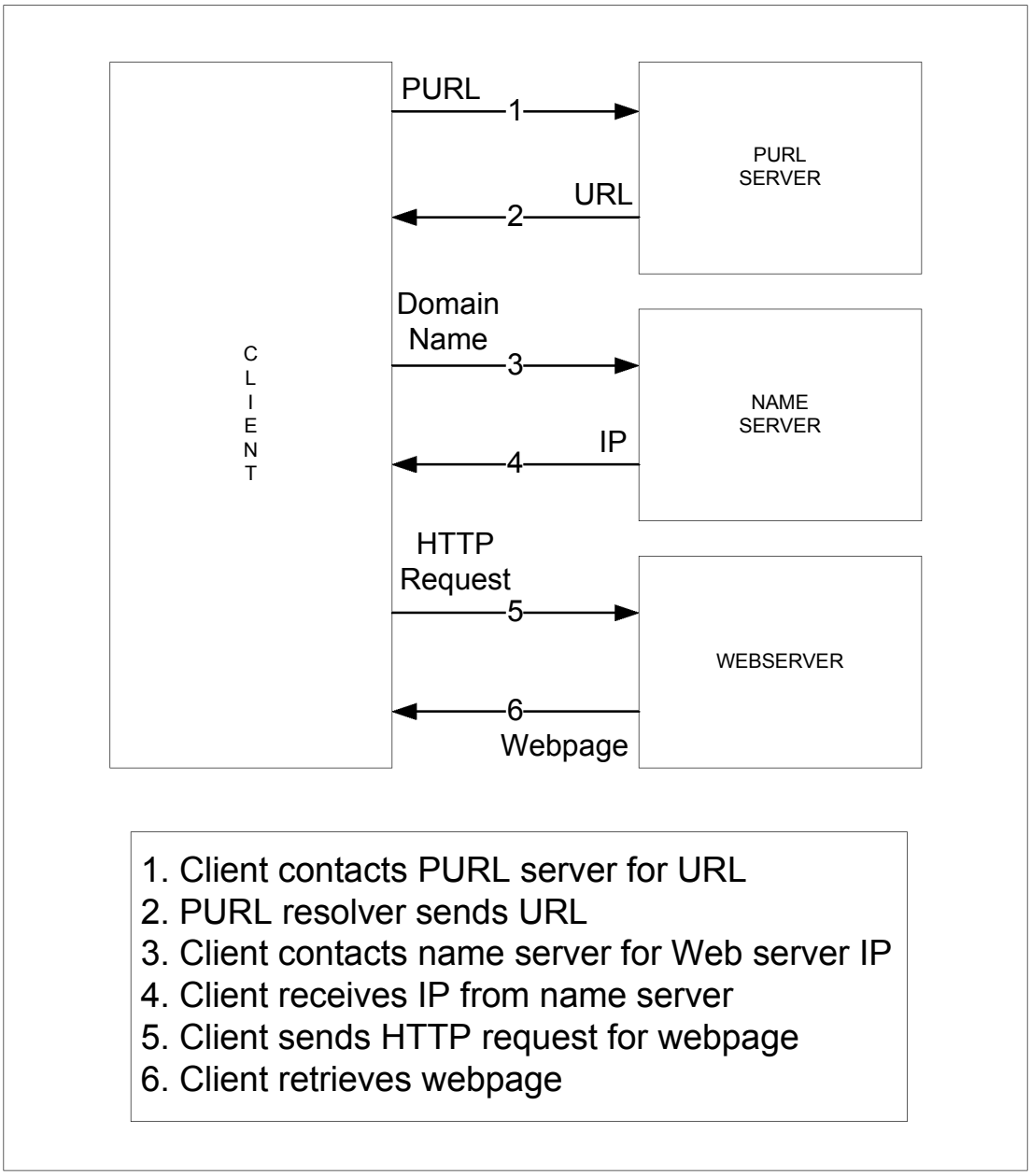

Figure 1: Client retrieving webpage using PURL

A PURL consists of three parts, namely: a protocol, a resolver address and a name. The IP of the resolver is obtained using the DNS. An example of a PURL would be:

http://purl.oclc.org/OCLC/PURL/FAQ

protocol resolver name

Using the PURL system, one can change what a PURL resolves to but not change the PURL itself. Thus PURLs can last longer than URLs provided that the associated endpoint of the PURL is maintained.

The PURL system provides a simple and elegant solution for mobile services. It ensures that the most recent URL is associated with a service despite the service having moved a number of times. The drawback with this system is that the PURL resolver becomes a single point of failure and a dedicated machine is required to provide this resolver service [57].

2.5.3 Universal Description, Discovery and Integration (UDDI) Protocol

UDDI protocol consists of a set of specifications which allow for Web Service description and discovery [58]. A UDDI registry contains programmatic descriptions of businesses and the services they provide. It also contains specifications or technical models (tModels) which describe how each Web Service works.

A registry is mainly populated with information from businesses and standards organizations. Descriptions of tModels, which describe specifications in a particular business or industry, are published on the registry by software companies, standards bodies and programmers. These tModels are then used by businesses to populate the registry with descriptions of the services they support. The registry assigns a unique identifier to each tModel and business registration and stores them in the registry. This data can then be queried by search engines, marketplaces and business applications in order to discover available services as well as to aid businesses interacting with one another over the Web.

UDDI offers multiple benefits at both design-time and run-time. These include [59]:

- code re-use,
- publishing information about Web Services and categorization rules specific to an organization,
- finding Web Services that meet certain criteria,
- determining the parameters, security and transport protocols supported by a given Web Service and
- providing a method to protect applications from failures and changes in invoked services.

2.6 Summary

This chapter has covered work related to this research project. It serves as a basis for this research and documents both advantages and disadvantages of similar systems and approaches used in the past. It started off with an introduction to Digital Libraries, including some DL systems currently available. It also highlighted the current trend of moving away from a single centrally managed architecture to a more distributed and

potentially scalable one. This was followed by a brief discussion on grid and cluster computing which noted the performance benefits that can be gained. The grid technologies reliance on high speed networks was noted as the barrier in developing countries. Mobile Agents were focused on in the AI section as they pertain to mobile services. This highlighted the benefits mobile services could provide as well as the potential risks which are introduced. Finally, this section concluded with a discussion of registry and routing systems.

Chapter 3

3. Design

3.1 Overview

The ODL system (discussed in the previous chapter) was used as a basis for this research. The shortcomings of this system, in terms of scalability, are highlighted in this chapter as well as the design decisions and approach used for the experimental system. This experimental system will introduce migration and replication, indirection as well as load balancing. These modifications to the ODL system should improve scalability and performance. In this chapter, the need for scalability, the additional components introduced to overcome this shortcoming as well the modifications made to the ODL components are highlighted.

As mentioned in the background section, the ODL system is composed of components which interact with one another in order to provide DL services. Components are templates of services and encompass all the behaviour and actions needed to be performed by a particular service. When a component is used for a DL system, an instance of the component is created and additional information is needed in order to ensure that the instance will work correctly. This additional information includes items such as the name of the component instance, the database and table names it uses, administrator email, etc. Without this information the component instance cannot function correctly.

A simple DL system is made up of a search and browse interface, a search and browse engine and an archive component. This DL can be seen in Figure 2 below. In this case, the system is spread across multiple machines but they may all run on one machine as well. Each component instance which is dependant on another is linked to it via a hard-coded URL. In this case, the search engine is linked to the search interface, browse engine is linked to the browse interface and both engines are linked to the archive component. During a browse request, the browse interface forwards the browse query to the browse engine. This engine then determines which items match the query and requests the associated data from the archive component. This data is then forwarded to the browse interface, via the browse engine and is displayed in a user-friendly format.

The hard-coded URLs force component instances to have a one-to-one dependency on one another. This one-to-one dependency introduces scalability problems. It forces instances to be static in terms of their location despite changes in the load or performance of the node they reside on. Additionally, an instance is dependant on a specific set of other instances whether there are others that can complete the request or not. Therefore, the use of migration and replication to increase availability or performance is not possible.

In the traditional system, allowing component instances to migrate and replicate would result in the hard-coded URLs being incorrect. This system would then be unable to function correctly if at all. This poses problems for the scalability of the system since the advantages of migration and replication cannot be utilized. The process of URLs being outdated is demonstrated in the Figures 2 and 3 below. In the first figure we see the system before the component instance has migrated.

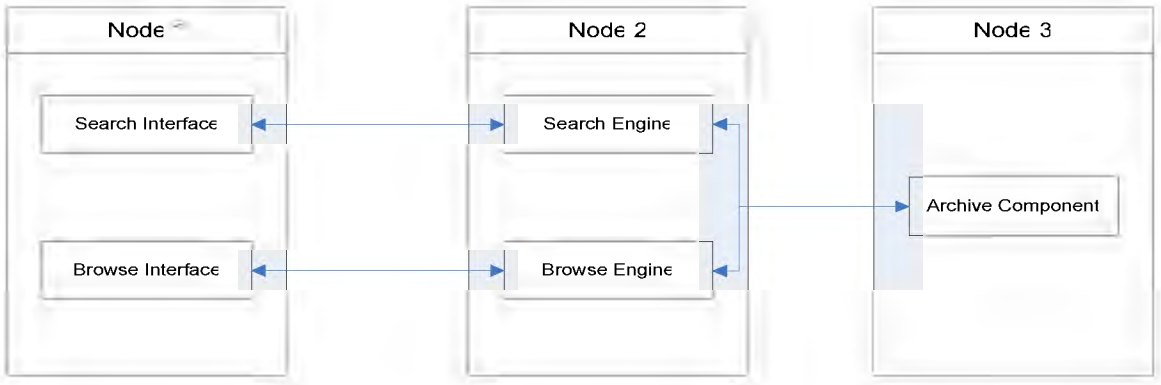

Figure 2: Traditional ODL system with hard-coded URLs

In the second figure (Figure 3), shown below, we see that the search interface has an incorrect URL to the search engine. This is due to that fact that the search engine has moved to another location. This outdated URL connection has been highlighted by using a thicker line.

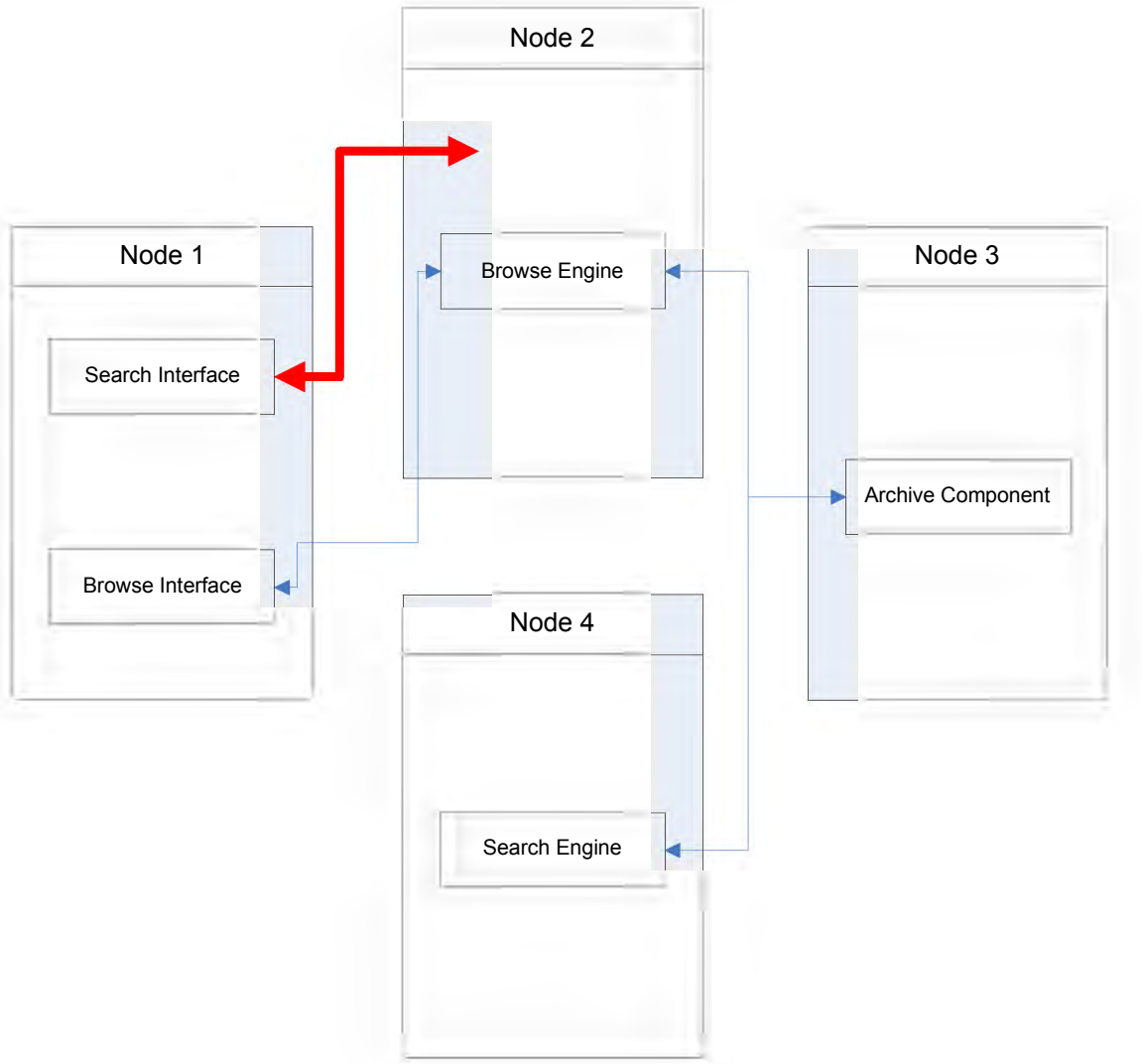

Figure 3: Traditional ODL system showing incorrect URL after instance migration

It is possible to have these URLs updated manually by the administrator but this becomes a daunting task as the number of component instances increases.

Another problem that arises is that the administrator is required to be available constantly in order to configure and reconfigure the system to meet the demands of user requests. This reconfiguration of the system depends on which component instances are being used most frequently. This makes the component highly dependant on the user requests on a continuous basis. For this reason, a dynamic and automated system is required to manage the reconfiguration of the system. By monitoring the requests and the tracking of components, a dynamic reconfiguration of the system can be achieved. Component instances that are frequently used may migrate and replicate to meet the needs of the user requests. The Registry in conjunction with the Load Monitor enables this reconfiguration.

As a result, two additional components and functionality had to be introduced to support scalability, to meet the needs of tracking migrating and replicating components. These are

the Resolver and Central Registry components. A basic load balancing system was also introduced to meet the demands of user requests and dynamic system reconfiguration. Each of these was written in Perl like the other ODL components.

The entire Registry and Resolver system can be seen in the figure below and is based on models such as the DNS and PURL system.

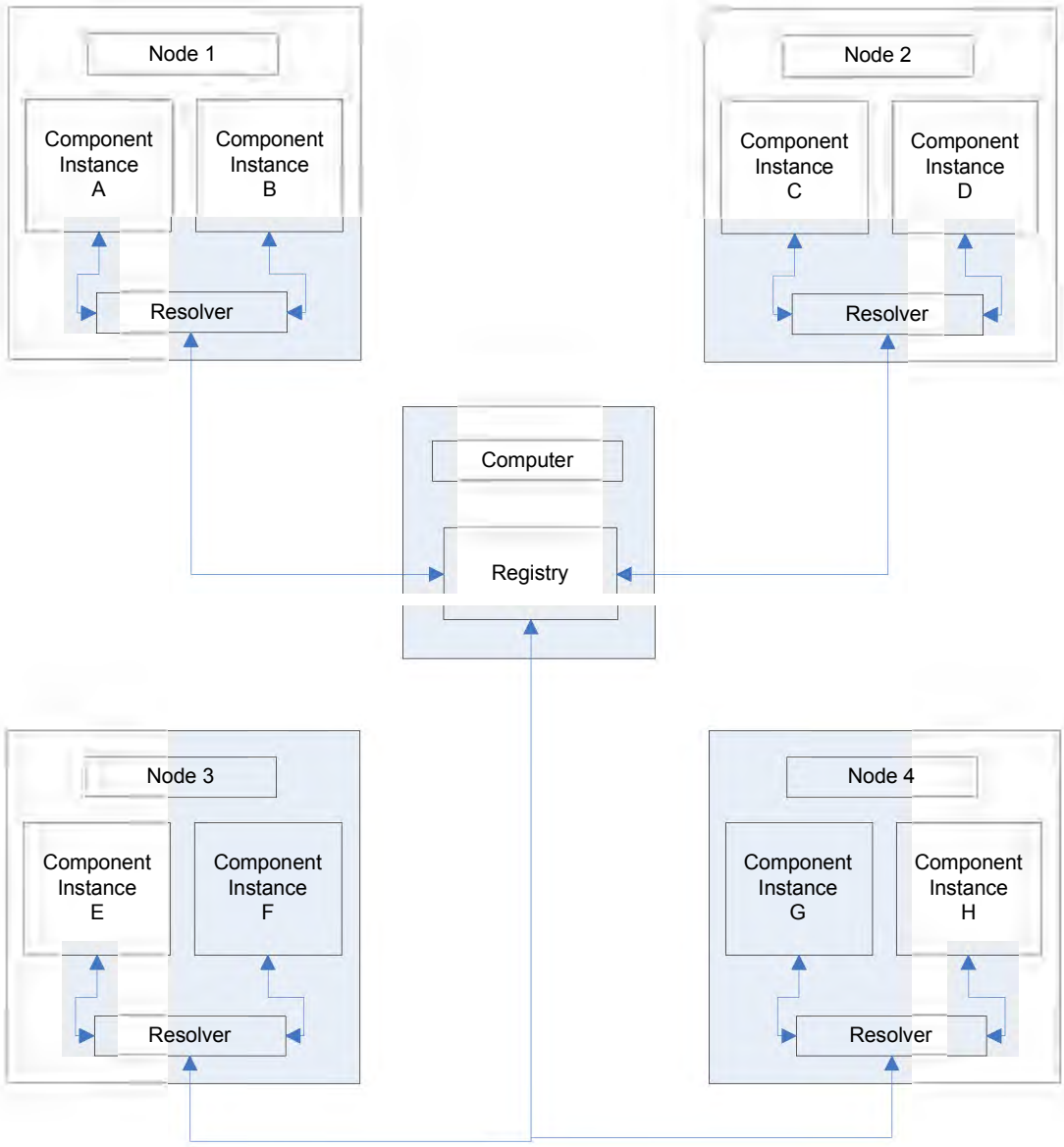

Figure 4: System with Registry and Resolver components

The Central Registry and Resolver components alleviate the issue of updating URLs to component instances. All components within the system are tracked and updates are done automatically by the system. The decisions on when the system should migrate or replicate instances and what needs to be migrated or replicated are the task of the load balancing system. This system is discussed later in this chapter.

3.2 Resolver

A Resolver resides on each node within the cluster. It is a Web service which is placed in the /home/user/public_html/cgi-bin/Resolver directory. This Resolver may work for multiple users. The main responsibility of this component is to keep track of the component instances on that node. This is done using a mySQL database, which is used to store information about the component instances on the node. The information associated with each component instance is as follows:

- Name of the instance
- Type of component
- Version of the component
- Owner of the instance
- Database it uses to store data
- URL to component instance

In essence, the Resolver can be seen as a local registry. It is also responsible for retrieving URLs to components from the Central Registry. The Resolver tracks the movement of component instances in and out of the node. This can happen in five different ways, as listed below:

Action	Resolver
Component migrates to node	Add
Component migrated from node	Remove
Component replicates to node	Add
New Component is created on node	Add
Component is deleted from node	Remove

Table 10: Possible changes to Resolver's list of components

The following scenario describes how the Resolver fits into the system:
A user sends a request to the search interface. This interface then contacts the local Resolver and requests a URL to a search engine. The Resolver then contacts the Registry for the appropriate URL and this is then forwarded to the search interface in order to complete the request. The process of a component instance retrieving a URL via the Resolver continues to occur until no more additional component instances are required to complete the request.

The following table documents the functionality provided by the Resolver as well as the processes that occur when it is invoked.

Function	Effect	Process
startup	Sends a list of all component instances residing on the machine to the Registry. The Registry then adds these component instances to the list of instances within the entire system.	Upon startup the Resolver executes this function in order to notify the Registry that the node is running. In this notification it sends the Registry a list of all component instances residing on the node. The Resolver reads the information from its database and sends the name, owner, type, version and URL for each component to the Registry. This information is encoded using XML.
shutdown	Sends a notification to the Registry that a node will be shutting down. All component instances residing on that node are removed from the system.	Upon shutting down, the Resolver executes this function. It signals the Registry that the node is going to be switched off and therefore all instances on the node are no longer accessible.
urlrequest	Component instances use this function to retrieve a URL to another component instance to complete the processing of a request.	A component instance contacts the Resolver for a URL to another component. The Resolver then checks its database for an appropriate URL. Once found, it is forwarded to the component instance to continue processing the query. If a URL cannot be found, the Resolver then forwards this request to the Registry. Once the URL has been obtained, from the Registry, it is forwarded to the component instance and the processing of the request may continue.
regcomp	Informs the Resolver and Registry that a new component instance is on the node.	When a component instance is added to the node, via migration, replication or creation, its information is added into the Resolver's database. For migrating and replicating components, this information is added automatically without user interaction. When a new instance is created, the user is required to supply the URL to the component.

remove	Removes a component instance from the Resolver and Registry.	A component instance may be removed from the node in different ways. This could be by migration or deletion. In each case this method is invoked and the instance is removed from the system. The function takes the URL as input and removes the associated instance from the Resolver's database. This then calls the Registry and the same operation is applied there.
compcount	Determines the number of component instances residing on a node.	This function is called by the Central Load Manager and is used to make load balancing decisions. The Resolver counts the number of component instances it has in its database and sends this value to the Load Manager in an XML document.
geturls	Retrieves a list of URLs to all the component instances on the node.	This function is used by the local Load Monitor on the node. After retrieving the list of instances, the local Load Monitor contacts each component to complete its task. This is discussed in more detail later in this chapter.
check	Determines whether a table is already being used by another component instance in the system.	This method is invoked by the migration component. Once a component instance is ready to be set up it invokes this method to determine whether or not the database table can be restored without overwriting any data.

Table 11: Functionality provided by the Resolver

Given the distributed nature of the system, a form of local caching was introduced. URL requests run through the Resolver and this provides an opportunity to use local information to complete the request. The Resolver first checks its database to determine whether a suitable component is located on the machine. If one is found, this URL is returned instead of going to the Registry. If one cannot be found, the Registry is then contacted to find a component and sends back its URL. This local caching minimizes communication between the component and the Registry and, as a result, it decreases request times.

3.3. Registry

There is one central Registry component running within the cluster. This component is placed in a fixed location, namely the /home/user/public_html/cgi-bin/Registry directory. This component runs on one machine only and is also independent of the users in the system. Its main responsibility is to keep track of all component instances within the

system and provide their associated URLs on demand. It stores this information in a mySQL database – it contains the following information for each instance:

- name of the component instance,
- type of component the instance,
- the actual URL to the instance,
- the version of the instance,
- the status of the instance ('Running' or 'Down' for migration and replication), and
- the owner of the instance.

The Registry allows for tracking of components within the cluster. Thus, it allows the component instances to be contacted despite their change in location over a period of time. While the registry does provide a single point of failure, a failover system should be in place. If registry goes down, the failover registry would come online. As this is not the focal point of the research, it will not be discussed here but is considered in the future work section.

The functionality provided by the Registry can be divided into three main parts. Each of these are discussed below including the functions used in them as well as how they fit into the system.

3.3.1 Node tracking

This part of the system includes the tracking of nodes as well as the component instances residing on them. When a node is started up it sends a list of all the components residing on it to the Central Registry. This comes in the form of an XML fragment and the data supplied is added to the Registry's database. Each component instance residing on the node has thus been added to the system. Conversely, when a node is shut down, all instances associated with that node are removed from the database. This tracks the nodes within the system but the component instances may still migrate and replicate.

The Registry is thus continuously keeping track of both component instances and nodes within the system.

3.3.2 URL service

This is the service which is most used in the system. It provides the URLs to other component instances within the system. Component instances contact the local Resolver in order to retrieve a URL to a component instance that they need to complete a request. The Resolver then contacts the Central Registry and invokes the requestURL or '*rq*' function. It also supplies the component type, version and owner information to the function. The associated URL is then extracted from the Registry and sent back to the Resolver and then to the component instance. This entire process can be seen in Figure 5 below.

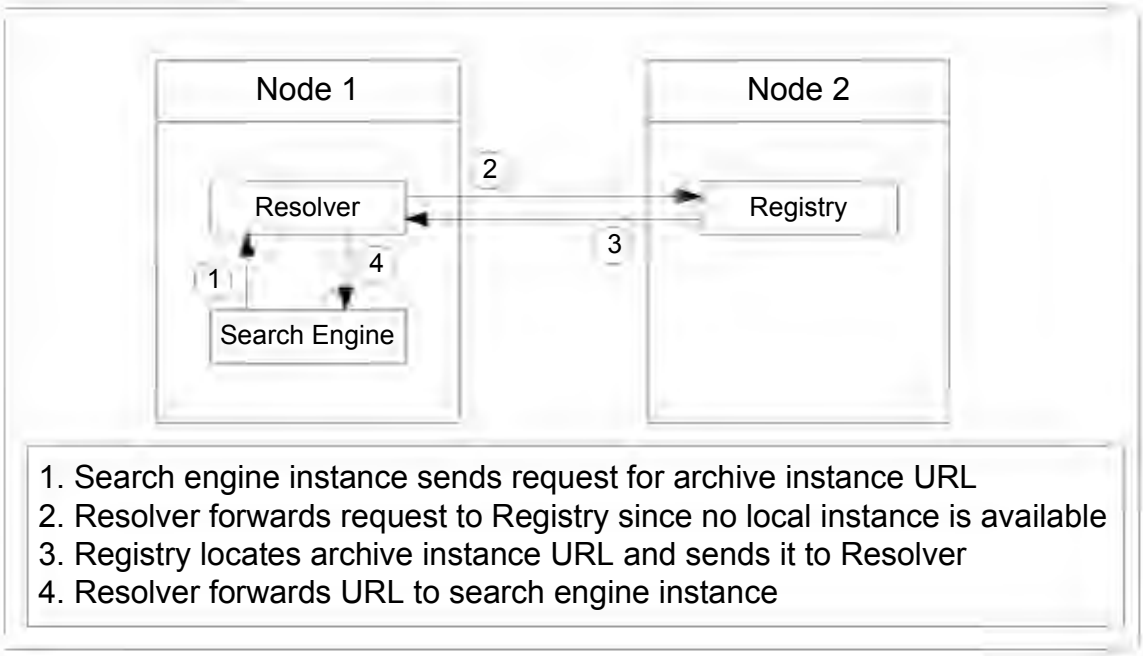

Figure 5: Process of retrieving an archive instance URL from Registry

Retrieval of URLs by the Registry occurs in two ways, excluding the possibility that no instance is found. When a URL is requested the Registry checks its database for all components matching the criteria. It then selects one of them based on a round robin algorithm in order to distribute the requests among all the instances.

In the second case, the only available instance may be migrating or replicating. As a result, this instance cannot process requests and will only be available once the migration or replication is complete. The Registry will then delay the request until such a time as the component instance is available (i.e., migration or replication is complete). The associated URL is then retrieved and the process continues as before.

If no component instance can be found the Registry sends back a notification to the Resolver saying that no instances are available. The request can therefore not be completed.

3.3.3 Migration and Replication Requests

This part of the functionality pertains to how the system replicates and migrates component instances. The Central Load Manager determines which machine is overloaded and whether migration or replication is needed. The request to allow migration or replication however is still made by the Registry. As illustrated in Figure 6, this request is sent, using the '*mgrequest*' function, by the instance to the Resolver. The Resolver then sets the instances' status to 'Down' and forwards the request to the Registry. The only factor considered, by the Registry, during this decision is whether or

not another component instance is migrating or replicating. If this is the case then the request is denied. If not then the migration or replication request is accepted. (Why multiple component migration was not chosen is discussed at the end of this section)

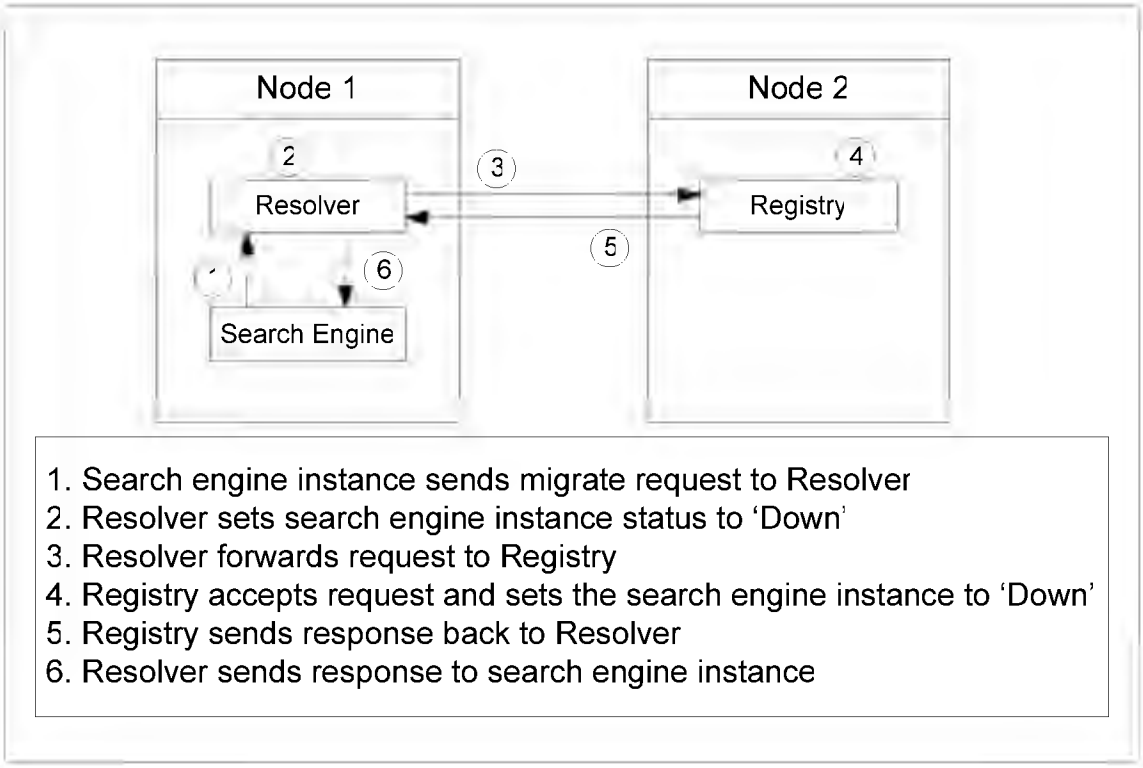

Figure 6: Steps involved in initiating a migration or replication request

Upon accepting the request, from the Resolver, the Central Registry will flag the status of this instance as 'Down' in its database. By setting this flag, the component instance is no longer available to process requests. Once the migration or replication is complete, the component instance information is updated. The URL for the associated component instance entry is updated and the status will be set to 'Running' again.

Allowing multiple components to migrate and replicate introduces additional complications as well as destabilizing the system. Problems which begin to arise include:

- which components are migrating and to which machines,
- how to determine the effects of multiple migrations to a single machine and
- ensuring that the same instance does not migrate again before the first migration is complete.

Having a single instance migrating or replicating allows the system to first stabilize before allowing another instance to do so. Additionally, the time to migrate (or replicate) is quite short and thus the time between migrations is short as well. While single instance migration (or replication) may not be the ideal it should serve adequately enough to show the benefits it presents. Having multiple migrations or replications may offer additional benefits but this falls beyond the scope of the project.

3.4 Load Balancer

In order to determine when component instances should migrate or replicate, some sort of system monitoring was required for the load on each machine as well as the load managed by each component instance. With this information the system would make an informed decision about which component instance to migrate, depending on its usage. A pair of simple Central Load Monitor and local Load Monitor components was introduced.

The Central Load Monitor is responsible for ensuring that the loads of all the nodes are similar while the local one would track the usage of component instances on a single node. This system can be seen visually in the diagram below (Figure 7).

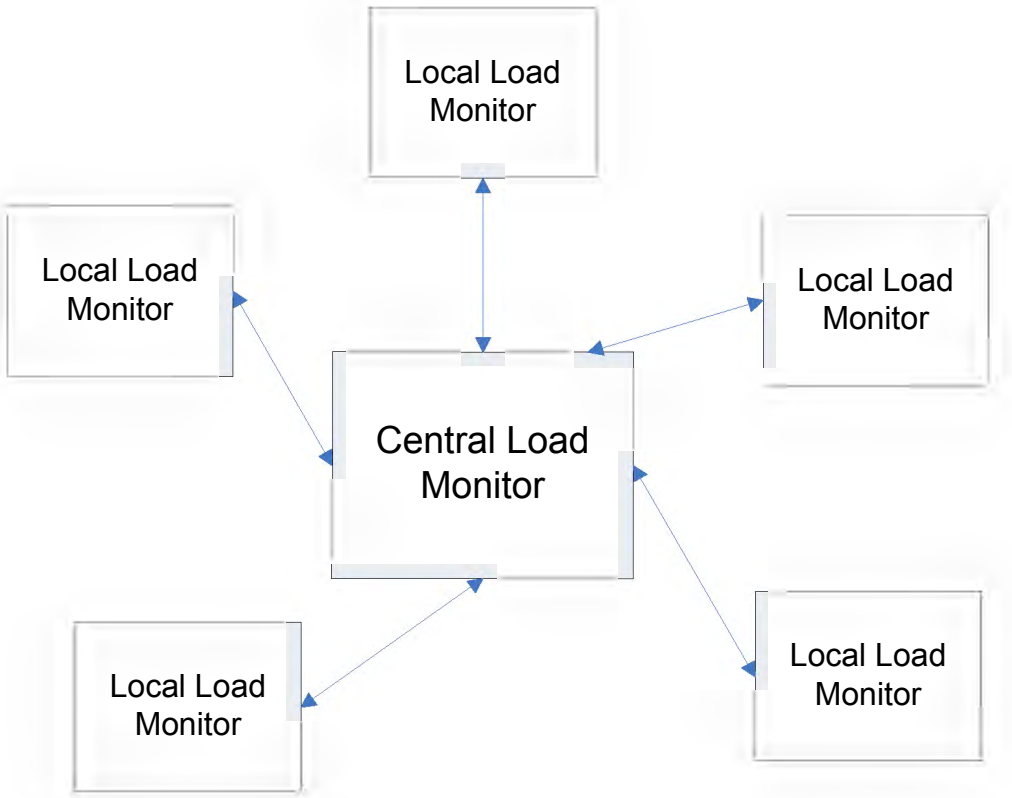

Figure 7: Central Load and Local Load Monitor overview

Having all the information recorded by the Central Load Monitor would be another option but this would increase network usage unnecessarily since the network load would increase as the number of instances increase. Thus, the approach of having the two components working together was chosen. Each of these components is discussed in the next section, starting with the Central Load Monitor.

3.4.1 Central Load Monitor

The Central Load Monitor resides in a fixed location within the cluster. It fetches the load from each node in the cluster on a regular interval. This is retrieved from the local Load Monitor. It uses this load information to determine whether all the nodes within the cluster have a similar load. This periodic fetching of loads provides the Central Load Monitor with a snapshot of the entire system's load for that period.

The Central Load Monitor makes use of simple statistics to determine when a node is overloaded. All the loads from the nodes in the cluster are collected and the average and standard deviation are determined. This calculation ignores nodes which do not have components on them as they do not form part of the system load. These nodes do not contribute to the request time as no components reside on them and therefore cannot be overloaded by the components within the system.

After this is calculated, the load on each node is compared to the average plus two standard deviations (as this would cover roughly 95% of the data in a standard bell curve). If a node's load is greater than this value it is deemed to be overloaded. While this is being calculated, the node with the least load is also determined. Once the node with the highest load has been determined the Central Load Monitor tells it that it is overloaded in relation to the rest of the system. The Local Load Monitor on the node is informed that it needs to migrate or replicate a component instance residing on it. The node to which a component instance should be sent to is the one with the least load, which may be a node with no component instances on it.

Once a component instance is in migration or replication, all load monitoring is stopped. This is done because the load on the overloaded machine will have a sharp increase while it is migrating or replicating a component instance. Once the migration or replication is completed, the Central Load Monitor waits for a period of 30 seconds for the system to stabilise before fetching the load values again. Which component instance to migrate is the decision of the local Load Monitor and this is discussed in the next section.

3.4.2 Local Load Monitor

Each node within the cluster has a Local Load Monitor which resides on it. This monitor sends the Central Load Monitor the load on demand and also determines whether to migrate or replicate a particular component instance.

When supplying the load information to the Central Load Monitor, it uses the system '*uptime*' command. This command returns the amount of time the node has been running as well as the load for the past minute, past 5 minutes and past 15 minutes. For the load analysis, only the load for the last 1 minute is used. This load is the load on the entire machine, whether components reside on it or not. This provides the Central Load Monitor with a snapshot of the load on the node for that interval.

Once a node has been determined to be overloaded, the local monitor is contacted to determine which component instance to migrate. The local monitor then requests the URLs to all the component instances residing on its node from the Resolver (using the geturls function). Once it has this list, it contacts each component instance for its average response time over the last period of load collection. The component instance with the greatest average time is chosen to be migrated or replicated.

The decision whether to migrate or replicate the component is based on the other components residing on the node. If no other components are found, the component instance is replicated to the node with the least load. This is done since the node may be overloaded with requests to this component instance and more of these instances are required. In all other cases, a component instance is migrated to the machine with the lightest load. The migration of a component instance is dependant on the load on the two nodes involved in the migration. The component instance is chosen based on whether it will minimize the load difference between the light node and the overloaded one. This process happens by simulating a migration of each component instance to the light node. This is done by taking the current load on the overloaded node and removing the contribution made by the component instance. The light node has its load increased by the amount the component instance would have contributed. The difference between the loads is then noted and the next component instance from the overloaded node is used in the simulation. After all the instances have been simulated, the instance which yielded the minimum difference in the two loads is migrated to the light node.

For the load monitoring and component tracking to be used correctly, modifications to the components had to be made. These are discussed in the following section as well as which components form part of the system.

3.5 Component Functionality

The components adapted for the experiments were the search and browse interfaces, the search and browse engines and the archive component. Some components have a common set of modifications made to them while others are only slightly modified. The modifications can be divided into three main categories and are discussed individually below.

3.5.1 URL Retrieval enhancements

This modification allows components to contact the Resolver to retrieve a URL for an additional component instance. This modification was made to the interface and engine components as they rely on other components when completing certain requests. The archive component does not need this enhancement as it does not rely on any other components to process its requests.

Each interface contacts the Resolver for a URL to its associated engine. The search interface would thus have the functionality to retrieve a search engine URL only. The same logic applies to the browse interface and the browse engine. In the case of the browse and search engines, they both rely on the archive component to complete certain requests. Thus both the engine components have been modified to retrieve an archive URL from the Resolver.

With this enhancement, the components do not have hard-coded URLs and will always be served with a valid URL to a particular component. It alleviates the need for the administrator to update these component instances manually. This also provides a platform for components to migrate and replicate.

3.5.2 Migration and Replication enhancements

Component migration and replication has been enabled for the search and browse engines as well as the archive components. Migration of interfaces, while possible, is not practical as the user would not know which URL to contact in order to use the system. A possible alternative would be to proxy the interface.

With this enhancement, a component instance can be requested to migrate or replicate to a specific location on its own. When a migrate or replicate request is received on the source node, the component instance copies itself to a fixed location, which in our case happens to be the '*/home/user/public_html/cgi-bin/MigratedComponents/workspace*' directory. This directory is then compressed into an archive file and encoded into an XML document using base 64 encoding. This document is then transferred to a migration service on the destination node using HTTP.

For migration, the encoded XML document is sent to the migration service on the destination node. Once the XML document has been received, the archive file is recreated and uncompressed, thereby re-creating the copy of the component instance. The component instance itself is then moved to the directory in which a template of the component exists. Any data dependencies are set up to use remote access. Once the instance has been set up and is working correctly, the original component instance is removed from the original node. The Registry and Resolver (on each node) will then update to node the change of the component location.

For replication of a component, the same steps are performed as was in the migration. The one exception is that the original component instance is not removed from the system. This leaves us with the original component on the source node and a copy on the destination node. Updates that occur to on Registry and Resolver (on the destination node) components to introduce the new component into the system.

The following diagrams (Figure 8, Figure 9 and Figure 10) illustrate the migration and replication of component instances.

Figure 8: Nodes before migration and replication

Figure 9: Nodes after component A has been migrated from Node 1 to Node 2. Component A is still using the database which resides on the original node.

Figure 10: Nodes after component A has been replicated from Node 1 to Node 2. Both components now use the dataset on the original node.

During the initial design phase, the component instance along with its data was migrated or replicated. This approach resulted in drastic time delays during migration or replication. The main reason for this delay was the copying of the data associated with the component instances. Remote access to the data was used to avoid data dependency issues, since service provision is the key element of this work.

3.5.3 Component monitoring enhancements

In order for migration and replication of component instances to work effectively, each component with migration and replication capability should be able to track its service times. Thus each component instance keeps a list of its request processing times in a file.

This file '*requestTimes*' is updated whenever the component services a request. It appends each request's time to the file. At the start of each cycle of load monitoring from the Central Load Monitor, this file is moved to the '*requestTimesOld*' file. This second file is used to determine the average request time of the component over the last load monitoring period. This average time is requested from the Local Load Monitor when the node is overloaded. The component reads the file, adds all the request times and calculates the average.

3.6 Summary

This section documented the modifications made to the ODL system as well as the introduction of additional components to support scalability. The introduction of Registry and Resolver components was discussed to show that components can be tracked in a dynamic environment of migrating and replicating components. A load balancing system was also discussed in order to demonstrate the effective use of migration and replication of components. All of this combined with the component modifications creates a potentially flexible and scalable DL system.

Chapter 4

4. Evaluation

4.1 Introduction

The previous sections documented the development of the experimental system as well as the background and related work. This section covers the evaluation of the system. It documents which tests were done, as well as what each test aimed to achieve. Different aspects of the system needed to be tested and compared to the traditional system in order to gauge the ability of the experimental one.

4.2 Expected Outcomes

Based on the design of the experimental system, we can make some assumptions about its performance. The expected performance of the experimental system (pink line) in terms of request times would be similar to that shown in Figure 11 below. This is shown in comparison to the expected performance of the traditional system (blue line). The graph below is an approximation and is not based on any testing results.

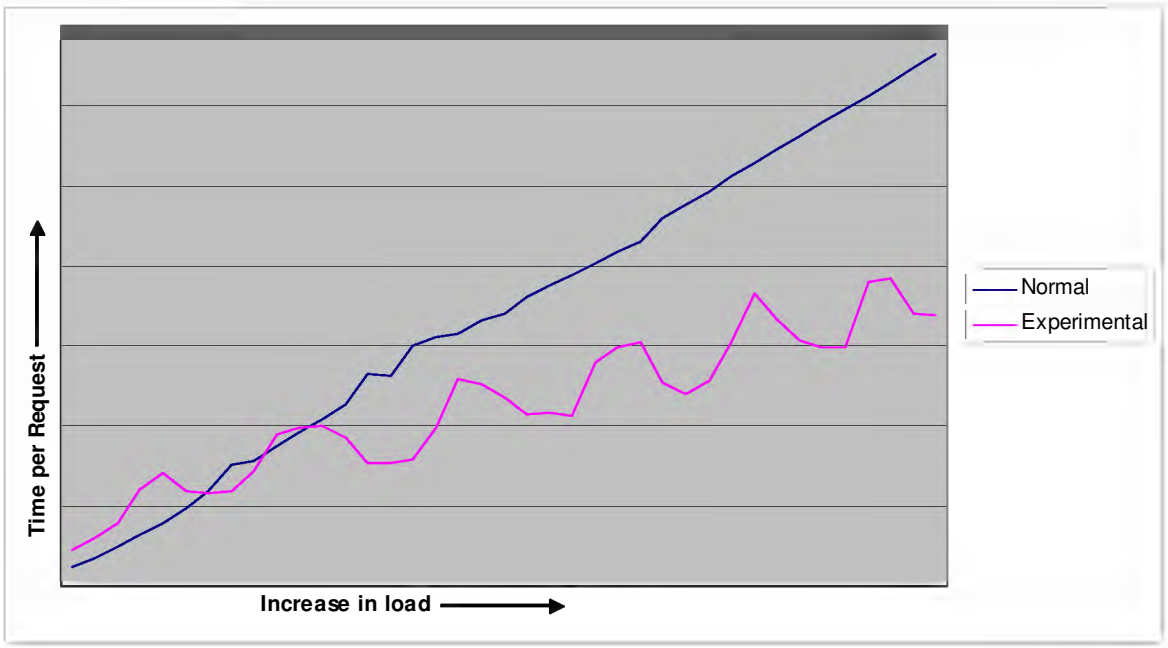

Figure 11: Graph of expected performance by both systems. The spikes for the experimental system would be due to migration and replication of components. This is followed by normal processing until the load increase to require migration or replication again.

In the above graph (Figure 11), we see the experimental line increasing then spiking and then dropping dramatically. This is expected because as the requests increase, so will the

load on the node. This increase on the node would cause the load balancer to migrate or replicate a component to another machine. This would result in the request time spiking until the migration or replication is completed. After the migration or replication, the system should stabilize and the request times decrease once again. This pattern will repeat if the load continues to increase as the system would be attempting to adjust to the optimal layout to service requests. In addition to this, we expect the experimental system to perform worse initially as it has the additional overheads of communicating with the Registry and Resolver components.

The request times of the traditional system are expected to continue increasing since the incoming requests are placed into a queue if they cannot be serviced due to too many requests. Initially, the traditional system should perform better but over time its service times will decrease and thus start performing worse than the experimental one.

The pattern of requests being received by the system will determine how each of them performs. The graph above (Figure 11) is expected when multiple requests are received at the same time by the system. In the case with requests being received serially, it is expected that the traditional system will perform better. Based on this, we know that the experimental system should perform better under certain conditions. To determine these conditions, a variety of tests have been conducted. These tests cover the performance of the system, the overheads introduced by additional components and correctness and flexibility.

4.3 Testing Procedure

The testing was conducted across 3 nodes inside a cluster. Each node within the cluster is a Pentium 4 3.0 GHz machine with 512 megabytes of RAM, an 80 gigabyte hard disk drive and runs on a 1 gigabit network. Only 3 nodes were used since all combinations of the component instances could be modelled on it. Additionally, limiting the number of available nodes would provide a good comparison to the traditional static system as it cannot use additional nodes dynamically.

One of the major drawbacks with the Perl programming language is that additional time is taken to start up the Perl interpreter. An interpreter is spawned for each incoming request that needs to be handled. To alleviate this problem, SpeedyCGI [61] was used. Instead of exiting the Perl interpreter after the script is run, it keeps it in memory to service other requests instead of starting up a new interpreter. This enabled the DL systems to simulate persistent component instances.

The search and browse components were selected for testing as they are used most often when querying information from a DL system. The requests for each test were timed and sent using a Perl script. Requests to the search interface was for the search term '*computer*' and the browse interface was required to display the records in descending order by time and date. These specific components were chosen as they are normally the first entry point when interacting with DL systems.

For timing the requests, the Perl Time::HiRes module was used. This module provides an interface to system calls for high resolution time and timers. Requests were sent using the EZHTTP module which is used by the ODL system. A simple test script can be seen in Appendix A. Simultaneous requests were obtained by replicating, by using the *fork* command, the process which is sending the requests.

All the tests were run using the test script and no manual testing was done. As the cluster was reserved for testing purposes, it was completely available for the testing procedure. A different test script was used depending on which test needed to be run and each test was run 3 times with the request times being noted. The mean or average request time for each test run was then determined and this was used as the final data. From here on, the term average will be used to denote the mean request values calculated for each test.

For the purpose of testing, an archive of 10200 records was used. These records were harvested from the Networked Digital Library of Theses and Dissertations (NDLTD) Union Catalog [60].

An overview of the tests conducted can be seen in the Table 12 below.

Test	**System**	**Description**
System Stability and Coherence Test	Experimental only	Ensures that the experimental system is working correctly and that the system was fully functional. This includes testing with and without the new components and functionality introduced in the experimental system.
System Performance Tests	Experimental and Traditional	These tests compared performance of the two systems. This included testing browsing and searching individually and simultaneously in order to determine the performance differences between the systems.
Load Pattern Testing	Experimental and Traditional	Tests in this section attempted different incoming request patterns. These included an increasing number of requests, random number of requests and a periodic request pattern. This test was mainly aimed at the experimental system as we could see its adjustment over time.
Load Monitor parameters	Experimental only	Tested the performance of the experimental system with different load monitoring intervals in order to ascertain the most effective values

		for each of the patterns mentioned.
Overheads Isolation Testing	Experimental only	This test was used to determine the overheads introduced in the experimental system. It tracked the communication between the Registry and Resolver components in order to determine the time added per request.

Table 12: Overview of tests conducted on the experimental and traditional system

The following sections describe each test and the methodology used in greater detail. It also discusses the results obtained and what can be inferred from them.

4.4 Testing

4.4.1 System Stability and Coherence Test

Description

As with all systems that are developed, a full test of the overall system is required to ensure that it is working correctly and according to the specification. The following test aims at ensuring that the system remains correct and consistent at all times. At the same time, we hope to show that migration and replication occurs and test that the time to service requests has been reduced.

Since this test was aimed at checking the experimental system, it is not compared to the traditional one. This test was conducted on the experimental system in various ways. In the first test, the experimental system was run without the load balancer, migration and replication, effectively checking the performance of the system with the Registry and Resolver communication overhead. In the second test, the loader balancer and replication was enabled but not the migration. The third test used the load balancer and migration with the replication turned off. The fourth and final test used the experimental system which included the load balancer, migration and replication.

The layout of the components is shown in Figure 12. The test with replication only had the components spread across the 3 nodes. This was needed since replication only occurs when a component instance is the only one on a node.

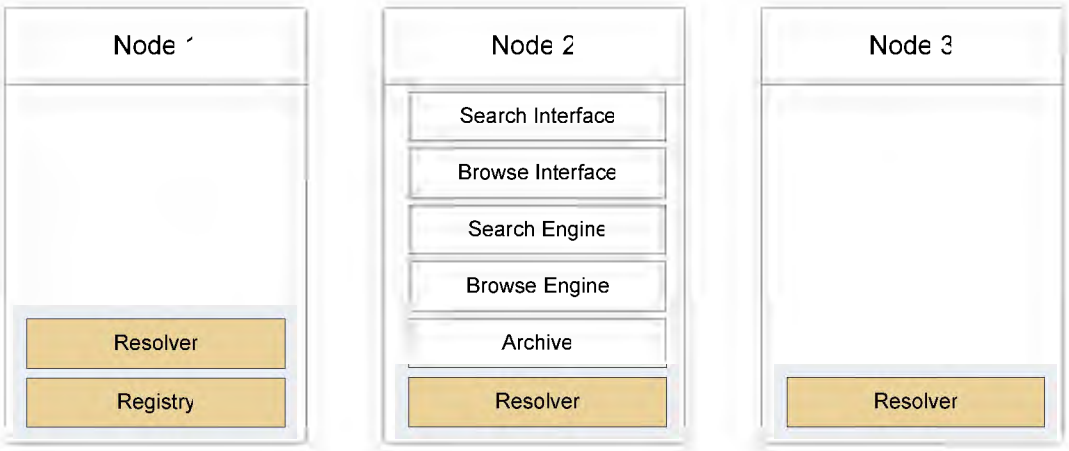

Figure 12: Layout of experimental system for testing.

The requests were sent to the system in three ways, namely:

- sending batch requests of 400 (4 requests simultaneously) to the search interface,
- sending batch requests of 400 (4 requests simultaneously) to the browse interface and
- sending batch requests of 400 (4 requests simultaneously) to both interfaces simultaneously.

For each case the time taken to complete all requests was noted as well as the times for single requests. This will allow us to gain an understanding of the system in terms of migration, replication and performance. It is expected that the system utilising the migration and replication will perform better.

Results

Figure 13 and Figure 14 show the results for the browse request times. The first one shows the results obtained by testing the browse interface alone. The second shows the browse interface results while the search interface is being queried as well. The second two graphs (Figure 15 and Figure 16) show the search request results in the same manner.

From the results obtained we can see that testing each interface individually does not provide much of a performance gain when comparing all the different tests. This is attributed to the number of requests being sent to the interface. Since the node only has to handle four requests simultaneously, no major gain is derived by using migration and/or replication. In all the graphs we see spikes in request time when only migration is enabled. This occurs since all requests dependant on a migrating instance are delayed until the migration is complete.

What is interesting to note here is that the browse interface performed best with migration and no replication. The browse response times are quite long, when compared to the search, and this affects its ability to replicate and migrate. After migration has occurred we see a drop in request time. The major factor here is that the component instances are able to leave the originating node, thus freeing up resources to be utilized by other

components. On the destination node though, the resource usage will increase as it will be used for computation by the migrated service. Since all database queries are still completed on the originating node, it will not become lightly loaded.

Using replication only would require the originating node to service search and/or browse requests in conjunction with the incoming database queries from replicated instances. This explains the repetitive peak and trough pattern in the graphs as the replication causes increase load followed by a decrease once the process is completed. This process increases the load on nodes with replicated instances and in turn also increases the load on the originating node as all database queries are redirected to it. Thus the overall system would show a slow down if replication occurs continuously for a long period of time.

When both interfaces are tested, a better improvement obtained for the browse and an even greater one for the search interface. This is due to the load monitor favoring the search component somewhat since it completes more requests in one load cycle. Again, we see the browse performance being the best with migration only and we see a drop in the request time near the end of the test. This drop occurs since all the search requests would have been completed and thus no longer uses the resources available. With the search we see that utilizing migration and replication performs best. This is attributed to multiple search and archive instances being created by replication and migration to other nodes (since it is favoured over the browse component).

A few patterns can be seen when using migration or replication only. With replication only we observe an increase in request time and then a decrease after. This pattern occurs since the overloaded machine still has the original component instance servicing requests while the replication is occurring. With migration only, we see that there is a spike in the request time during migration. Since no other instances of the component are available, the request has to be delayed until the migration is complete.

Another point highlighted here is that the system without the load balancer, replication and migration (pink series on figures 13, 14, 15 and 16) generally performed worse than the other configurations. This highlights the fact that the flexibility of a dynamic system would result in better performance when configured correctly.

The other tests that follow pit the experimental system developed against the traditional one it was based on.

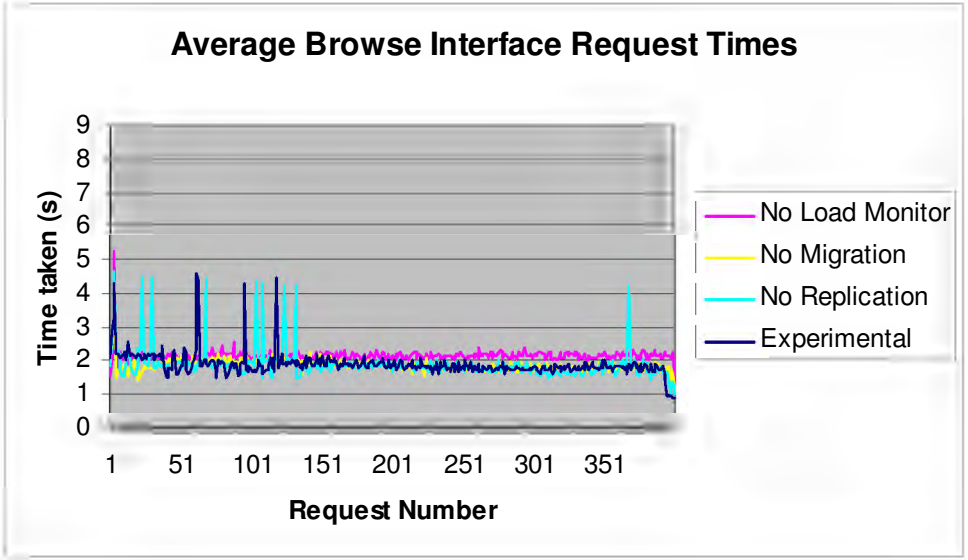

Figure 13: Browse request times with and without migration and replication.

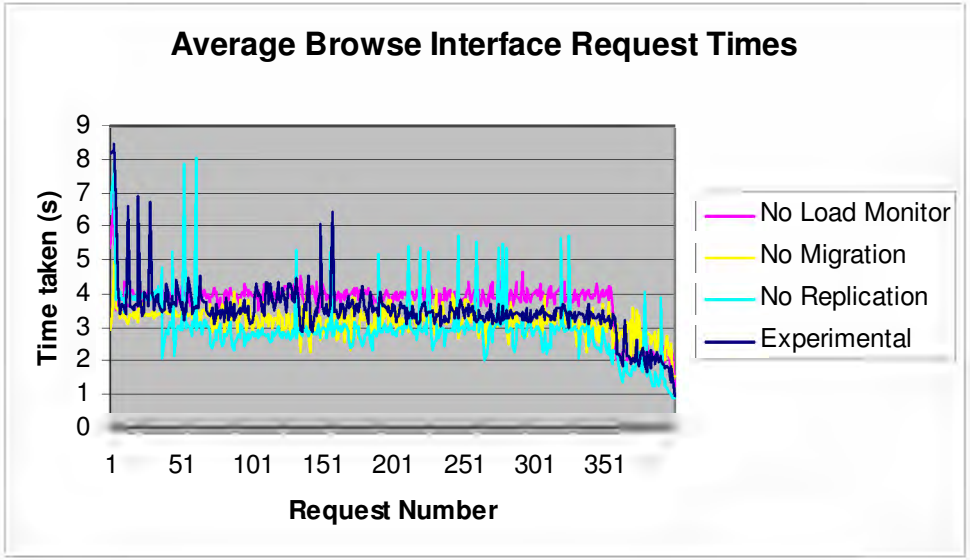

Figure 14: Browse request times with and without migration and replication, running while search is being tested.

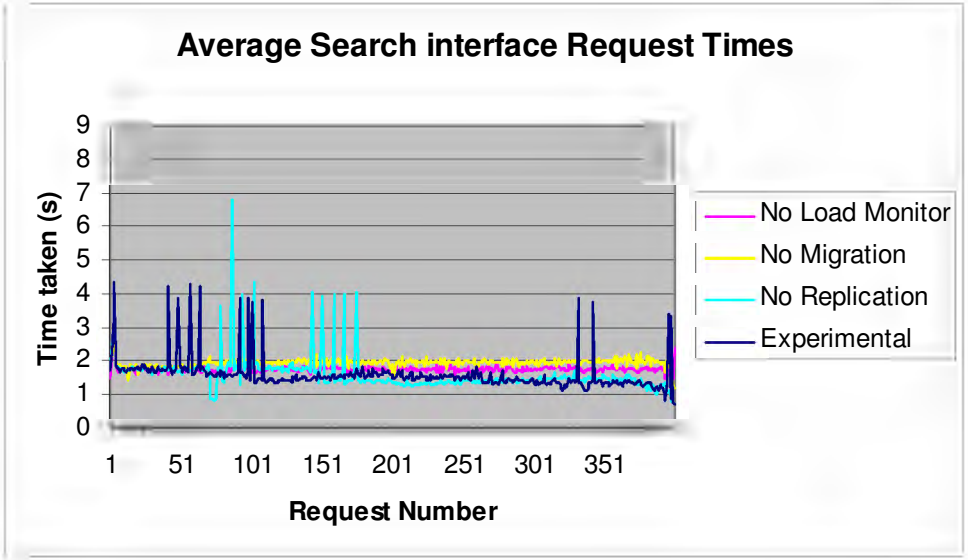

Figure 15: Search request times with and without migration and replication.

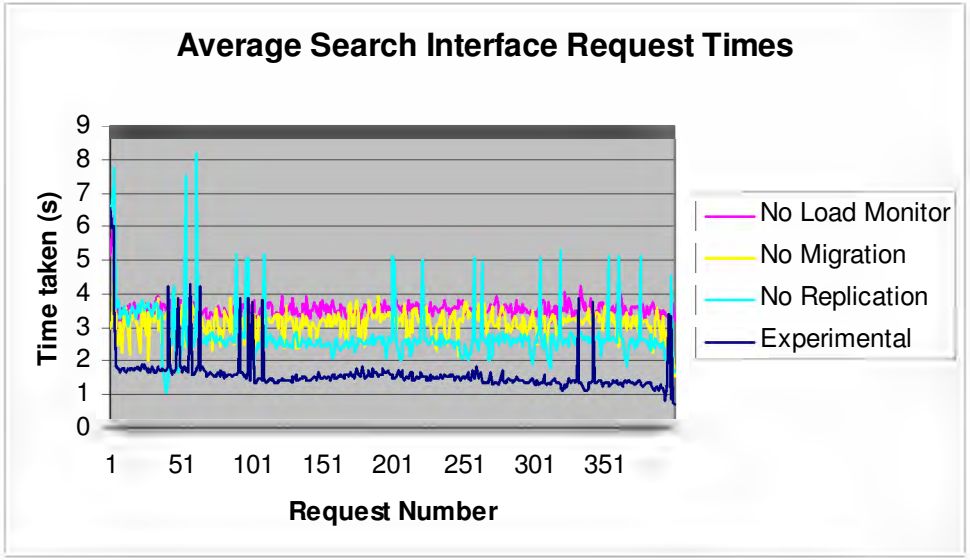

Figure 16: Search request times with and without migration and replication running while browse is being tested.

4.4.2 System Performance Tests

The second set of testing is aimed at comparing the performance of each system under different conditions. Each system was spread across the 3 nodes in different layouts and tested with a series of requests as described in the previous test. The results for these tests depend largely on whether the requests are sent serially or simultaneously. We expect that traditional system would perform better with serial requests while the experimental system should perform better when dealing with multiple simultaneous requests.

Serial testing

The layout (Layout1) for the first performance test is illustrated in the diagram below (Figure 17). The components shaded are those that are specific to the experimental system. The layout for the traditional system is the same with those components removed. The central load balancer and load monitors are not depicted in the layout.

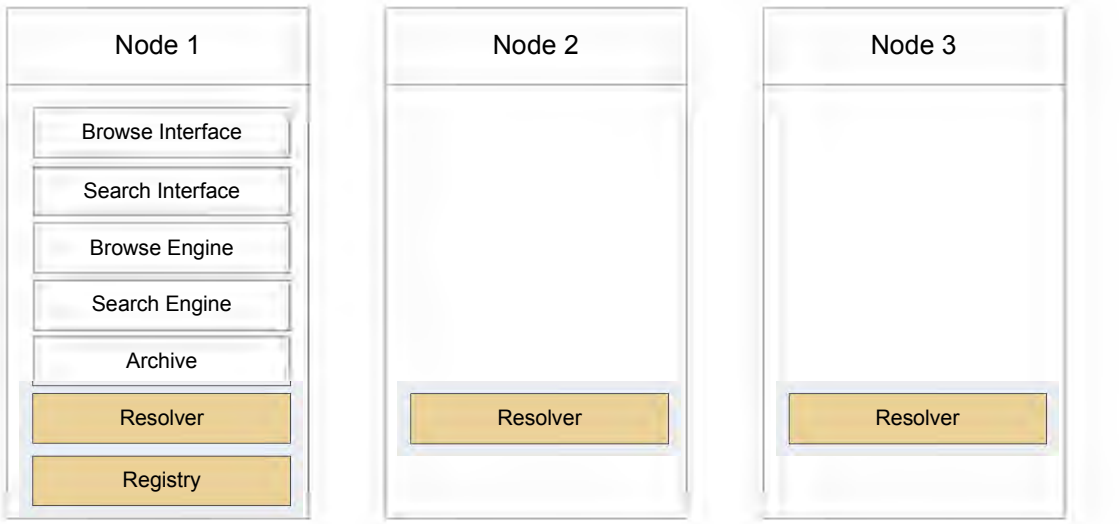

Figure 17: Layout of components across the cluster for first performance test. This does not show the load balancing related components.

The second layout used placed the interfaces on the first node, the engines and archive on the second one and in the case of the experimental system, the Registry on the third. In the third layout each interface along with its associated engine was placed on a node. The archive and Registry (for the experimental system) was placed on the last available one. So the first node had the search interface and engine, the second the browse interface and engine and the third the archive and Registry.

For each layout used, each interface was tested individually and then both were tested simultaneously. These tests consisted of sending 500 requests to the interface serially. The first test was to the browse interface only, followed by the second test to the search

interface only. The third test was done on both the browse and search interface simultaneously but requests were still send serially to each one.

With the first and second test, the traditional system was expected to perform slightly better as the experimental system has additional overheads per request. Since the requests are sent serially, the system only needs to service one request before moving to the next one. The load on the machine would thus only increase by the amount of processing required to complete one request. In the third test we expect the experimental system to perform as well as or better than the traditional system. This is due to the fact that the nodes' load will increase by the amount of processing required to service two requests simultaneously. The experimental system should use migration or replication to relocate the load across the 3 nodes, thus spreading the overall load and possibly increasing performance. Since the traditional system cannot move its components, the performance is expected to remain static.

Results

The first two graphs (Figure 18 and Figure 20), show the browse and search request times for serial requests. In these tests, each interface was tested without the other one being queried at all. The second set of graphs, (Figure 19 and Figure 21), show the request times when both interfaces were queried at the same time. Again, requests were sent serially to each interface. These graphs show the results of the first layout (Layout1) while the results obtained from the other layouts can be found in Appendix B as they are similar to the graphs depicted below.

From the graphs below (Figures 18 – 21) and those in Appendix B, we can see that the difference between the two systems in terms of request times is quite small. These time differences are below 1 second. From Table 13 below and those found in Appendix C, we can see that both migration and replication are active in the experimental system. With each interface being tested individually, we see the migration and replication of the engines being tested, as well as the archive component they rely on. From the tables we can see that the migrations and replications tend to stay clear of the originating node. This is attributed to the remote database access on those nodes. Even with no components on these nodes, the load is dependant on the number of remote database accesses.

Run 1		**Node 1**	**Node 2**	**Node 3**
Initial	Browse	1	0	0
Initial	Search	1	0	0
Initial	Archive	1	0	0
Final	Browse	0	1	0
Final	Search	1	0	0
Final	Archive	0	1	1

Table 13: Instance locations before and after sending test requests to the browse interface.

In Figure 18 and Figure 20, we see that the migration and replication of components do not provide any overall gain as only one request is sent at a time. This is a result of the load balancer detecting that the node providing the service is overloaded when compared to the nodes with no components on them. These migrations and replications actually slow the system performance as they create a sharp increase in response time and do not provide any major gain as requests are being serviced serially.

When sending serial requests to both interfaces simultaneously (Figure 19 and Figure 21), we see how the migration and replication has benefited the experimental system. In these tests, both interfaces are querying the same archive, and as a result, a small delay in the traditional system is noted when compared with serial requests to one specific interface only. Since the experimental system can migrate and replicate the archive, a small gain in performance can be obtained. This allows the experimental system to perform similar or better than the traditional one. Given that the requests are sent serially, the traditional system still performs well despite needing to service both components. When querying both interfaces, migration and replication cause the load to be spread across the nodes. As a result, migrations and replications would become less frequent. With the single interface tests, we can see the experimental system spiking more often as migrations and replications are limited to one component instance type.

This test shows clearly that serial requests are more suited for the traditional system. This was not unexpected since the design of the experimental system is more suited to multiple simultaneous requests.

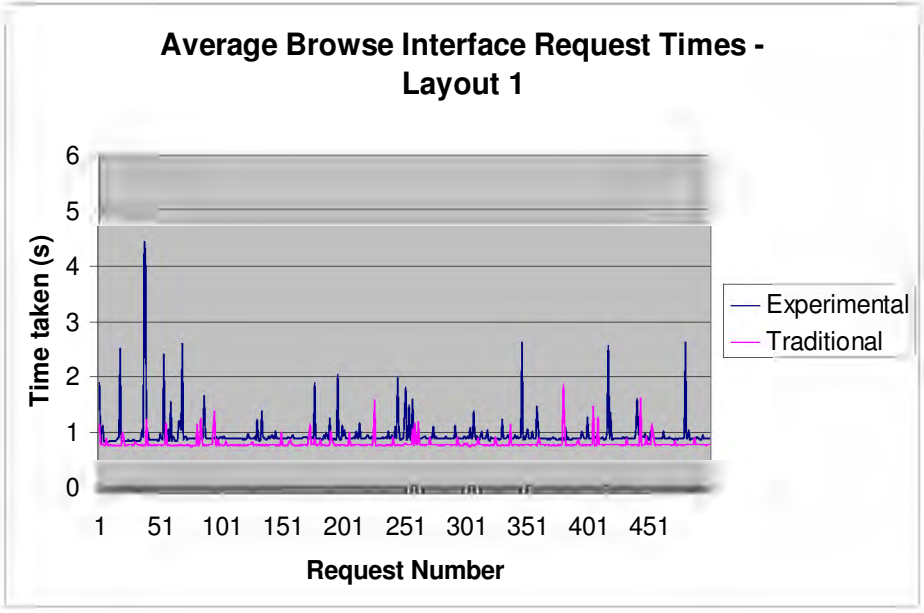

Figure 18: Browse request time for 500 serial requests. Experimental system shows no gain due to requests being serviced serially.

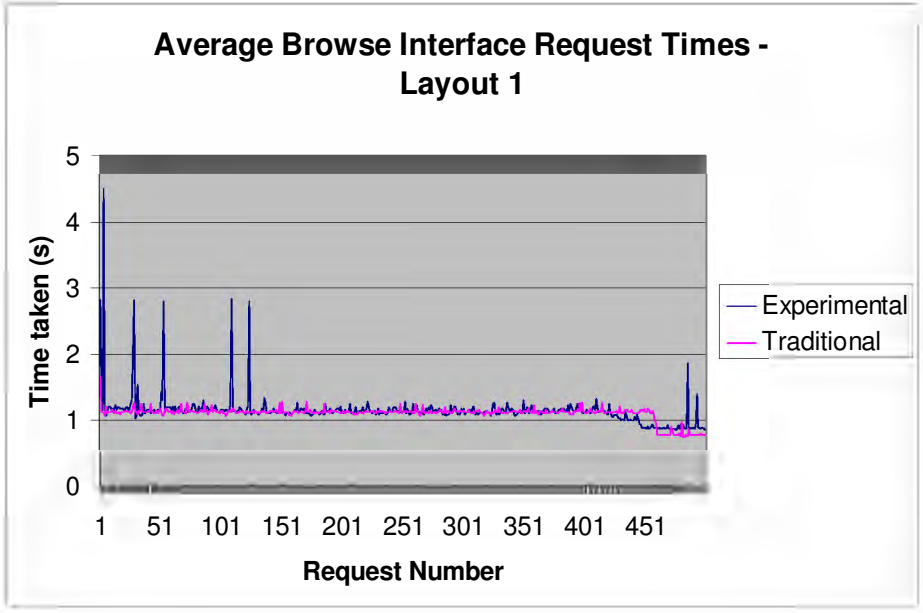

Figure 19: Browse request time for 500 serial requests while search is being tested. Experimental system shows no gain due to requests being serviced serially.

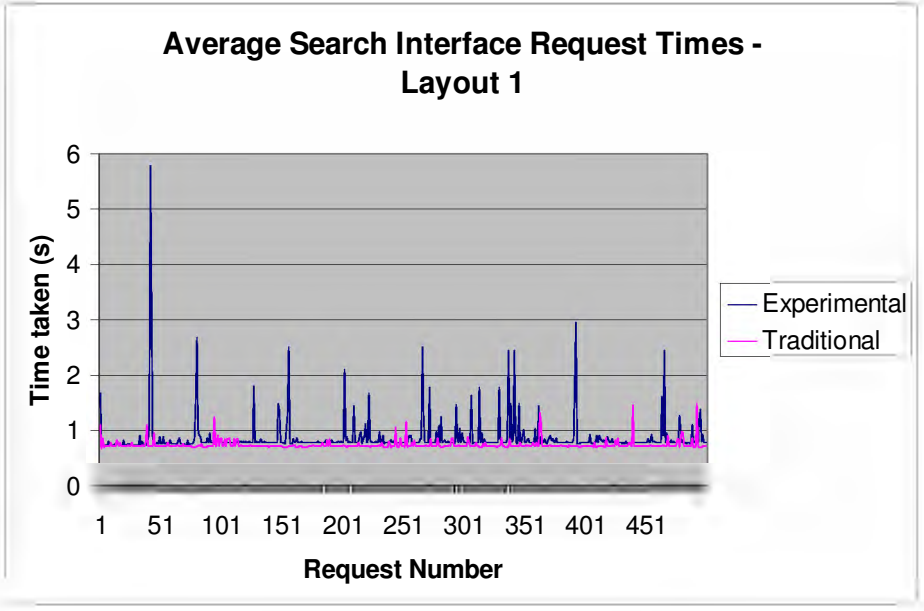

Figure 20: Search request time for 500 serial requests. Experimental system shows no gain due to requests being serviced serially.

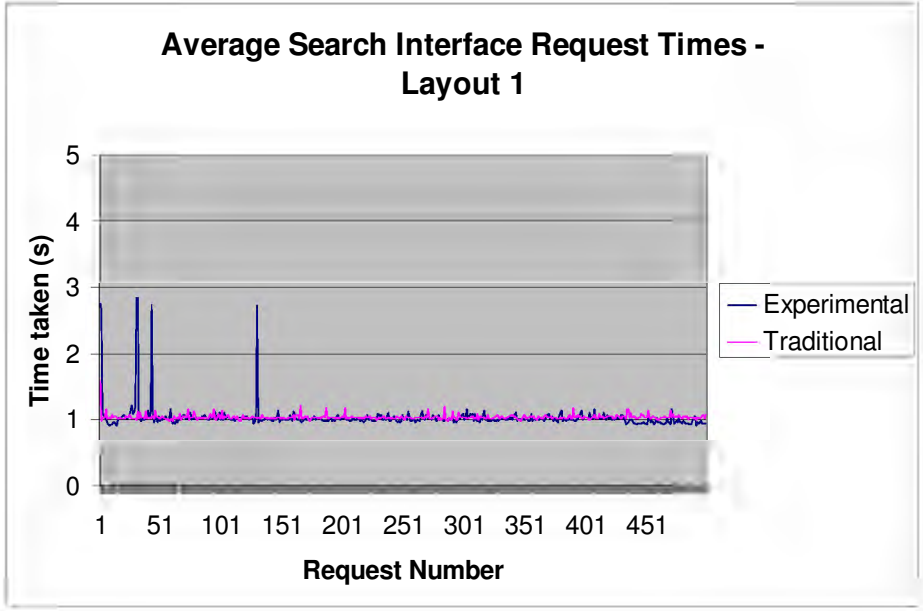

Figure 21: Search request times for 500 serial requests while browse is being tested. Experimental system shows no gain due to requests being serviced serially.

Simultaneous testing

For the fourth performance test, simultaneous requests were used. The layouts used in this test are as depicted in Figure 12.

The requests for this test were sent to both systems as follows:

- Both interfaces were queried with requests simultaneously. The requests were sent using 16 processes each one sending 128 requests. Half of the requests were sent to the search interface and the others to the browse, thus resulting in 1024 requests per interface to the system.
- The browse interface was queried on its own by 8 processes each sending 128 requests. This resulted in 1024 requests being sent to the browse interface while no requests were sent to the search interface at all.
- The search interface was queried on its own by 8 processes each sending 128 requests. This resulted in 1024 requests being sent to the search interface while no requests were sent to the browse interface at all.

With this test, the experimental system should perform better. The main reason behind this is that requests need to be serviced simultaneously and one component instance could easily degrade in performance when needing to handle multiple requests. With the ability to migrate and replicate component instances, the experimental system should increase the availability of the component instances servicing requests. Thus request service time should increase initially as migration and replication occurs followed by a decrease once the load has been spread.

In the traditional system, the static layout cannot be changed and hence the request times are expected to increase. This would remain constant until all the requests have been completed as the system does not adjust at all.

Results

The two graphs below (Figures 22 and 23) show the results for both interfaces being tested at the same time.

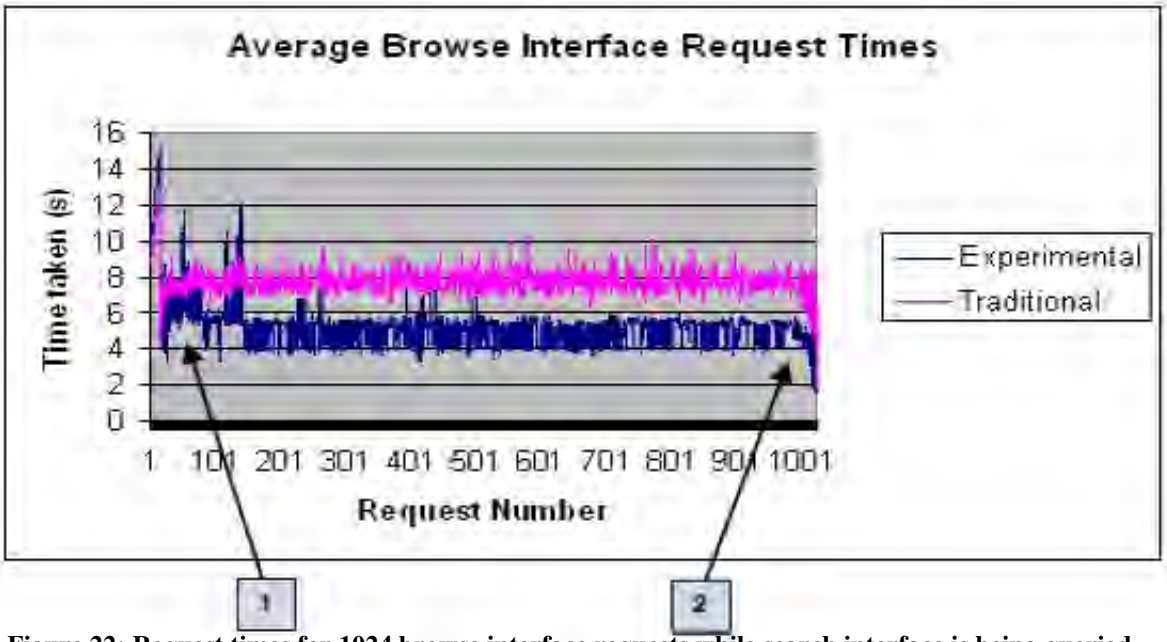

Figure 22: Request times for 1024 browse interface requests while search interface is being queried. Experimental system performs worse initially (1) followed by better performance once the load has been spread across the nodes. Request times also taper off (2) at the end as no more requests are serviced.

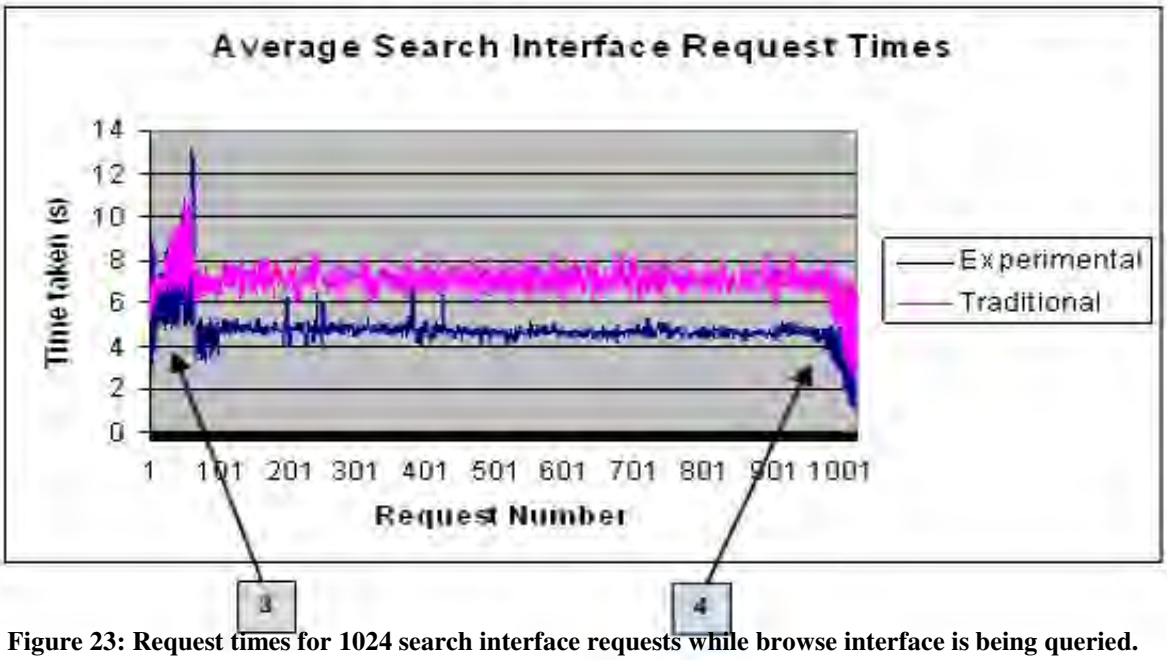

Figure 23: Request times for 1024 search interface requests while browse interface is being queried. Experimental system performs worse initially (3) followed by better performance once the load has been spread across the nodes. Request times also taper off (4) at the end as no more requests are serviced.

The next two graphs (Figures 24 and 25) show the request times for each interface being tested individually.

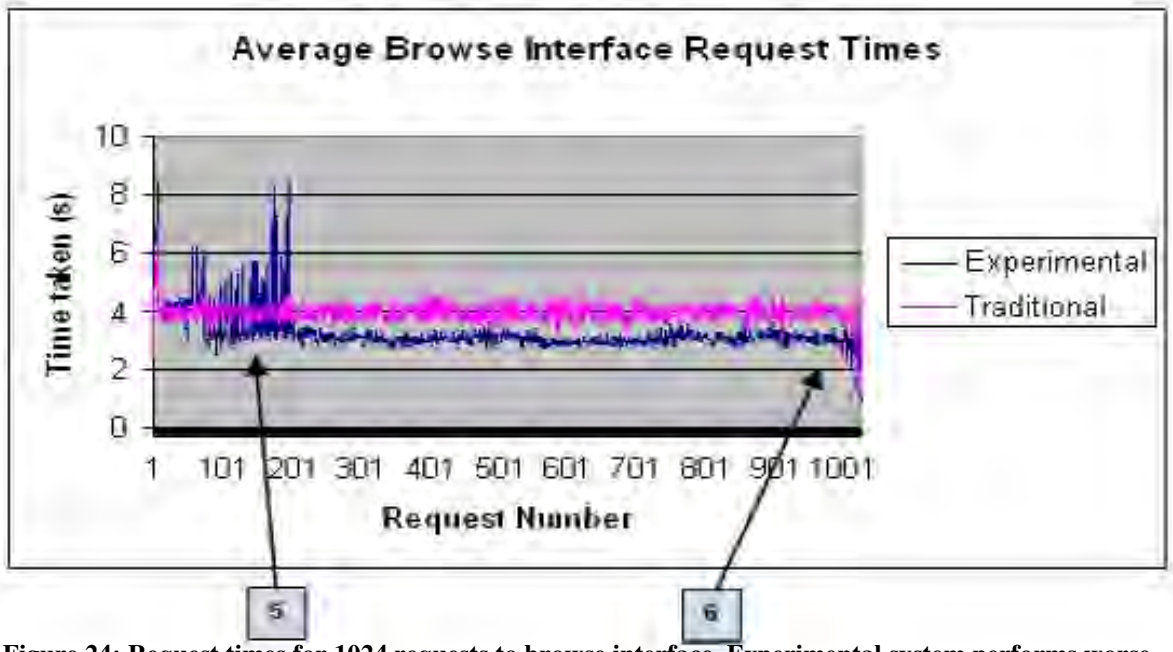

Figure 24: Request times for 1024 requests to browse interface. Experimental system performs worse initially (5) followed by better performance once the load has been spread across the nodes. Request times also taper off (6) at the end as no more requests are serviced.

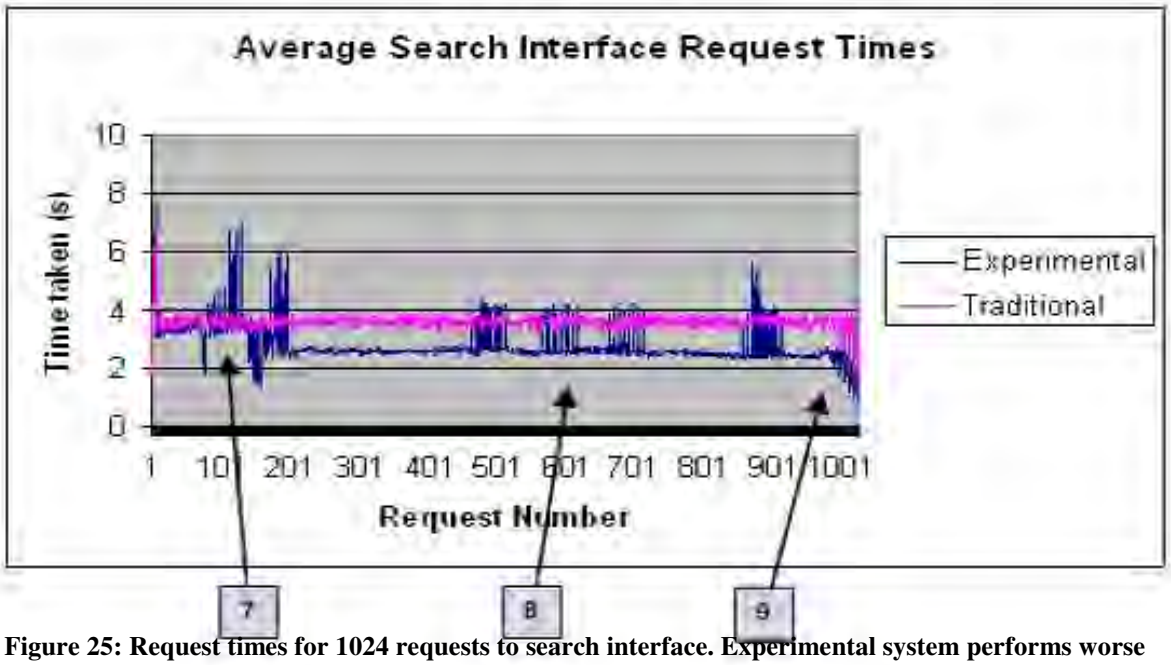

Figure 25: Request times for 1024 requests to search interface. Experimental system performs worse initially (7) followed by better performance once the load has been spread across the nodes. Migration and replication impacts as system attempts to adjust (8) to load. Request times also taper off (9) at the end as no more requests are serviced.

In this test we see the conditions which favour the experimental system as it is better suited to deal with simultaneous requests.

With the single interface testing, benefits are obtained through the migration and replication of the component instances. While the traditional system manages requests at a regular pace, we see the experimental system spiking during migration or replication initially. This allows the system to spread the load across multiple nodes and eventually we see a decrease in request times, as was expected. For the traditional testing we see an average or mean response time of 3.95 seconds with a standard deviation of 0.32 for the browse requests. With the experimental system, we see the average response time reduced to 3.23 seconds with a standard deviation of 0.63 for the browse requests. For the search interfaces we see a similar pattern. The traditional systems average request time is 3.59 with a standard deviation of 0.28 while the experimental systems are 2.78 and 0.67 respectively. The larger standard deviations associated with the experimental system can be attributed to the migration and replication processes which cause spikes in response times.

With both interfaces being tested, the load on the entire system is higher. Here the dynamic nature of the experimental system is highlighted as the gains achieved are even bigger. The average response times associated with the traditional components have increased from 3.95 seconds to 7.77 seconds for the browse requests and from 3.59 seconds to 7.10 seconds for the search requests. The standard deviations have also increased to 0.76 and 0.67 for browse and search. The delay in the traditional system can be attributed to the fact that both interfaces are relying on one archive component which has to service multiple requests. For the experimental system, we see increases in the average time per request but they are not as drastic as those of the traditional system. For the browse component, the average request time is 5.26 seconds with a standard deviation of 1.37. With the search component, we see the average time increasing to 4.79 seconds with a standard deviation of 0.85. Again, the larger standard deviations for the experimental system can be attributed to migration and replication. As the load on the entire system is greater, more replications and migrations would be needed for the system to stabilize.

The graphs above (Figure 22 – Figure 25) show how the experimental system reacts to increasing numbers of requests. Markers 1, 3, 5 and 7 show the system trying to adjust itself, through migration and replication, in order to meet the demands of incoming requests. This is then followed by a phase where the system has stabilized and minimal or no adjustments are being made. This results in a relatively uniform response time range except for the jumps in request time shown by marker 8 in Figure 25. Here, the load balancer has determined a node to be overloaded and an adjustment is made. After the adjustment we see a minute decrease in performance time. This process, of re-adjusting, occurs numerous times in order to distribute the load evenly across the nodes within the system. Markers 2, 4, 6 and 9 show the request times decreasing. This decrease occurs due to the decrease in incoming requests since some of the processes have already completed the specified number of requests.

The simultaneous testing has shown that the experimental system performs better under larger loads. This is attributed to its ability to spread the load across multiple nodes and adjust the components to meet the required need as it changes.

4.4.3 Load Pattern Testing and Load Monitor parameters

It is clear that certain conditions of load and request patterns would help to determine the optimum operating conditions of both systems. A series of different request patterns was thus used in order to better understand the operation of both systems. Again, the layout in Figure 12 was used.

Different request patterns were administered to both systems. These include an increasing number of requests, random number of requests and a periodic request pattern. Each pattern should reveal information about the use of migration and replication. Additionally, inferences can be made about the Central Load Monitor by varying the parameters it uses.

The first pattern that was tested was the one in which the number of simultaneous requests was doubled after every 25 requests. At the start of the test, only one request was sent to each interface. After 25 requests, the number was increased to two requests per interface simultaneously. After 50, this was increased to four requests per interface, and finally, at 75, to eight requests per interface. This resulted in 385 requests being sent in total. Each interface was tested individually and no simultaneous testing was done. Simultaneous testing would allow one component's resource requirements to affect the other. This could potentially skew the results of a component's performance when comparing the two systems.

The results for the two systems can be seen in the graphs below.

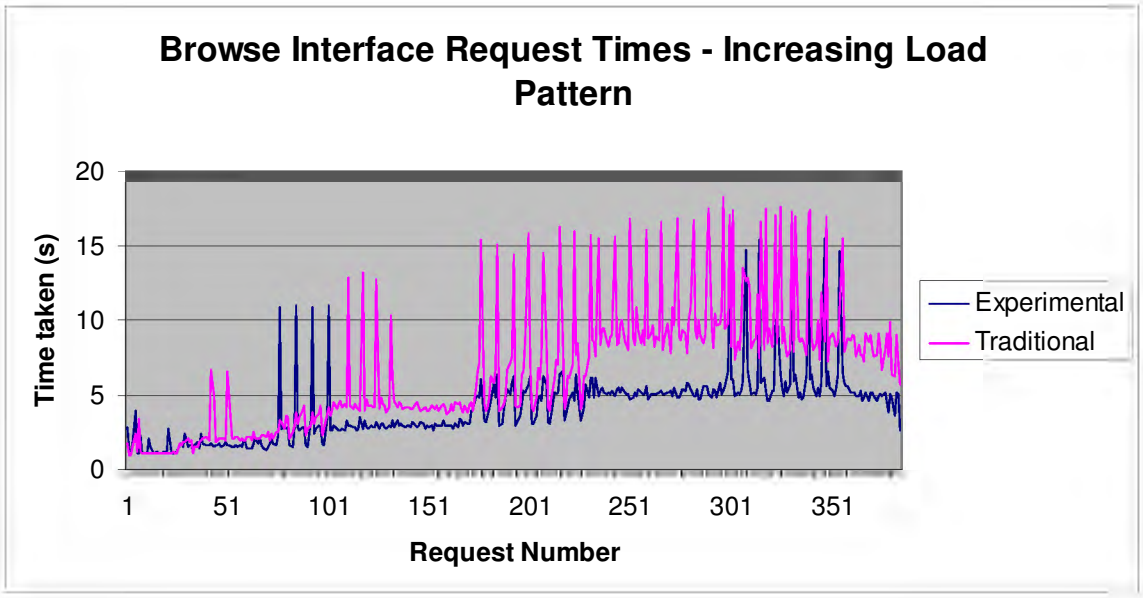

Figure 26: Browse request times for increasing load pattern.

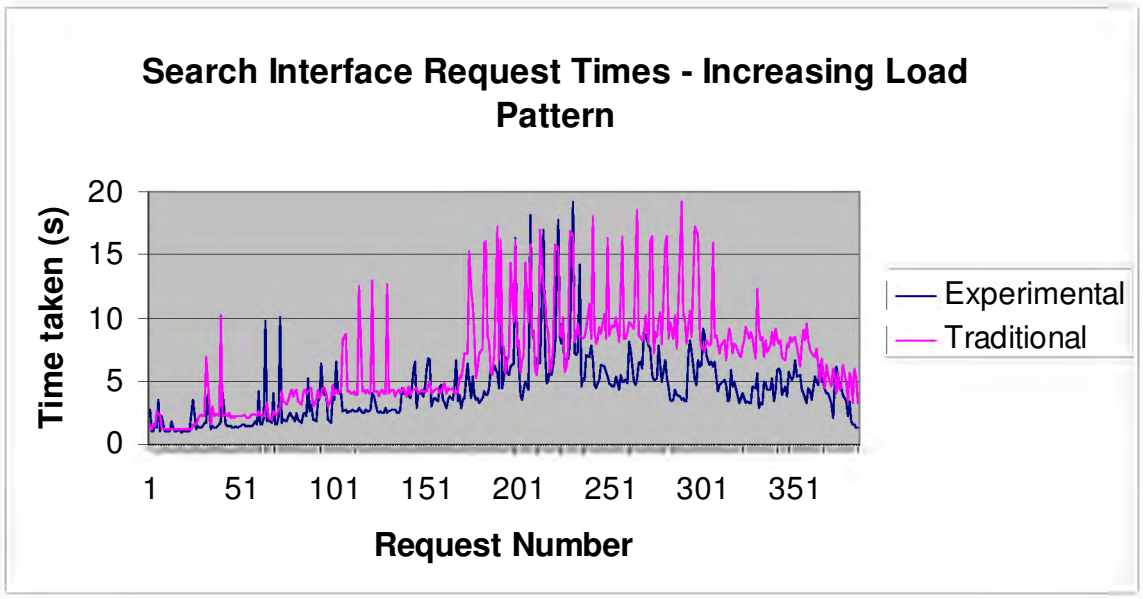

Figure 27: Search interface request times for increasing load pattern.

The graphs above (Figure 26 and Figure 27) show that the experimental system performs better than the traditional one as the load increases. Initially, we can see that the traditional and experimental systems perform relatively the same. As the load increases, we can see the response times diverge between the two systems with the traditional one taking longer. This difference is at a maximum when the load peaks, which is as expected based on the previous performance testing. Additionally, we can see that minimal improvement is achieved when 2 simultaneous requests are sent. When requests are above 4 simultaneously, the results show that the experimental system's performance improvement is much more.

To evaluate another aspect of this test, the load monitor was adjusted to evaluate its effect on the experimental system's performance. The load monitoring interval was adjusted and tested for intervals of 10, 20, 30, 40 and 50 seconds. The results of these tests, for the search and browse engines, can be seen in Figure 28 and Figure 29.

At markers 2, 4 and 5 we can see that the monitoring interval of 10 seconds causes the system to adjust itself faster. At the same time, we see an increase in request time is experienced at marker 1 and 3. This occurs due to increased number of migrations and replications, due to the shorter monitoring interval, to the point where it may be detrimental to the system. At the other extreme, with a monitoring interval of 50 seconds, we see a slow reaction time to increased requests. This increases request time as replication and migrations take longer to occur. What this does allow though is a chance for the load balancer to make a better decision on which instance to migrate since more request information is available. It does however leave the system open to adapting too slowly to changes in request patterns.

In both the search and browse graphs we see spikes in the request time. The majority of the spikes occur when the load interval is greater than 30 seconds. When a migration or replication does occur, while using larger intervals, we see the time increasing dramatically. This is attributed to the lack of previous adjustments since the load monitoring interval is too large.

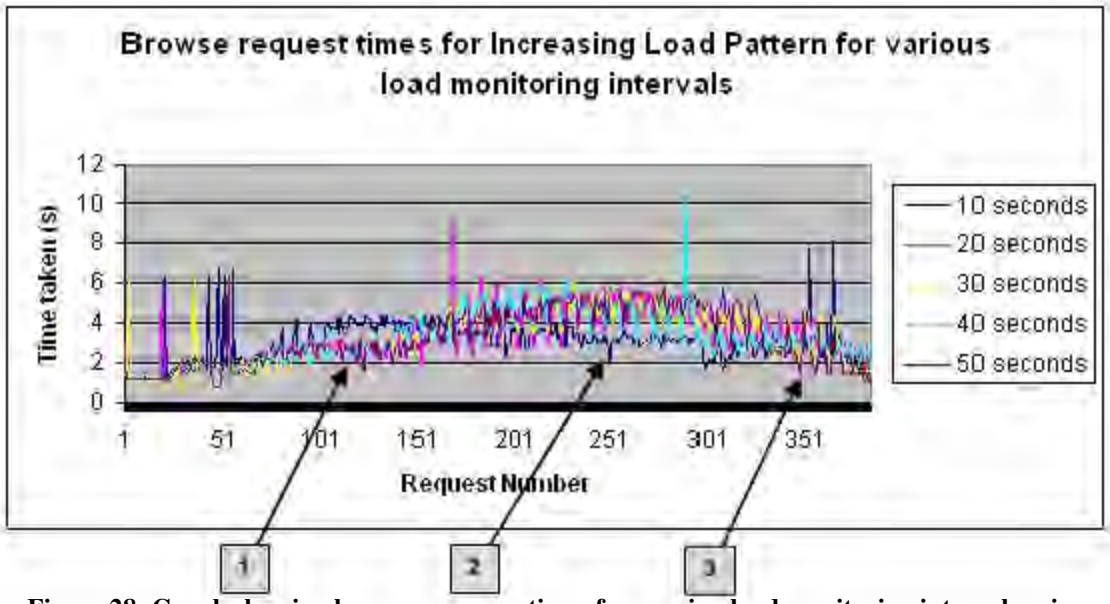

Figure 28: Graph showing browse response times for varying load monitoring intervals using increasing load pattern. We see the 10 second interval adjusting faster (2) when the load increases and increase in request time due to adjusting to quickly (at 1 and 3).

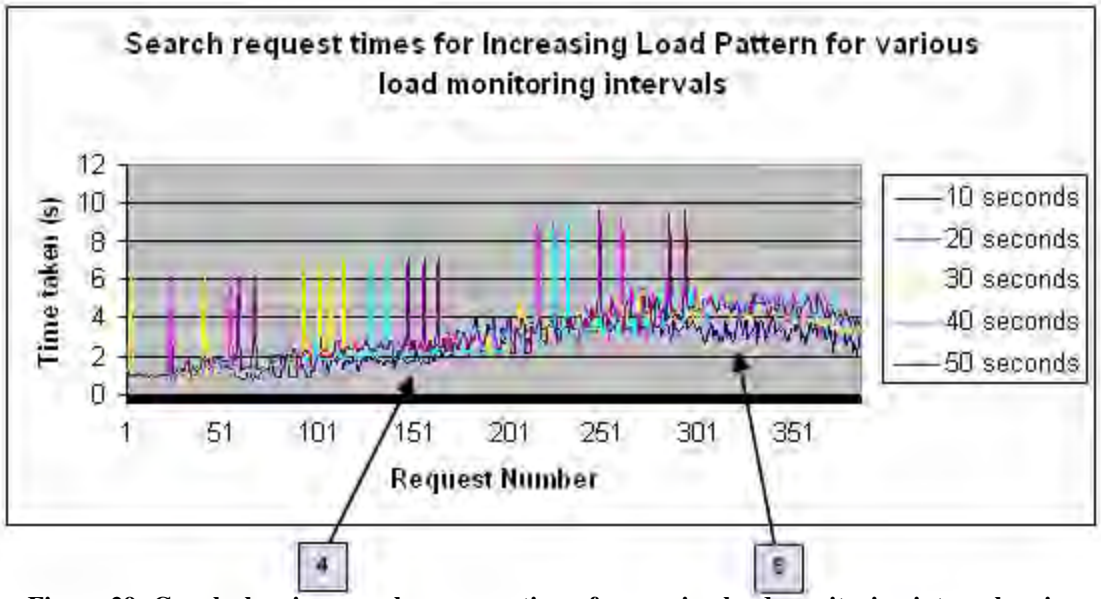

Figure 29: Graph showing search response times for varying load monitoring intervals using increasing load pattern. We see the 10 second interval adjusting faster (at 4 and 5) when the load increases.

The next parameter to be monitored was the overload threshold since the variations in load across the nodes determine whether migration and replications occur. Altering this value would allow the load balancer to be more or less sensitive to variations in the loads within the cluster. This was varied from the average load plus 1, 2, 3, 4 and 5 standard deviations. The results for these tests can be seen in Figure 30 and Figure 31 below.

Here we see that the smallest deviation causes spikes in the request time when the number of requests is small (1 or 2). At higher number of requests, greater than 4, we see it performing better with minimal spiking. The largest deviation obtains similar results to the smallest except that it tends to spike with a higher number of requests as well. In the phases between markers 1 and 2 and markers 4 and 5, we see consistent request times with minimal or no spiking for the smallest and largest deviations. A standard deviation of 3 or 2 seems to perform most optimally in both conditions of small and large numbers of requests.

From our results, we see that the overload criterion is slightly dependant on the number of requests. With fewer or serial requests, a larger load difference between nodes is required and with many simultaneous requests, a medium to large variation is required. This highlights the need for the system to adapt based on the current need and which services are being utilized at the time.

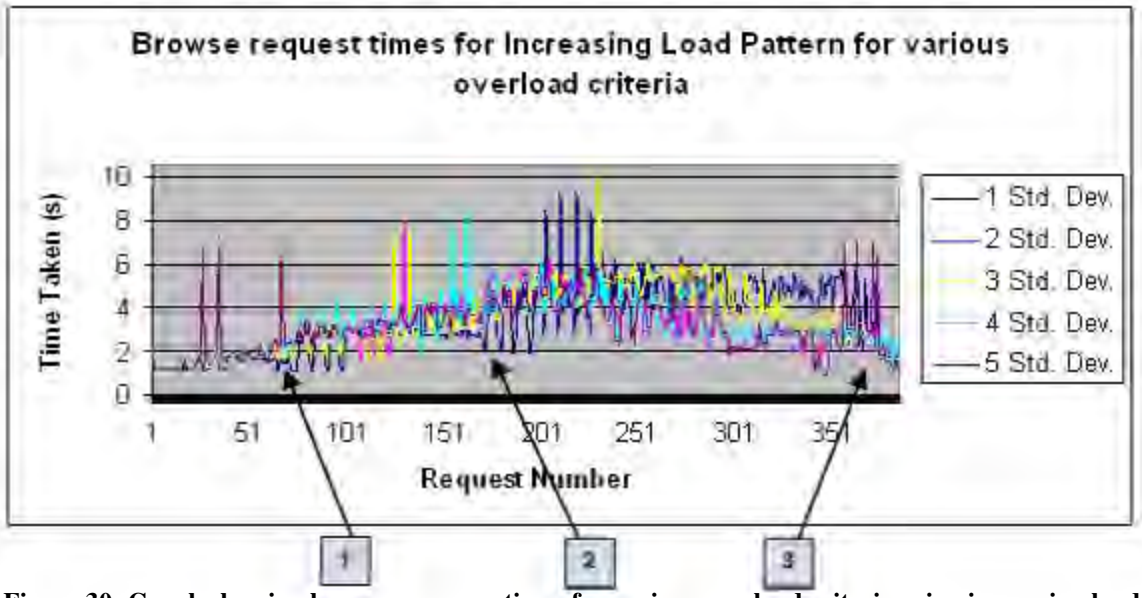

Figure 30: Graph showing browse response times for various overload criteria using increasing load pattern. Small and larger standard deviations perform better during lower loads (1 and 2) while the request times taper off as number of request decrease (3).

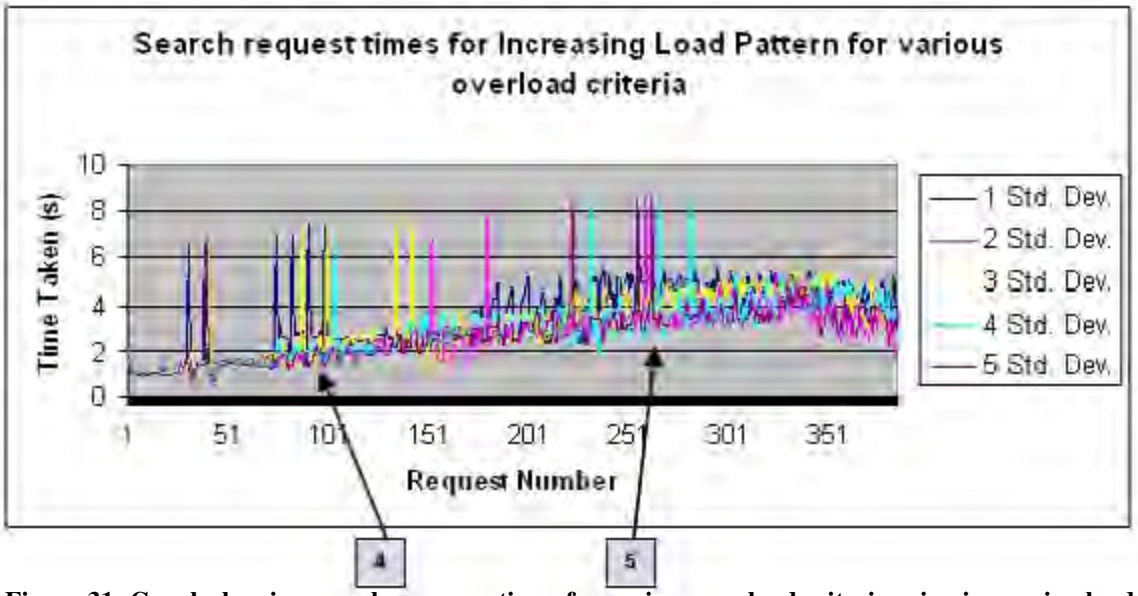

Figure 31: Graph showing search response times for various overload criteria using increasing load pattern. Larger standard deviations perform better at low loads (4 and 5) while it tends to adapt to slowly at higher loads.

The second load pattern used a random number of requests to both interfaces simultaneously. Again the results from each system can be seen. A base of 1 request was sent to each interface simultaneously. Additionally, a random amount of additional requests were sent. These ranged from 2 additional requests to 16 additional requests. As the number of requests was completely random, each test run may have resulted in a different total number of requests being sent. The request run with the least amount of requests was used as the cut-off. Test data from other runs would thus be limited to the

cut-off value in order to have a fixed size dataset to work with for calculations. The results obtained can be seen in Figure 32 and Figure 33, for browse and search respectively.

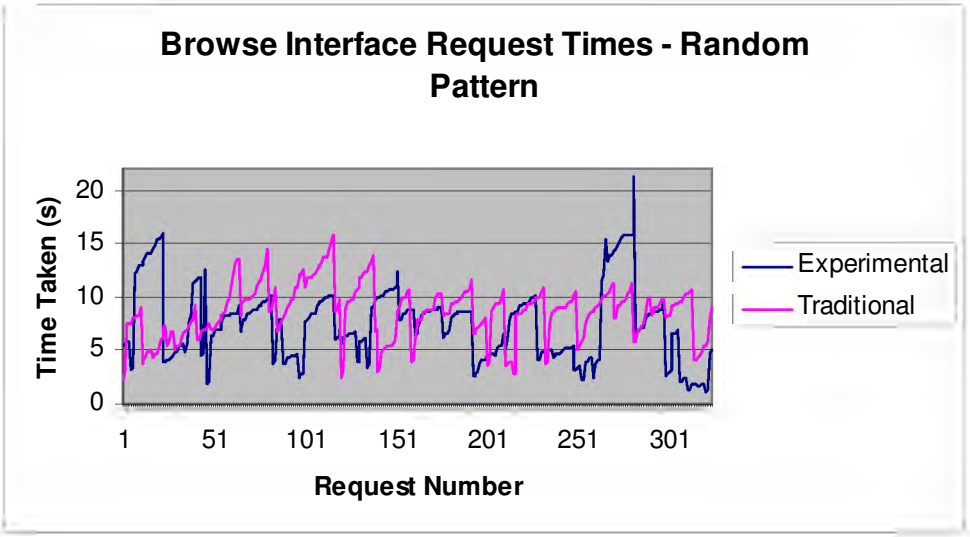

Figure 32: Browse interface request times for random load pattern.

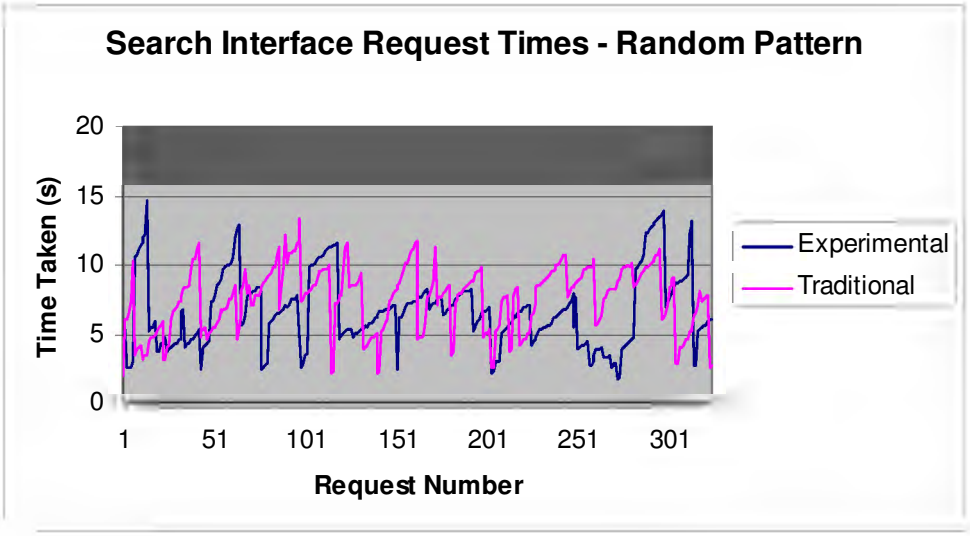

Figure 33: Search interface request times for random load pattern.

Since the requests generated were totally random, a minor improvement can only be expected from the experimental system. This can be seen in the Figure 32 and Figure 33, since the request times for the experimental system remain below that of the traditional one. This may be a purely random result but the spikes of the experimental system tend to be smaller than those of the traditional one. This is likely attributed to the migration and replication since the experimental system resulted in having 5 search engines, 1 browse engine and 2 archive engines after the test was complete. This is compared to the 1 browse, search and archive within the traditional system.

The average response time for the traditional system was 7.56 seconds for the search requests and 8.55 seconds for the browse requests with standard deviations of 2.30 and 2.57 respectively. For the experimental system, the average time was 6.66 seconds for search requests and 7.33 seconds for the browse requests. The standard deviations for the experimental system were 2.70 for search requests and 3.51 for the browse ones. In both cases, we see the experimental system performing better. With the standard deviation of the experimental system being higher than that of the traditional system, we can see that the data is more spread. For the search requests this difference is small and the gain in average response time outweighs this. For the browse requests the other hand, the difference between the standard deviations is bigger. This could indicate that the traditional system may perform better. We could explain this as the experimental system attempts to adjust by looking into the amount of incoming requests. As this is completely random, some adjustments may cause the system to perform worse if the incorrect decision is made.

The random request pattern was also tested by varying the load cycle time and overload criteria. The graphs for this can be seen on the following two pages from Figure 34 until Figure 37. In these graphs we see that there is a decrease in the request time. Although this cannot be seen directly from the graph, it exists. By adding a trend line to the graph we can gauge the direction in which it is moving over time. The graphs with various trend lines added can be seen in Appendix D. This helps us to see how the request times are changing over time. From Appendix D we see that the 30 second load interval time has a decreasing trend whereas the 10 second one has a more flat or constant one. This highlights the need to monitor the request pattern over a longer period in order to make the best decision for migration or replication. The same process can be applied to the overload criteria tests in order to determine whether the request time is increasing or decreasing. The graphs for these can be seen in Appendix E.

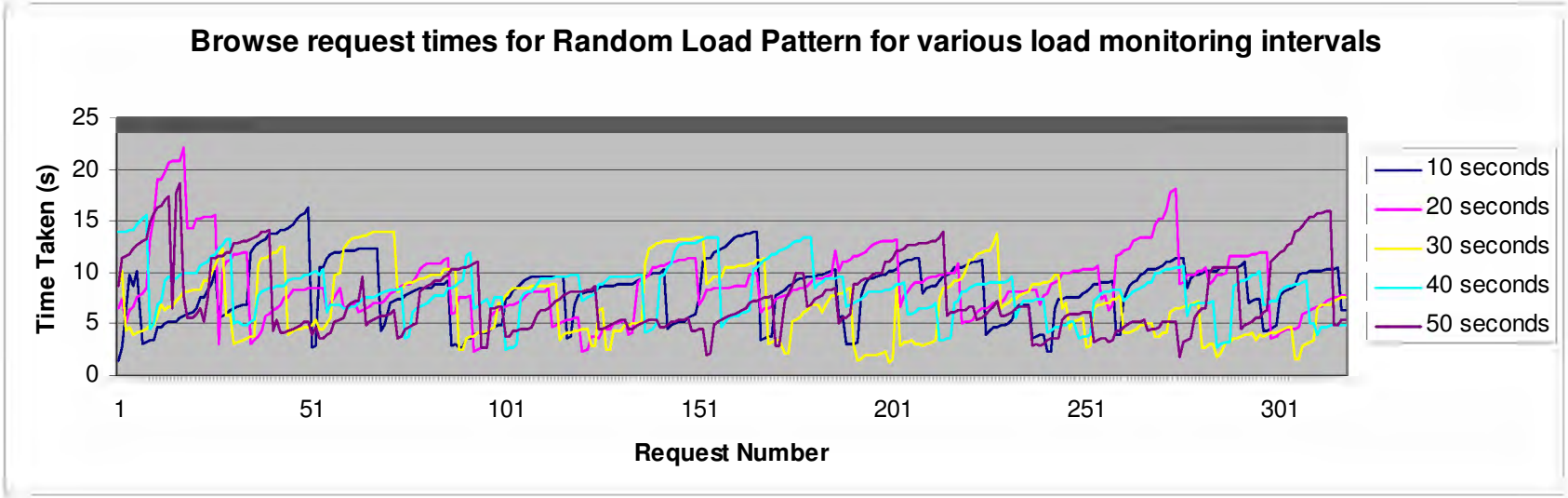

Figure 34: Browse interface response times for various load monitoring intervals using random load pattern.

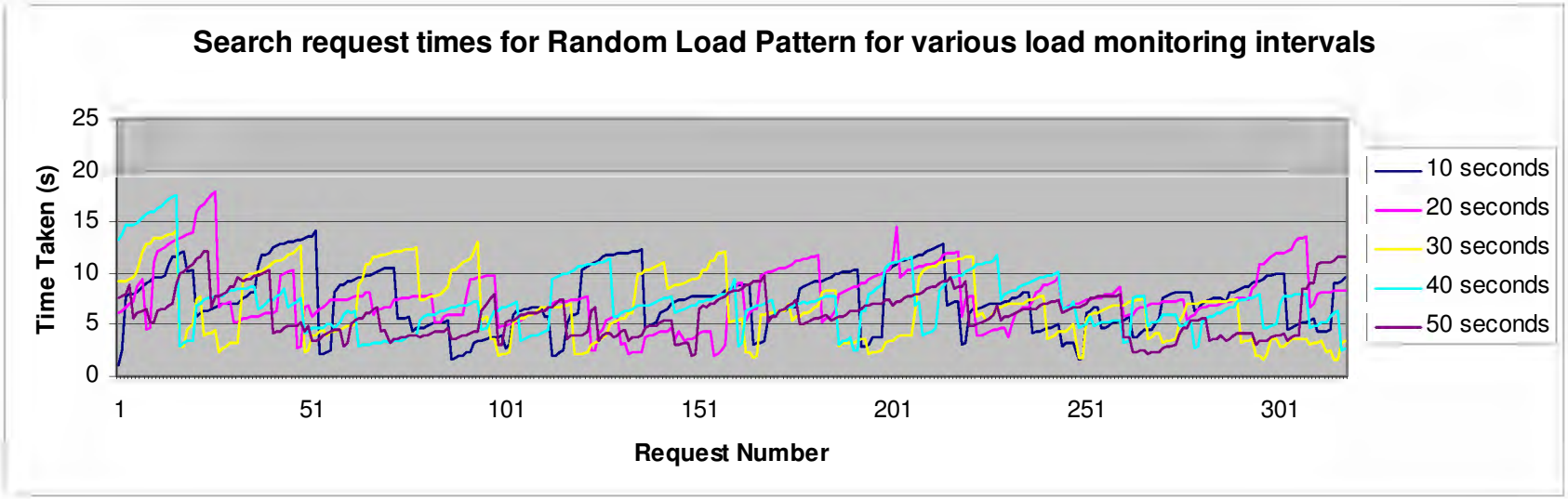

Figure 35: Search interface response times for various load monitoring intervals using random load pattern.

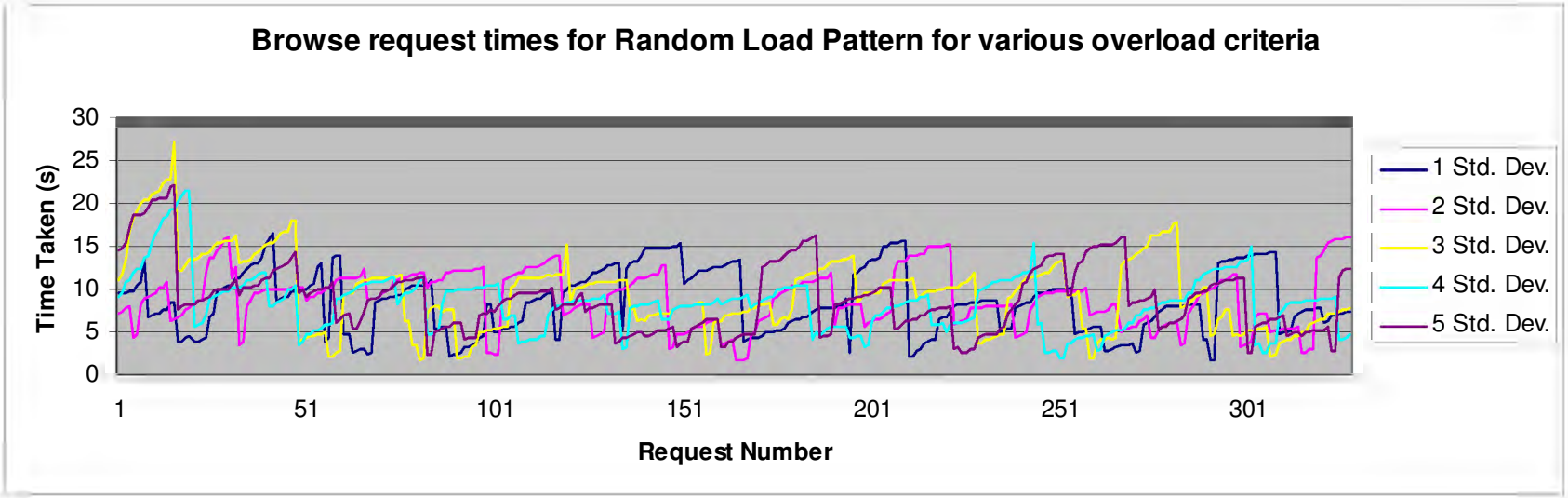

Figure 36: Browse interface response times for various overload criteria using random load pattern.

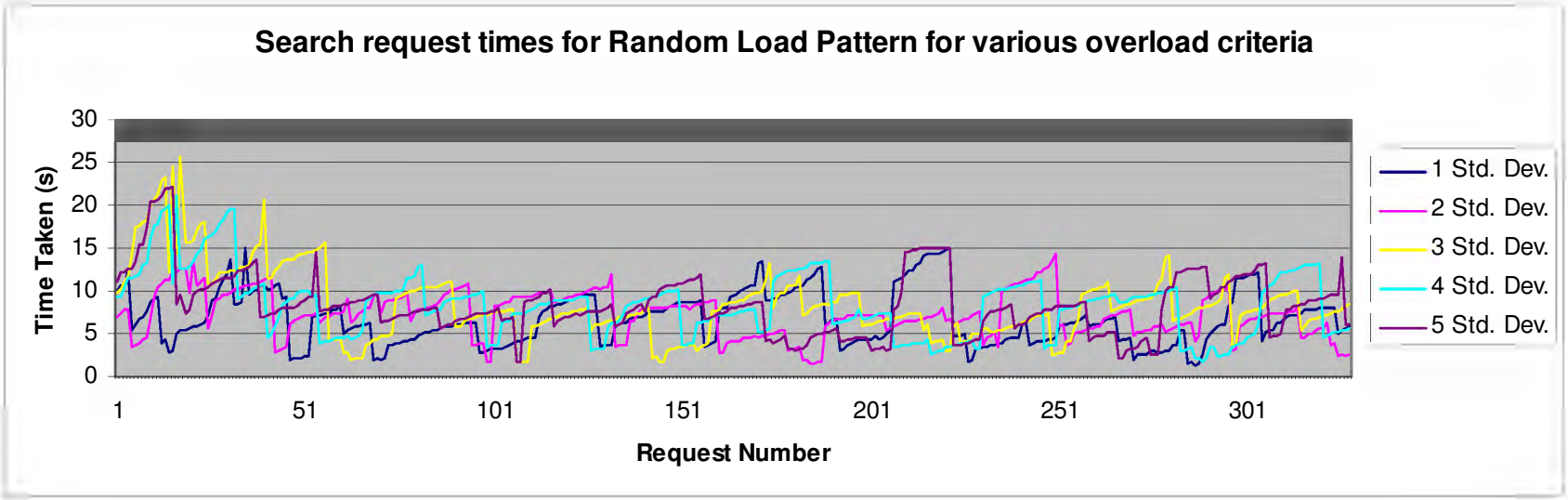

Figure 37: Search interface response times for various overload criteria using random load pattern.

The final load pattern used was one that was periodic or segmented and tested both interfaces simultaneously. In this test, the approach was to send a single request at a time for the first 50 requests (Phase 1) and then send 30 requests from 8 processes simultaneously (Phase 2). After the 30 requests have been completed, the test cycle was repeated once more. This resulted in a total number of requests of 594, made up of 297 requests per cycle. The graphs for the experimental and traditional system can be seen below (Figure 38 and Figure 39).

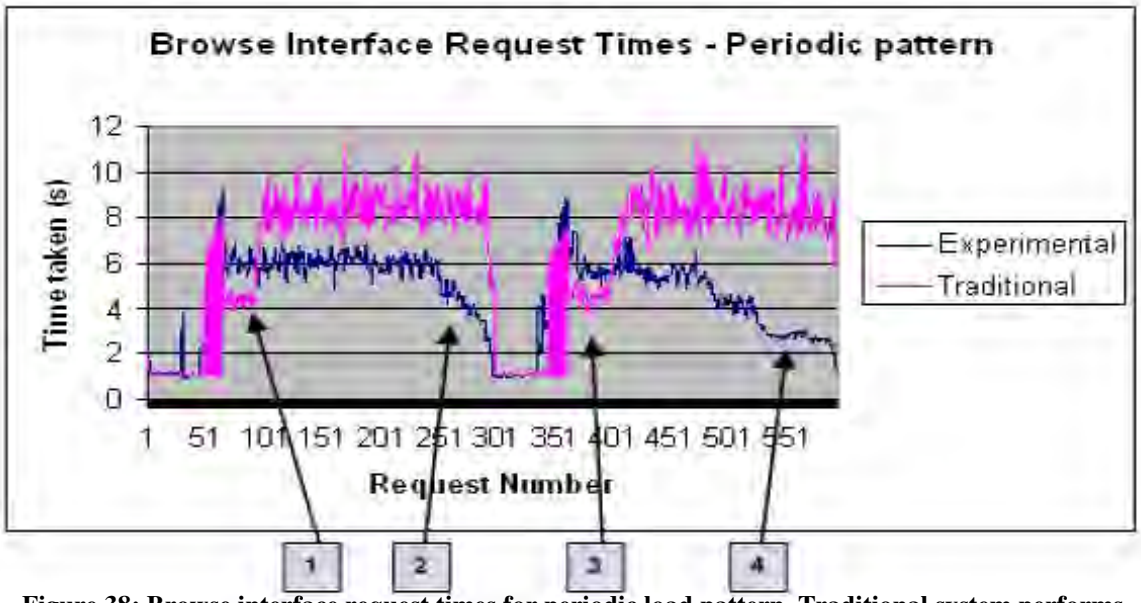

Figure 38: Browse interface request times for periodic load pattern. Traditional system performs better at markers 1 and 3 but as load increases (start of Phase 2), performance drops. Markers 2 and 4 indicate the drop in request time as number of requests drop (end of Phase 2).

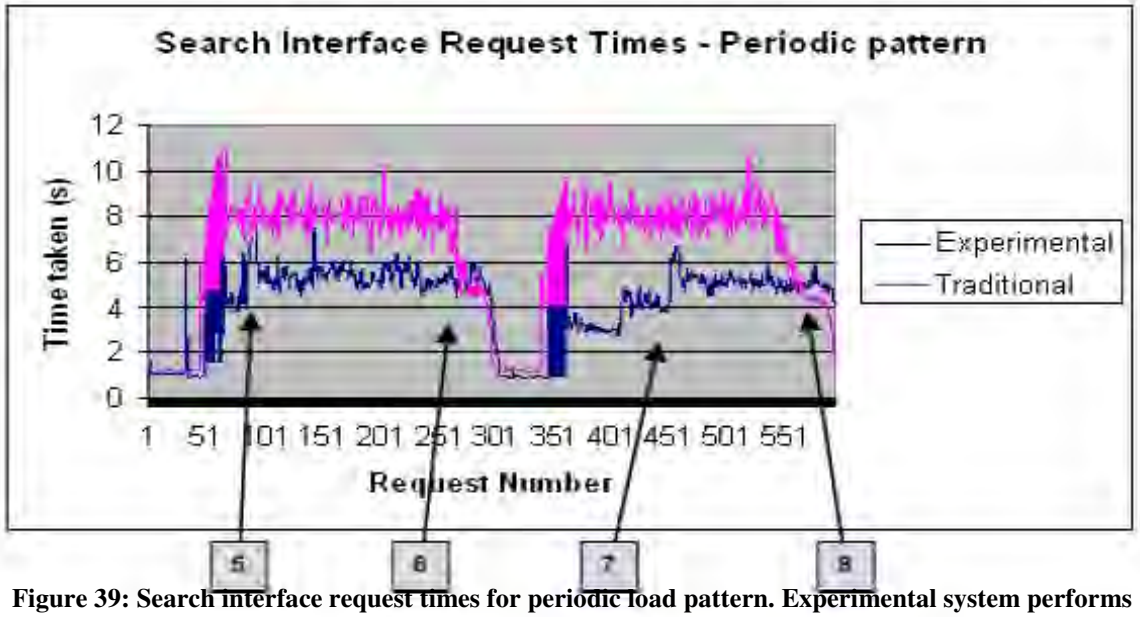

Figure 39: Search interface request times for periodic load pattern. Experimental system performs better as loads increases (5 and 7, start of Phase 2). Request time drops due to lower load at 6 and 8 (end of Phase 2).

For this load pattern, the experimental system performs better. Again this is largely due to the number of incoming requests to the systems. In both graphs (Figure 38 and Figure 39), we see the experimental system spiking at first (markers 1 and 5) and then developing a uniform response time. The decrease in requests at markers 2 and 6 occur as the first cycle completes.

As the search component is favoured by the load balancer, it completes each cycle faster than the browse component. This is highlighted in the results depicted in Figure 38. For the browse requests, we see a sharp decrease in the request time at marker 2. The drop in the browse requests are attributed to the fact that the search requests have already been completed for Phase 1 and Phase 2 for the first cycle. As they search completes its cycle faster more resources are available to be used. Conversely, at marker 7 for the experimental system, we see a sudden increase in request time for the search requests. This is attributed to the browse requests reaching Phase 2 of its second cycle and requiring more resources. For the browse requests at marker 3, we see a decrease in request time as the search requests are completing Phase 2 of the second cycle (marker 8). As the search cycles are complete, the browse components are free to use the released resources and hence we see the consistent decline in request time (marker 4) until the end of the second cycle.

The above test shows the experimental system adapting to the service requests being sent to the system. As a particular service requires more resources, the experimental system makes adjustments in an attempt to manage the new load requirements.

Varying the load monitoring interval and overload criterion was also performed for this load pattern. The results varying the load interval can be seen in Figure 40 and Figure 41 on the next page. The results for the overload criterion (Figure 42 and Figure 43) can be seen in on the page following the next one.

In the segmented load pattern we see that the shortest interval time performs the best. While it still creates spikes in the request times due to increased migrations and replications, its ability to react to the changes make it the best for this load pattern. Again, we see the increase in the search request time (markers 6 and 9) due to the browse starting to receive an increased number of requests. A decrease in the browse (markers 2 and 4) is seen as well when the search is receiving less requests. Both the load interval and overload criteria are a key factor in this load pattern. With long load intervals the system fails to react quickly enough and the request times increase. With the overload criteria set too have high or low we have the same problem.

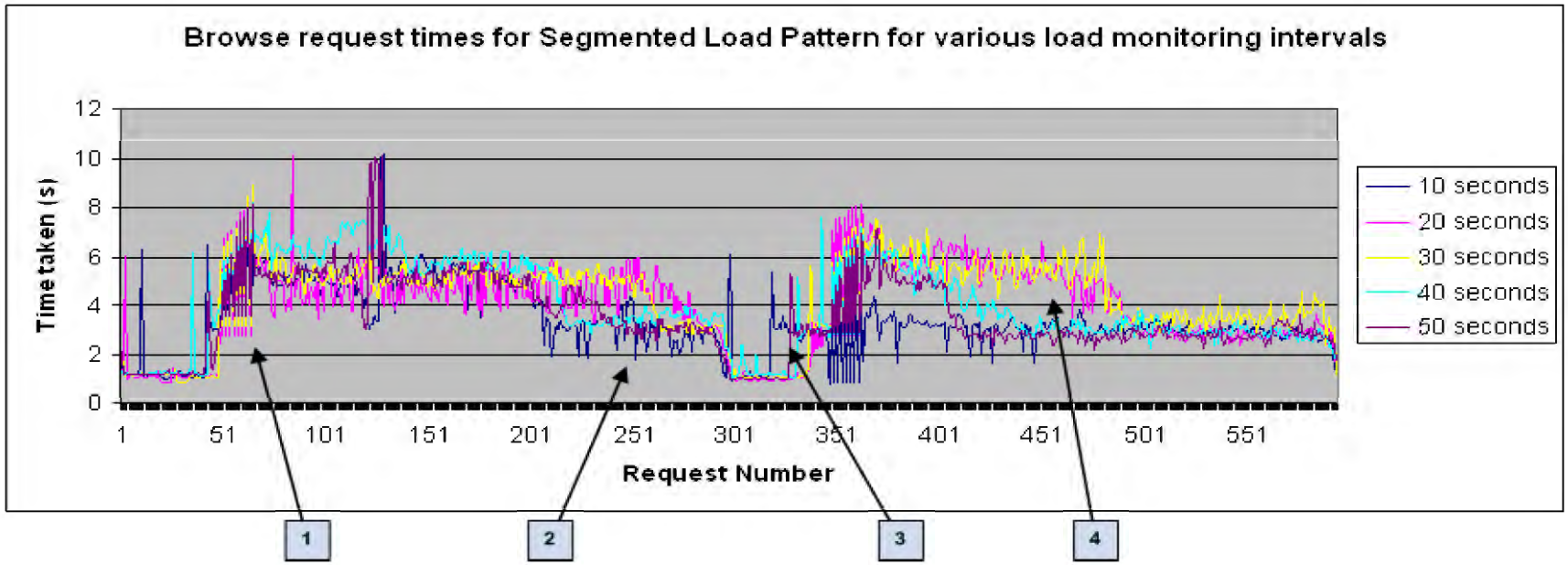

Figure 40: Browse interface response times for various load monitoring intervals for segmented load pattern. Browse request times increase as Phase 2 starts (1 and 3) and decrease due to search entering Phase 1 of the second cycle at 2 and 4. Shorter monitoring interval tends to work best.

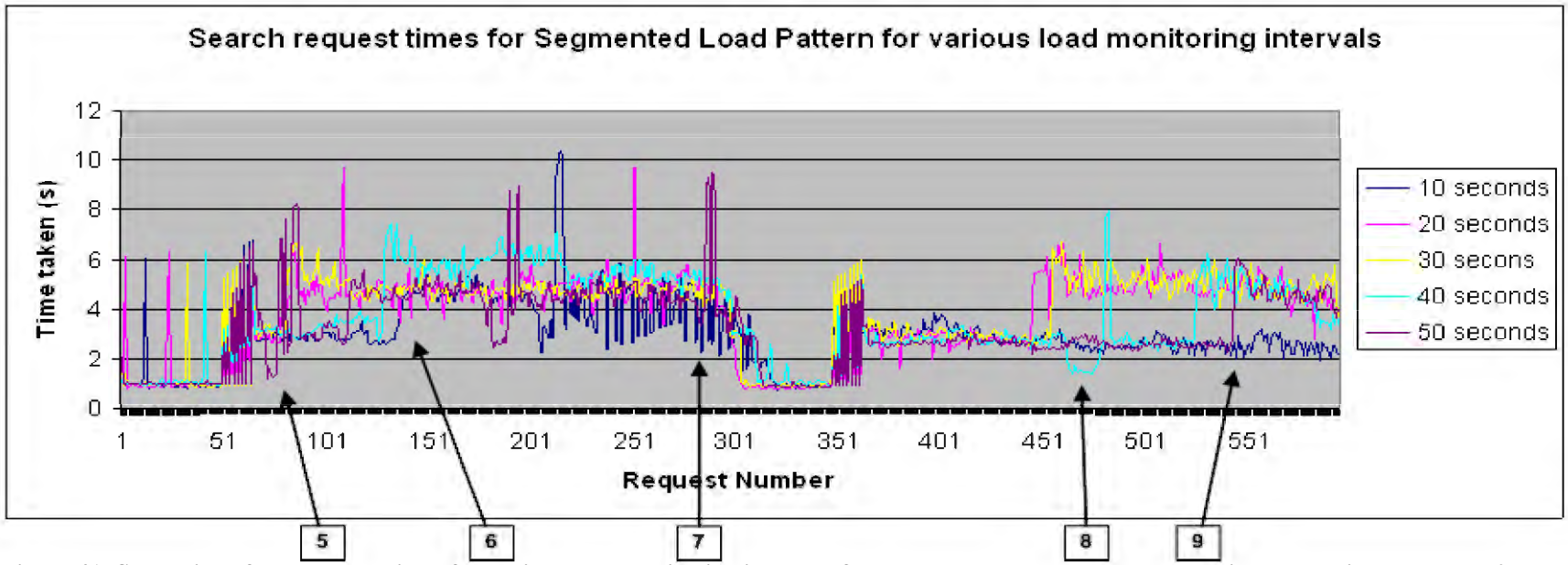

Figure 41: Search interface response times for various load monitoring intervals for segmented load pattern. Increase in request times as Phase 2 starts at 5 and at 6 as browse enters Phase 2 of the first cycle. Increase in request times at 8 and 9 as browse enters Phase 2 of the second cycle. Shorter monitoring interval tends to work best.

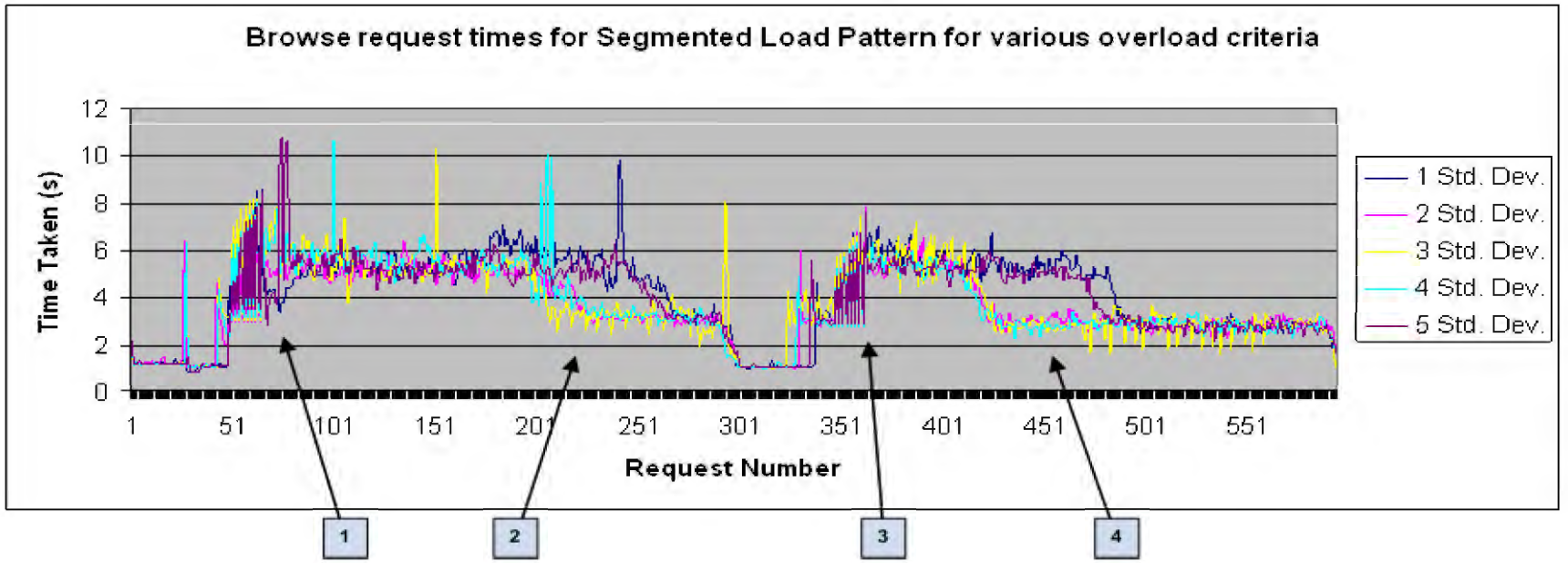

Figure 42: Browse interface response times for various overload criteria using periodic or segmented load pattern. Increase in response time due to Phase 2 starting at 1 and 3, while decreases at 2 and 4 are attributed to the search finishing Phase 2 early. Standard deviations of 3 and 4 tend to work best.

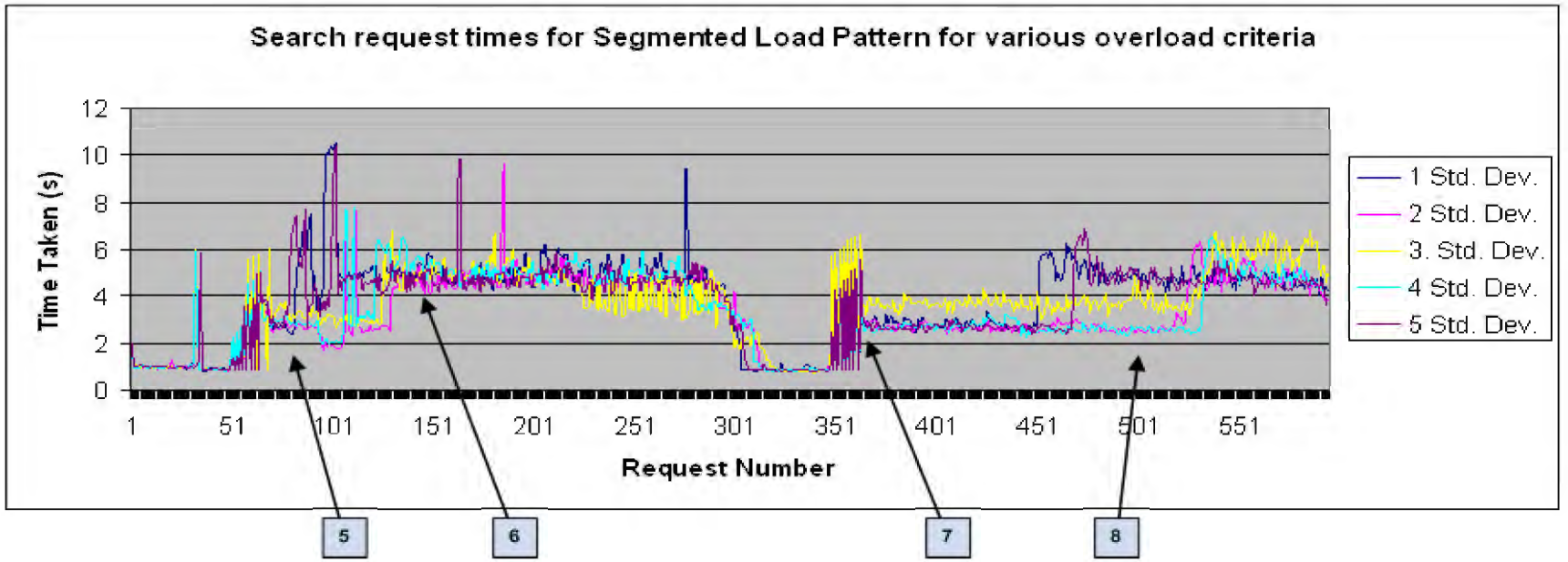

Figure 43: Search interface response times for various overload criteria using periodic or segmented load pattern. Increase in response time due to Phase 2 starting at 5 and 7, while increases at 6 and 8 are attributed to browse starting Phase 2 late. Standard deviations of 3 and 4 tend to work best.

4.4.4 Overheads Isolation Testing

The next phase of testing was to evaluate the network usage of the layers of the experimental system. As additional communication has been introduced, it is necessary to determine the amount of time it takes as well as the amount of extra information that is being sent. With this it can be determined whether or not the network is being used optimally and the system can be tweaked if necessary in order to improve performance. Communication between the Resolver, Registry, load monitors and migration subsystem was monitored. Each section that follows documents the component instances being monitored, the time taken as well as the number of bytes sent and received between the components.

4.4.4.1 Registry and Resolver Communication

Communication between component instances was not monitored as there is no change in the way the modified component instances communicate with one another. Also, each component communicates with the Resolver in order to get the URL to another component and never contacts the Registry. The network usage between the two can be seen below as well as the size of the data being transferred. This was monitored over a series of 400 requests. These tables show the communication between the Registry and Resolver when residing on the same node.

Component	Bytes sent	Bytes Received	Average Time	Request for
Search Interface	87	388	0.082	Search Engine URL
Browse Interface	91	400	0.080	Browse Engine URL
Search Engine	90	393	0.079	Archive URL
Browse Engine	90	393	0.079	Archive URL

Table 14: Time taken to send and receive data from component instance to Resolver

Component	Request from	Bytes sent	Bytes Received	Average Time	Request for
Resolver	Search Interface	42	388	0.030	Search Engine URL
Resolver	Browse Interface	45	393	0.029	Browse Engine URL
Resolver	Search Engine	45	393	0.031	Archive URL
Resolver	Browse Engine	45	393	0.029	Archive URL

Table 15: Time taken to send and receive data from Resolver to Registry

As can be seen from the results in the Table 14, the overall time used for networking for each component instance is quite small. In Table 15, we have the time it takes a component to get a URL from the Registry for each component respectively. The time for connecting an instance to the Resolver is just under 0.09 seconds and the time for the Resolver to connect to the Registry is 0.035 seconds on average. From this we can gauge that most of the time during communication may be spent in establishing connections

with the other component instances and also computation time for extracting the URL at the component level.

4.4.4.2 Central Load Monitor and Local Load Monitor communication

An additional contribution to the total time is made during the load monitoring phase. Each node is polled for its load and overloaded nodes are determined from this. Additional communication is then introduced when a node is deemed to be overloaded. This includes communication between the load monitors and the component instances residing on the overloaded node. The communications involved in each of the above-mentioned processes are shown below.

The average communication times between the local load monitors and the central load monitor is 2.113 seconds. The amount of data being sent and received is 67 bytes and 72 bytes respectively. This was done with the system processing queries as it normally would for 927 load queries.

The average time for the central load monitor to complete one load cycle (during a test), excluding waiting for migration and replication is 4.124124 seconds. This time was monitored over a series of 304 load cycles. A single load cycle consists of querying each node within the system for its load, calculating the average load and determining which nodes are overloaded.

4.4.4.3 Migration negotiation and communication

During migration communication between multiple component instances occurs. This communication includes the migration end point on the destination node, the Registry and Resolvers on each machine. The increase in load during migration is much larger than the other communications introduced into the system and therefore has a greater effect on request time. The communication between component instances and the data being sent can be seen in the following paragraphs. The following times for migration and replication were taken manually without a simulated load. These numbers therefore reflect the migration (or replication) times under ideal conditions.

The average time for a migration decision to be made by the local load monitor is 1.148763809 seconds. This average was derived from 86 migration decisions and includes communication with the components residing on the node for the number of requests they completed over the last load cycle. After the migration decision is made, it is the component instance's responsibility to complete the migration. The component then zips itself up and proceeds to migrate to the destination node. The average size of the migrating component instance is 277.0101 kilobytes and this takes an average of 0.789983 seconds to be unpacked and setup on the destination node. The migration or replication time varies from a maximum of 1.436515 seconds to a minimum of 0.379633 seconds based on the 86 recorded values and which component instance is being migrated.

4.5 Conclusions

From the test cases above, it is clear that high load conditions are better suited for the experimental system.

With the serial testing, the additional overheads of migration, replication, load monitoring and communication show a decrease in the response time. This is not a drastic reduction though and was as expected as extra tasks now need to be performed on each request. With the introduction of local resolution by the Resolver we obtain a speed quite close to that of the traditional system for individual request handling. Another major factor which has enabled the experimental system to perform this well is the use of remote database connections. Without having to migrate the data along with the component instance, the migration time has been greatly reduced. Migration or replication decisions are made in just less than 1.5 seconds, with the packaging, transferring and setting up of the component taking up to a maximum of 1.5 seconds. This totals about 3 seconds to complete a migration or replication.

The simultaneous testing highlighted that the ability to spread the load across multiple machines is of a great benefit. This allowed the experimental system to outperform the traditional one whether a single or multiple services were used. Its ability to adapt was highlighted in the load pattern testing as the system attempted to meet the needs of incoming requests as they changed over time. The pattern testing also provided us with input on the load monitor parameters.

Non-optimal parameter values can lead to poor performance which is worse than that of the traditional system. When monitoring intervals are too short, we see replications and migrations frequently which impacts performance negatively. On the other extreme, longer intervals cause the system to adapt slowly. This same sort of behavior is noted when the overload criteria is adjusted.

The use of SpeedyCGI to utilize the benefits of persistence is another factor. Having the process already in memory instead of having to be started up by the Web server reduces the total times drastically.

4.5.1 Summary

The experimental system was shown to perform as well as the traditional one under low load conditions. Under extreme load conditions though, the experimental one has proven to be superior. This is mainly due to its ability to migrate and replicate heavily loaded component instances. The additional communication introduced is small and the benefits provided have proven to outweigh it. This shows that a dynamic cluster-based system has better performance in many cases where it receives a high load of simultaneous requests, which is a realistic scenario.

Chapter 5

5. Conclusion and future work

Most digital libraries do not provide inherent support for scalability. The system presented in this dissertation focuses on building scalability into the DL system. Transparent migration and replication of DL components has enabled this goal.

The ODL components were used as the basis for the experimental framework but this can easily be implemented in other systems. It has been used as a proof-of-concept system. This chapter concludes the dissertation and documents the outcomes as well as possible future extensions for this type or research.

5.1 Outcomes

The experimental system shown in this dissertation incorporates scalability into its design. It allows the system to re-configure itself dynamically and autonomously without human intervention.

The system is composed of three main parts. The first part gave components added functionality to migrate and replicate across a group of computers. The second part is a Registry and Resolver system to track components with this additional functionality. The final part of the system is a load balancing subsystem which allowed the components to reconfigure themselves dynamically for improved performance.

The methodology used to evaluate the system is discussed in Chapter 4. It shows that certain conditions favour the system and that it performs acceptably under normal conditions.

The system has demonstrated that:

- it is flexible and adjust to users needs,
- is utilizes the migration and replication to meet these needs,
- it does not require human intervention to adjust and
- it performs acceptably under both normal and extreme load conditions.

5.2 Future Work

While the experimental system has proven to work better under certain conditions there is room for improvement in different aspects of the system. Future research for this system would be aimed at improving the performance, introducing better security and redundancy for key components. A discussion on the potential improvements is discussed in the rest of this section.

Load Balancing

The current system uses a simple load balancing algorithm and only uses the current load and number of component instances on the machines to make load decisions. This is a very limited view of the component instances and the system as a whole.

Firstly, future research could improve on this by using multiple factors to enhance load balancing decisions. Factors which could be used include CPU usage time, I/O time, number of databases accesses and number of components on the nodes. This would provide better information on how each component instance is performing as well as its usage. This would also allow complimentary component instances to be placed alongside one another. By complementary, we mean a component instance with large I/O time would be on the same node as a component instance with more CPU time. Factoring all of these into a load balancing decision would improve component placement as well as decisions as to which components should be migrated or replicated.

Secondly, database access and migration must be considered. In the current system, component instances use remote database access after they have been migrated or replicated. While in some cases this situation may be ideal it is not always the case. Therefore, a decision should be made on whether the database should be migrated or replicated along with the component instance. This would be ideal for nodes on which multiple databases reside since each remote connection would be contributing to its load. Alternatively, the database can be replicated at some other stage and some component instances could have the database queries redirected to it. This could improve the spread of the load across the nodes.

Thirdly, load pattern monitoring and pre-emptive migrations or replications must be considered. The current system does not use any of its past knowledge to predict future loads. This sort of system would prove vital in situations where certain batch jobs are run periodically or daily. Some sort of load monitoring agent could be used to determine patterns in the load and component instance usage. With this, it could then migrate and replicate component instances ahead of time in order to meet the upcoming request demands. Also, changing loads would be picked up by the agent, which would then adapt the system most appropriately in order to match the changes.

All the above-mentioned improvements incorporated into one load balancing system could enhance performance as well as how the component instances are placed as it adapts to load changes.

Registry Replication

As was noted in the design chapter, the Registry is a single point of failure and is something which needs to be addressed. The current solution is to use a failover Registry which comes online when the active one goes down.

As an addition to this, Registry replication could be introduced. When the failover Registry comes online, it should then be able to replicate itself. This would allow it to create another failover Registry entity which could be used if it goes down.

This would require additional changes to the components as it would need to be able to update the location of the Registry on demand. During this process, request times would most likely increase as requests will not be able to be routed correctly.

Security

No security has been developed for the migrating and replicating components. This aspect has not been focused on since the system is running on a cluster behind a secure firewall.

The initial solution to this problem was to tag each component instance with a hash of its contents. After migrating or replicating, the contents would be rehashed and compared to the hash sent along with the instance. The hashes should match, and if not, the instance has been tampered with and therefore the migration or replication would be re-initiated.

This basic solution can be substituted by simply encrypting the component instance on the sending node and then decrypting it on the receiving node. Since the component instances are quite small, the encryption should not increase the migration or replication time dramatically. The type of encryption used would be dependant on the component instances being used and the sensitivity of the data being accessed.

By adding encryption to the migration and replication schemes, this system could work in an open environment across multiple nodes in different places running different security levels. This would be similar to the way Globus [62] handles grid security.

Querying

Knowledge of multiple component instances allows the system to manage the number of requests sent to each component instance. This therefore improves the service time for requests. This information is not utilized to its full potential though.

This system can also be used to enhance querying. Queries could be split across multiple instances and the results merged together once received by the interface. This would involve the use of some sort of distributed inverted index and updates would be required to be incremental [63]. This would allow queries to be sent to multiple component instances thereby minimizing the amount of processing each component needs to do and possibly improve times when the system is under severe load stress.

Interface Proxying

Interface proxying allows the system to move the interface around instead of remaining static as it currently works in the system. The interface can thus be moved around and still be accessed normally. Moving the interface around provides the benefits described in the previous sections. An additional advantage exists in that the interface can moved to the engine it uses to complete queries. This could then improve performance under certain conditions.

Component Removal

With the current system, components are migrated and replicated across the cluster. This helps with spreading the load but there is currently no mechanism to remove components. If a component is replicated multiple times, it may exist in various places throughout the cluster. When the need for the component has slowed or stopped, it would be useful to be able to remove replicated instances. This would allow the node to reclaim the resources utilized by the component as well as lessen the amount of data the local Resolver, Registry and Load balancer need to track.

Summary

The experimental system exists as a proof of concept system and has proven to be successful under certain conditions. Enhancements that could be made include a better load balancing algorithm, Registry replication, migration and replication security, enhanced system querying, interface proxying and component removal. Each of these presents a new avenue of research for the type of system described in this thesis.

References

[1] Haujing, L., Councill, I.G., Bolelli, L., Zhou, D., Song, Y., Lee, W.C., Sivasubramaniam, A., Giles, C.L. 2006. CiteSeer$^{\chi}$: a scalable autonomous scientific digital library. *Proceedings of the 1st international conference on Scalable information systems*, Hong Kong, Vol. 152, No. 18. Retrieved: December 5, 2011, from http://portal.acm.org/dl.cfm

[2] Trnkoczy, J., Stankovski, V. 2008. Improving the performance of Federated Digital Library services. *Future Generation Computer Systems*, Netherlands, Amsterdam, Vol. 24, No. 8, pp. 824–832. Retrieved: December 6, 2011, from http://portal.acm.org/dl.cfm

[3] Witten, I.H., Boddie, S.J., Bainbridge, D., McNab, R.J. 2000. Greenstone: A Comprehensive Open-Source Digital Library Software System. *Proceedings of the fifth ACM conference on Digital libraries,* San Antonio, Texas, 113 – 121. Retrieved: June 14, 2005 from http://portal.acm.org/dl.cfm

[4] Carpenter, L. 2003. History and Development of the OAI-PMH, UKOLN, Bath. Retrieved: June 14, 2005, from http://www.oaforum.org/tutorial/english/page2.htm

[5] Lagoze, C., Van de Sompel, H. 2001. The Open Archives Initiative: Building a low barrier interoperability framework. *Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries,* 54 – 62. Retrieved: June 14, 2005, from http://www.openarchives.org/documents/jcdl2001-oai.pdf

[6] Suleman, H. 2002. Open Digital Libraries, PhD Thesis. Virginia Tech. Retrieved: June 14, 2005, from http://www.husseinsspace.com/publications/odl.pdf

[7] Suleman, H., Fox, E.A., Kelapure, R., Krowne, A., Luo, M. 2003. Building digital libraries from simple building blocks. *Online Information Review*, Vol. 27, No. 5, pp. 301-310. Retrieved: June 15, 2005, from http://pubs.cs.uct.ac.za/archive/00000013/01/oir_2003_oaiodl_revised2.pdf

[8] Castelli, D., Pagano, P. 2003. A system for building expandable digital libraries. *Proceedings of the 3rd ACM/IEEE-CS Joint Conference on Digital libraries*, Houston, Texas, 335 – 345. Retrieved: June 15, 2005, from http://portal.acm.org/dl.cfm

[9] MIT Libraries, Hewlett-Packard Company. 2005. Introducing DSpace: DSpace Federation. Retrieved: July 5, 2005, from http://dspace.org/introduction/index.html

[10] Tansley, R., Bass, M., Stuve, D., Branschofsky, M., Chudnov, D., McClellan, G., Smith, M. 2003. The DSpace institutional digital repository system: current functionality. *Proceedings of the 3rd ACM/IEEE-CS Joint Conference on Digital libraries,* Houston, Texas, 87-97. Retrieved: June 15, 2005, from http://portal.acm.org/dl.cfm

[11] Tansley, R., Bass, M., Stuve, D., Branschofsky, M., Chudnov, D., Breton, P., Carmichael, P., Cattey, B., Ng, J. 2002. DSpace – A Sustainable Solution for Institutional Digital Asset Services. Retrieved: June 15, 2005, from http://www.dspace.org/technology/architecture.pdf

[12] Don, K., Buchanan, G., Witten, I.H. (n.d.). Greenstone 3: A modular digital library. Retrieved: February 13, 2007, from http://www.greenstone.org/docs/greenstone3/manual.pdf

[13] Fedora Project. (n.d.). Fedora. Retrieved: November 24, 2007, from http://www.fedora.info/about/history.shtml

[14] Daniel, R., Lagoze, C., Payette, S.D. 1998. A Metadata Architecture for Digital Libraries. *Proceedings of the IEEE Forum on Research and Technology Advances in Digital Libraries*. Retrieved: November 24, 2007, from http://www.cs.cornell.edu/lagoze/papers/ADL98/dar-adl.html

[15] Fedora Project. 2007. Overview: The Fedora Digital Object Model. Retrieved: November 25, 2007, from http://www.fedora.info/download/2.2.1/userdocs/digitalobjects/objectModel.html

[16] Fedora Project. (n.d.). Fedora Service Framework. Retrieved: November 25, 2007, from http://www.fedora.info/download/2.2.1/userdocs/server/features/serviceframework.htm

[17] Bradley, K., Lei, J., Blackall, C. 2007. Towards an Open Source Repository and Preservation System, *D-Lib Magazine*, Vol. 13, No. 5. Retrieved: December 7, 2011, from http://portal.unesco.org/ci/en/files/24700/11824297751towards_open_source_repository.doc/towards_open_source_repository.doc.

[18] The National Science Digital Library. (n.d.). NSDL.org - About NSDL - The National Science Digital Library. Retrieved: November 25, 2007, from http://nsdl.org/about/

[19] The National Science Digital Library. 2005. NSDL Library Architecture: An Overview. Retrieved: 25 November, 2007, from nsdl.comm.nsdl.org/docs/nsdl_arch_overview.pdf

[20] Fulker, D., Janée, G. 2002. Components of an NSDL Architecture: Technical Scope and Functional Model, Retrieved: December 6, 2011 from http://www.alexandria.ucsb.edu/~gjanee/archive/2002/fulker-janee-paper.pdf

[21] Sumner, T., Giersch, S., Jones, C. 2003. Steps Towards Establishing Shared Evaluation Goals and Procedures in the National Science Digital Library. Proceedings of the 3rd ACM/IEEE-CS joint conference on Digital libraries, Houston, Texas, pp. 407. Retrieved: December 6, 2011, from http://portal.acm.org/dl.cfm

[22] Feijen, M., Horstmann, W., Manghi, P., Robinson, M., Russel, R. 2007. Driver: Building the Network for Accessing Digital Repositories across Europe. *Ariadne*, Vol. 2007, No. 53. Retrieved: December 6, 2011, from http://www.ariadne.ac.uk/issue53/feijen-et-al/

[23] Lossau, N., Peters, D. 2008. DRIVER: Building a Sustainable Infrastructure of European Scientific Repositories. *Liber Quarterly: The Journal of European Research Libraries*, Vol. 18, No.3, pp. 437 – 448. Retrieved: December 6, 2011, from http://www.doaj.org/doaj

[24] Lossau, N., Peters, D. 2010. DRIVER: building a sustainable infrastructure for global repositories. *The Electronic Library*, Vol. 29, No. 2, pp. 249 – 260. Retrieved: December 6, 2011, from http://www.emeraldinsight.com/journals.htm

[25] Manghi, P., Candela, L., Castelli, D., Mikulicic, M., Pagano P. D-Net Software Toolkit: Realizing and Maintaining Sustainable Aggregative Digital Library Systems. *D-Lib Magazine*, Vol. 6, No. 3. Retrieved: December 6, 2011, from http://www.dlib.org/dlib/march10/manghi/03manghi.html

[26] D-NET LAB. (n.d.). System Architecture. Retrieved: December 6, 2011, from http://www.d-net.research-infrastructures.eu/node/5

[27] Diligent Project. 2008. Home – diligentproject.org. Retrieved: December 6, 2011, from http://diligent.ercim.eu/

[28] Candela, L., Akal, F., Avancini, H., Castelli, D., Fusco, L., Guidetti, V., Langguth, C., Manzi, A., Pagano, P., Schuldt, H., Simi, M., Springmann, M., Voicu, L. 2007. DILIGENT: integrating digital library and Grid technologies for a new Earth observation research infrastructure. *International Journal of Digital Libraries*, Vol. 7, No. 1 – 2, pp. 59 – 80. Retrieved: December 7, 2011, from http://portal.acm.org/dl.cfm

[29] Simeoni, F., Crestani, F., Bierig, R. 2007. The DILIGENT Framework for Distributed Information Retrieval. *Proceedings of the 30th annual international*

ACM SIGIR conference on Research and development in information retrieval, Amsterdam, Netherlands, pp. 781 – 782. Retrieved: December 6, 2011, from http://portal.acm.org/dl.cfm

[30] Diligent Project. 2008. Diligent releases the gCube system version 1.0 - diligentproject.org. Retrieved: December 6, 2011, from http://diligent.ercim.eu/content/view/199/235/

[31] Casanova, H. 2002. Distributed Computing Research Issues in Grid Computing. *ACM SIGACT News*, Vol. 33, No. 2, pp. 50 – 70. Retrieved: December 7, 2011, from http://portal.acm.org/dl.cfm

[32] Jacob, B., Brown, M., Fukui, K., Trivedi, N. 2005. Introduction to Grid Computing. [pdf] USA: Redbooks. Available at: https://www.redbooks.ibm.com/redbooks/pdfs/sg246778.pdf

[33] Meeker, R.D. 2005. Comparative system performance for a Beowulf cluster. Journal of Computing Sciences in Colleges, Vol. 21, No.2, pp. 114 – 119. Retrieved: December 7, 2011, from http://portal.acm.org/dl.cfm

[34] Baker, M., Buyya, R., Hawick, K., James, H., Hai, J. 2000. Cluster Computing R&D in Australia. Technical Report, ATIP. Retrieved: June 15, 2005, from http://www.buyya.com/papers/ClusterComputingAU.pdf

[35] Feeley, M., Miller, J.S. 1990. A parallel virtual machine for efficient scheme compilation. *Proceedings of the 1990 ACM conference on LISP and functional programming,* Nice, France, pp. 119 – 130. Retrieved: November 23, 2007, from http://portal.acm.org/dl.cfm

[36] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderman, V. 1994. PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing. Massachusetts: The MIT Press.

[37] Gray, P., Joiner, D., Murphey, T., Neeman, H., Peck, C. 2004. Parallel and Cluster Computing. PowerPoint slides in PDF format. Retrieved: June 24, 2005, from http://www.oscer.ou.edu/NCSIOU2004/ncsiou2004_mpi.pdf

[38] The MPI Forum. 1993. MPI: a message passing interface. *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Portland, Oregon, pp. 878 - 883 Retrieved: June 23, 2005, from http://portal.acm.org/dl.cfm

[39] ClusterMonkey. 2005. Processes, Processors, and MPI, Oh My! Retrieved: June 24, 2005, from http://www.clustermonkey.net//content/view/26/28/

[40] MPI Program Structure. (n.d.). PowerPoint Slides. Retrieved: June 24, 2005, from www.sci.hkbu.edu.hk/tdgc/tutorial/ParallelProgrammingWithMPI/03-MPIProgramStructure.ppt

[41] Open Cluster Group. 2005. OSCAR | Open Source Cluster Application Resources. Retrieved: June 24, 2005, from http://oscar.openclustergroup.org/

[42] Mattson, T. 2001. OSCAR: A packaged cluster software stack for high performance computing. Technical Report, Open Cluster Group. Retrieved: June 24, 2005, from http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/PDF01/Mattson.pdf

[43] Bar, M. 2005. openMosix, an Open Source Linux Cluster Project. Retrieved: June 23, 2005, from http://openmosix.sourceforge.net/

[44] Bar, M., MAASK. 2003. OpenMosix. Retrieved: June 27, 2005, from http://openmosix.sourceforge.net/linux-kongress_2003_openMosix.pdf

[45] Franklin, S., Graesser, A. 1996. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages,* 21 – 35. Retrieved: May 27, 2005, from http://portal.acm.org/dl.cfm

[46] Terveen, l.G., Murray, L.T. 1996. Helping users program their personal agents. *Proceedings of the SIGCHI conference on Human factors in computing systems: common ground*, Vancouver, British Columbia, 355 – 361. Retrieved: November 24, 2007, from http://portal.acm.org/dl.cfm

[47] BotKnowledge. 2006. BotKnowledge. Retrieved: June 27, 2005, from http://www.botknowledge.com/

[48] Lange, D.B., Oshima, M. 1999. Seven good reasons for mobile agents. *Communications of the ACM*, Vol. 42, No.3, pp. 88 – 89. Retrieved: May 28, 2005, from http://portal.acm.org/dl.cfm

[49] Claessens, J., Preneel, B., Vandewalle, J. 2003. (How) can mobile agents do secure electronic transactions on untrusted hosts? A survey of the security issues and the current solutions. *ACM Transactions on Internet Technology*, Volume 3 No. 1, pp. 28 – 48. Retrieved: December 7, 2011, from http://portal.acm.org/dl.cfm

[50] Kun, Y., Xin, G., Dayou, L. 2000. Security in Mobile Agents: problems and approaches. *ACM SIGOPS Operating Systems Review*, Vol. 34, No. 1, pp. 21 – 28. Retrieved: December 7, 2011, from http://portal.acm.org/dl.cfm

[51] Network Working Group. 1987. RFC 1035 Domain Names – Implementation and Specification. Retrieved: June 21, 2005, from http://www.rfc-editor.org/rfc/rfc1035.txt

[52] HowStuffWorks, Inc. (n.d.). Howstuffworks "How Domain Name Servers Work". Retrieved: June 21, 2005, from http://computer.howstuffworks.com/dns.htm/printable

[53] Network Working Group. 1987. RFC 1034 Domain Names – Concepts and Facilities. Retrieved: June 21, 2005, from http://rfc.dotsrc.org/rfc/rfc1034.html

[54] Mehaffey, J. (n.d.). How to setup a Dynamic DNS service on dyndns.org. Retrieved: June 21, 2005, from http://www.gpsinformation.org/joe/dynamicdns.html

[55] Berinato, S. 2002. The DNS Attack: A Success Story for the Good Guys, CSO Magazine. Retrieved: June 21, 2005, from http://www.csoonline.com/read/120902/briefing_dns.html

[56] Shafer, K., Weibel, S., Jul, E., Fausey, J., OCLC Online Computer Library Centre, Inc. (n.d.). Introduction to PURLs. Retrieved: June 24, 2005, from http://purl.oclc.org/docs/inet96.html

[57] Goerwitz III, R.L. (n.d.) OpenURLs, Citations and Two-Level SRV – Record Based Resolution. Retrieved: July 1, 2005, from http://goerwitz.com/papers/ucla/

[58] Bellwood, T.H., IBM developerWorks. 2001. UDDI – A Foundation for Web Services. Retrieved: July 13, 2005, from http://www.idealliance.org/papers/xml2001/papers/pdf/03-02-03.pdf

[59] OASIS. 2004. Introduction to UDDI: Important Features and Functional Concepts. Organization for the Advancement of Structured Information Standards. Retrieved: July 13, 2005, from http://uddi.org/pubs/uddi-tech-wp.pdf

[60] Lagoze, C., Van de Sompel, H., Nelson, M., Warner, S. 2008. The Open Archives Initiative Protocol for Metadata Harvesting. Retrieved: February 13, 2007, from http://www.openarchives.org/OAI/openarchivesprotocol.html#Identify

[61] Jupitermedia Corporation. 2000. WebRef Update: Featured Article: Persistent Perl on the Virtual Host. Retrieved: February 13, 3007, from http://www.webreference.com/new/speedycgi.html

[62] Kanaskar, N.V., Topaloglu, U., Bayrak, C. 2005. Globus security model for grid environment. ACM SIGSOFT Software Engineering Notes, 30(6). Retrieved: November 24, 2007, from http://portal.acm.org/dl.cfm

[63] Sornil, O. 2001. Parallel Inverted Indices for Large-Scale, Dynamic Digital Libraries, PhD. Virginia Tech. Retrieved: February 13, 2007, from http://scholar.lib.vt.edu/theses/available/etd-02062001-114915/unrestricted/dissertation.pdf

Appendix A

Simple Perl Test Script

A simple version of the Perl script used to simulate requests for testing.

```
#!/usr/bin/perl

use Pure::EZHTTP;
use Time::HiRes qw(gettimeofday tv_interval);


my $temp = “http://machine1.cluster.aim/~momar/cgi-
bin/FinalTest/UI/search.pl?query=computer”;

my $max = 50;

for( my $i=0; $i<$max; $i++)
{
        my $t0 = [gettimeofday];
        my $request = new Pure::EZHTTP GET => $temp;
        my $resp = $agent->request($request);
        my $ret = "";
        $ret .= $resp->code;
        $elapsed = tv_interval ($t0, [gettimeofday]);

        open(HAND,”>>SearchSerialTimes”);
        flock(HAND,2);
        print $elapsed.”\n”;
        close(HAND);
}
```

Appendix B

Graphs for the different layouts used in serial testing.

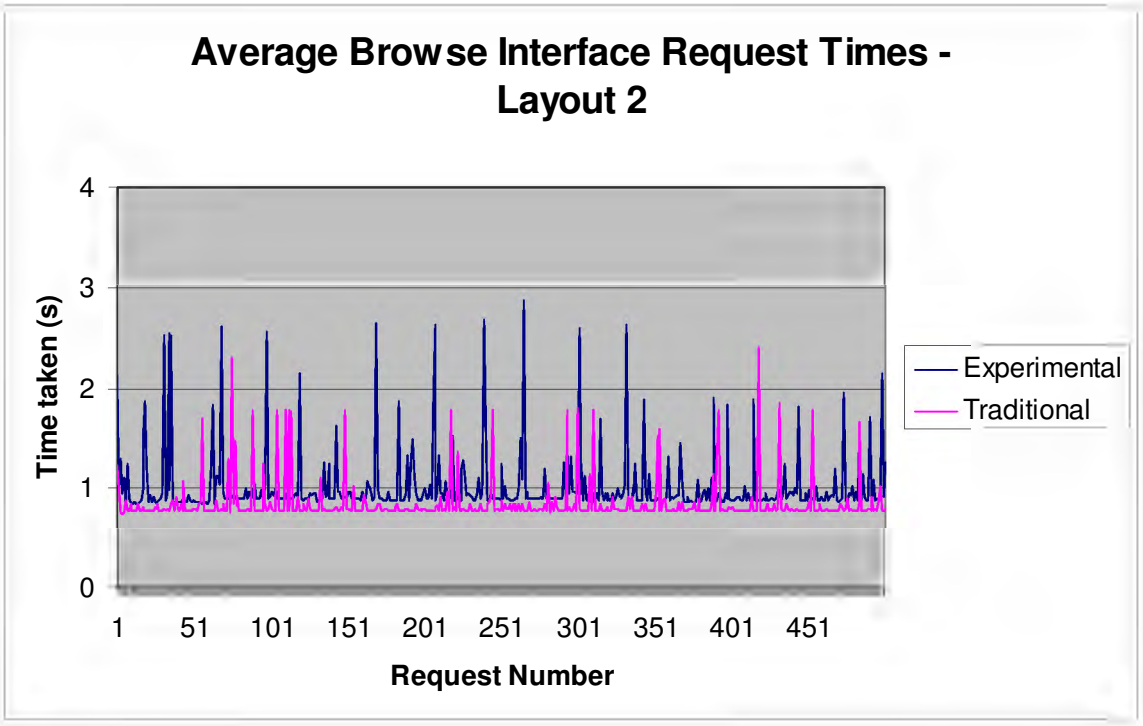

Figure 44: Browse request times for 500 serial requests.

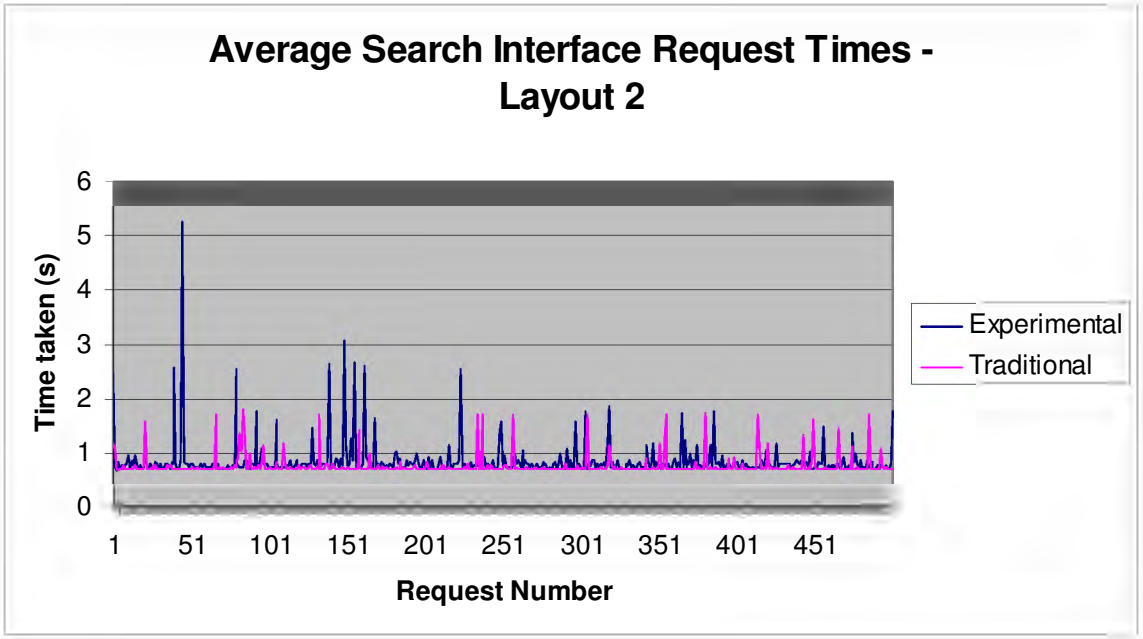

Figure 45: Search request times for 500 serial requests.

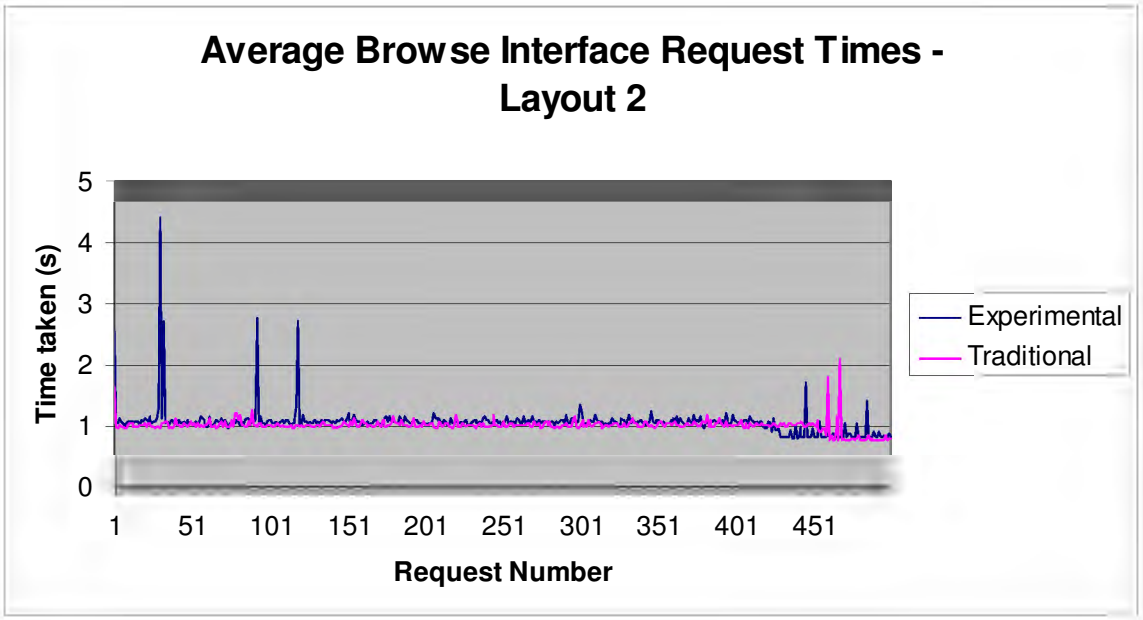

Figure 46: Browse request time for 500 serial requests while search is being tested.

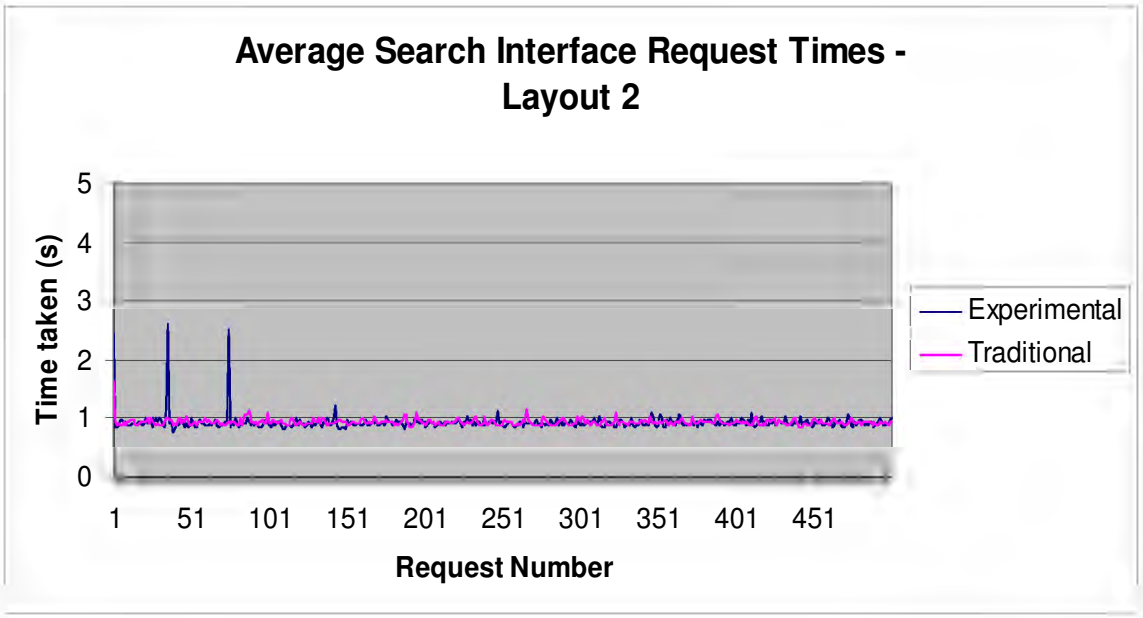

Figure 47: Search request time for 500 serial requests while browse is being tested.

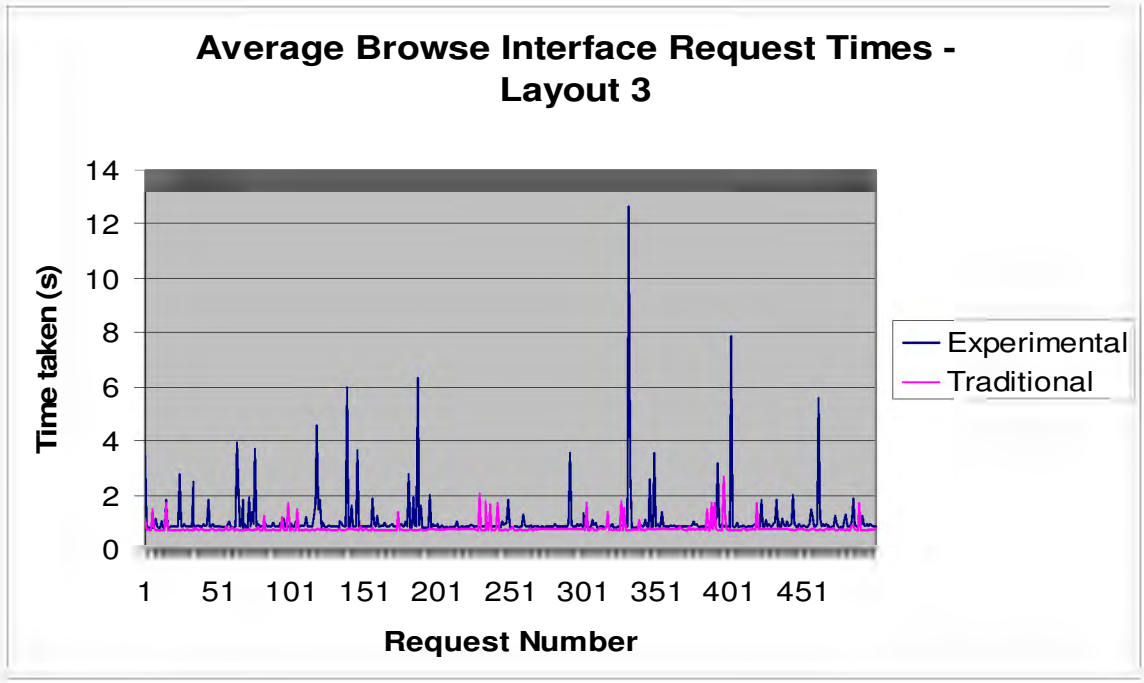

Figure 48: Browse request times for 500 serial requests.

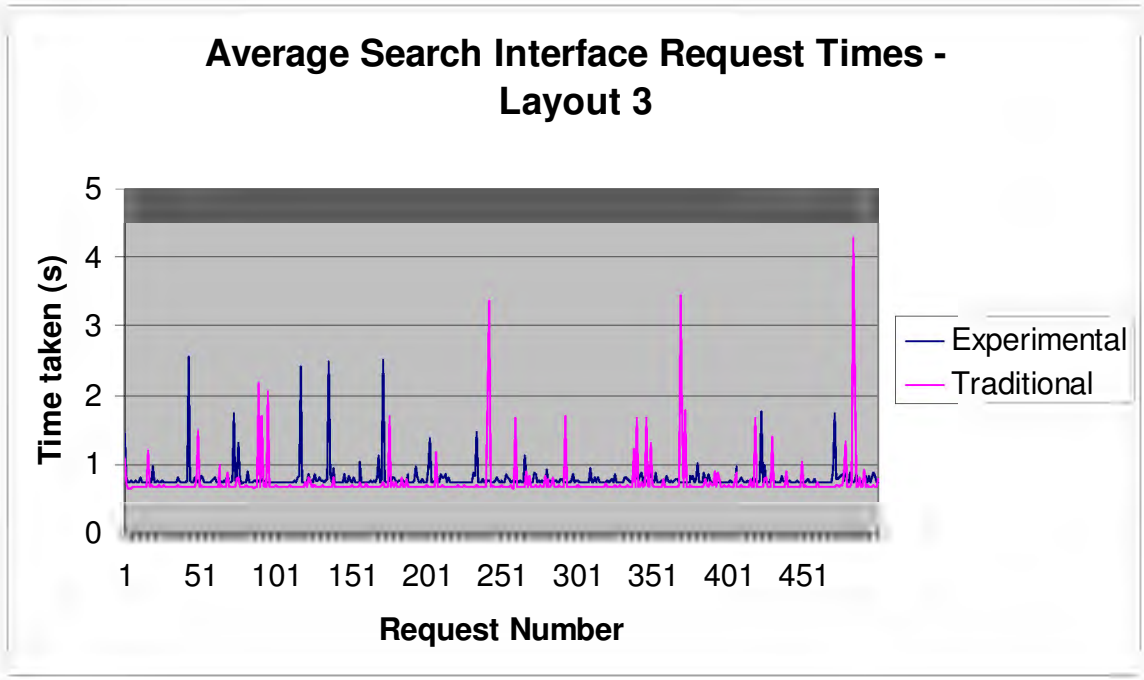

Figure 49: Search request times for 500 serial requests.

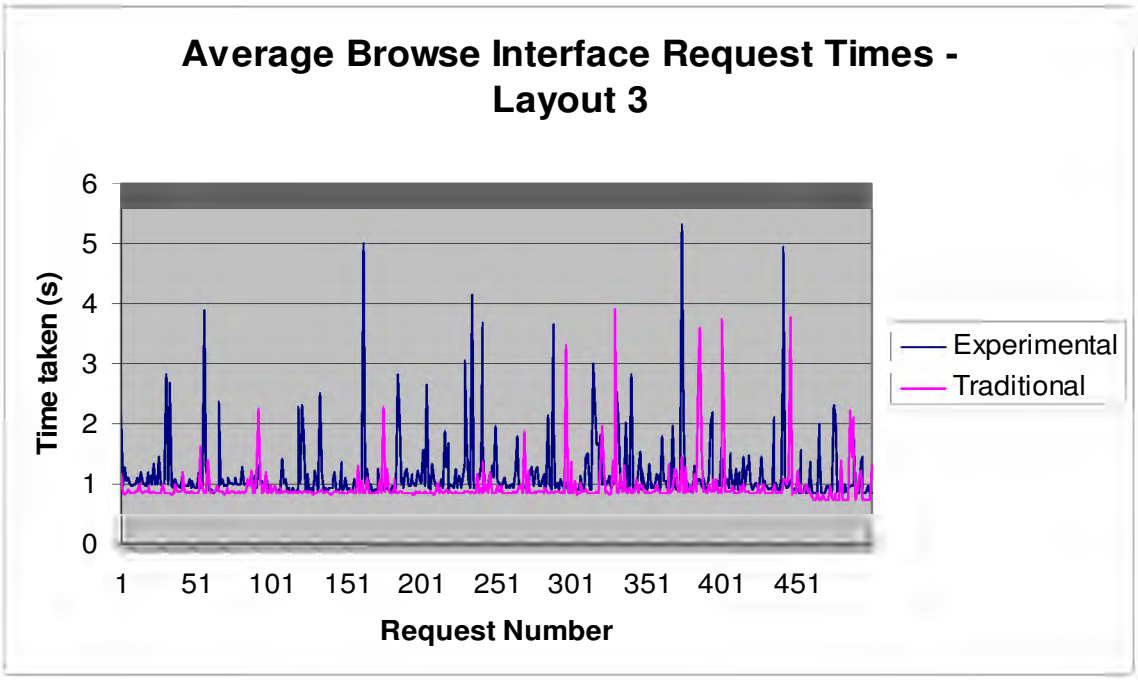

Figure 50: Browse request time for 500 serial requests while search is being tested.

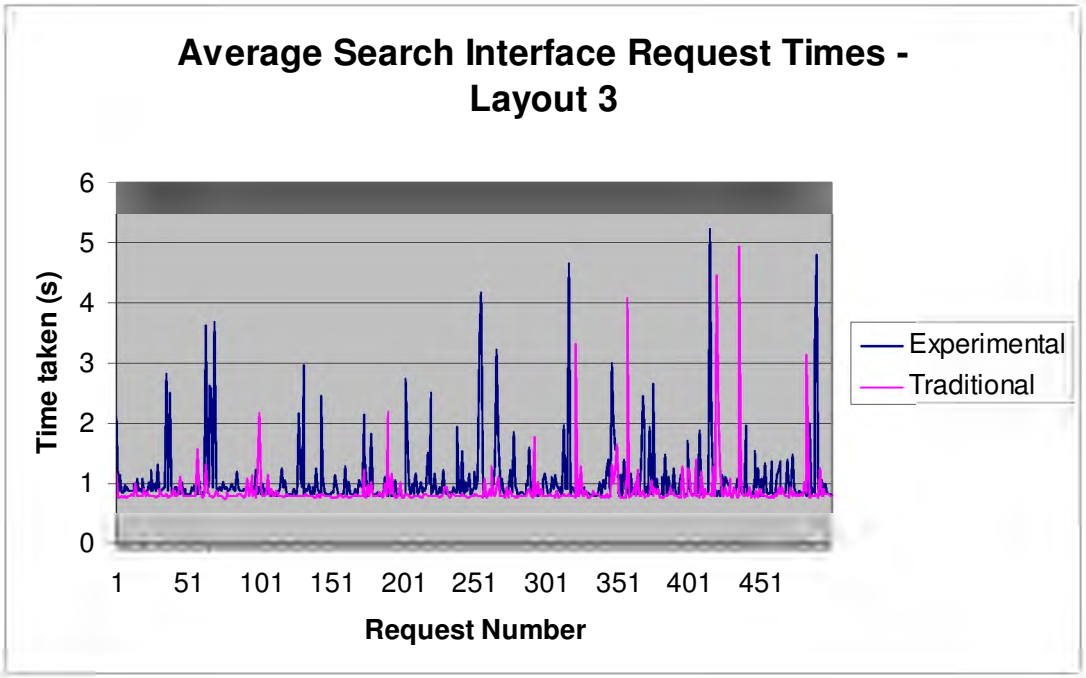

Figure 51: Search request time for 500 serial requests while browse is being tested.

Appendix C

The following tables show the status of the experimental system in terms of where the component instances have migrated and replicated to from its initial starting point.

Registry Tables - Layout 1

Search Run

Run 1		Node 1	Node 2	Node 3
Initial	Browse	1	0	0
Initial	Search	1	0	0
Initial	Archive	1	0	0
Final	Browse	1	0	0
Final	Search	0	1	0
Final	Archive	0	2	1

Registry Tables – Layout 2

Browse Run

Run 1		Node 1	Node 2	Node 3
Initial	Browse	0	1	0
Initial	Search	0	1	0
Initial	Archive	0	1	0
Final	Browse	1	0	0
Final	Search	0	0	1
Final	Archive	2	1	0

Search

Run 1		Node 1	Node 2	Node 3
Initial	Browse	0	1	0
Initial	Search	0	1	0
Initial	Archive	0	1	0
Final	Browse	0	0	1
Final	Search	1	0	0
Final	Archive	2	1	0

Registry Tables – Layout 3

Browse

Run 1		Node 1	Node 2	Node 3
Initial	Browse	0	0	1
Initial	Search	0	1	0
Initial	Archive	1	0	0
Final	Browse	2	1	0
Final	Search	0	1	0
Final	Archive	1	1	0

Search

Run 1		Node 1	Node 2	Node 3
Initial	Browse	0	0	1
Initial	Search	0	1	0
Initial	Archive	1	0	0
Final	Browse	0	0	1
Final	Search	2	0	2
Final	Archive	1	1	0

Appendix D

Graphs (with trend lines) depicting the response times for various load monitor intervals for random load pattern.

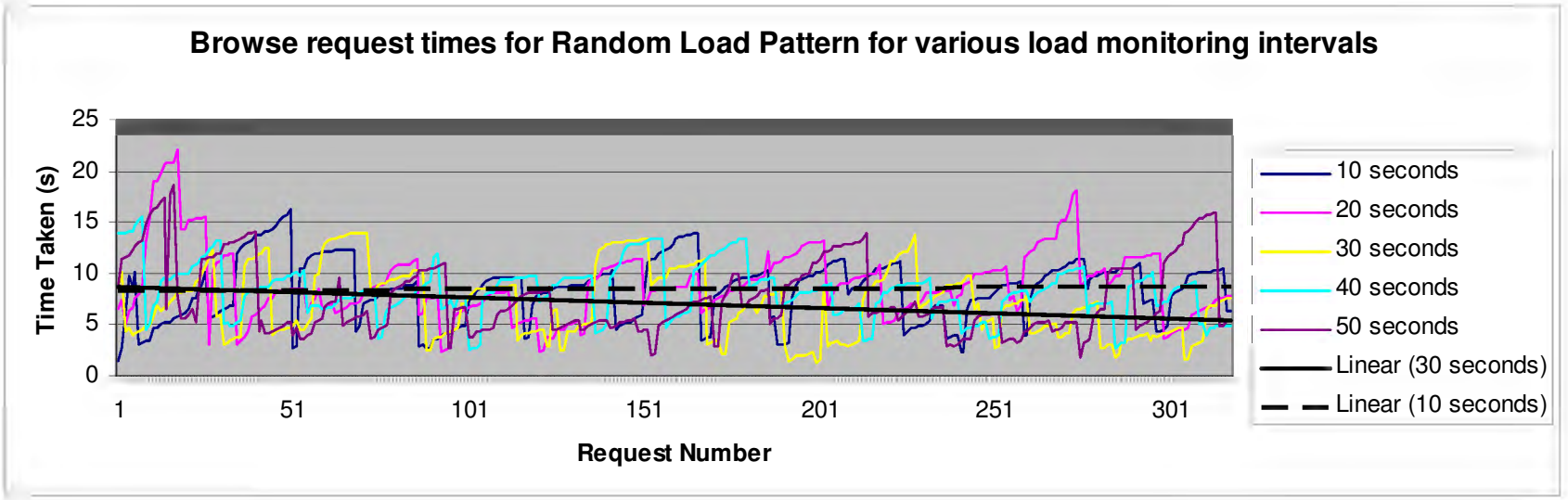

Figure 52: Browse interface response times for various load monitoring intervals using random load pattern with linear trend lines

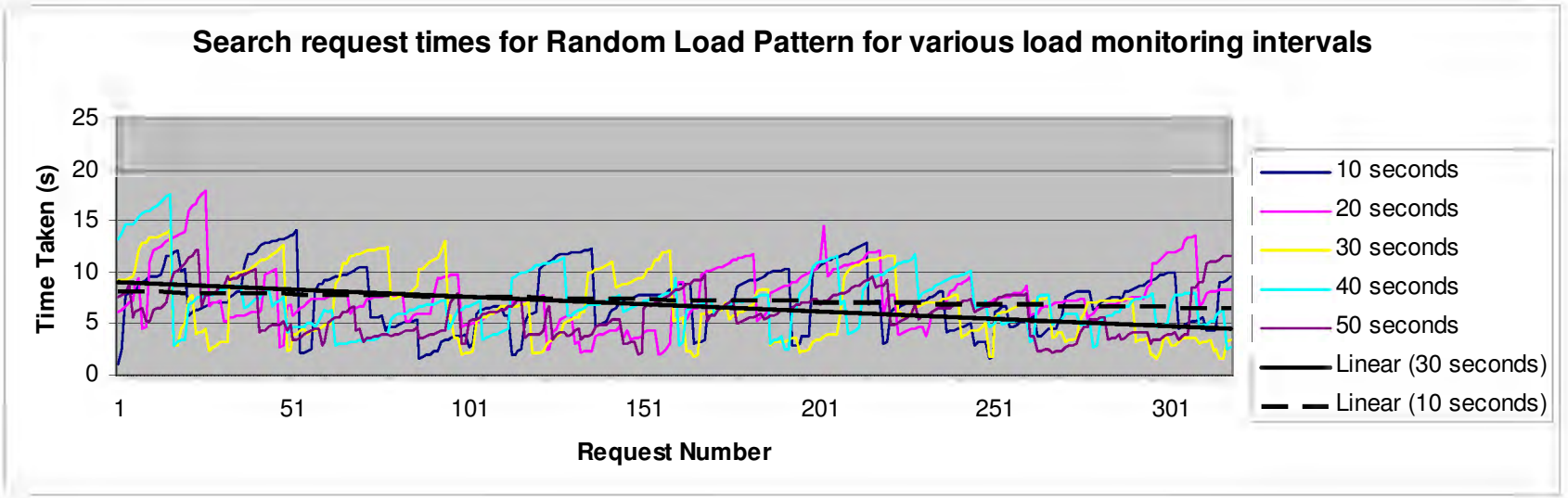

Figure 53: Search interface response times for various load monitoring intervals using random load pattern with linear trend lines

Appendix E

Graphs (with trend lines) depicting the response times for various overload criteria for random load pattern.