Component-based Digital Library scalability using Cluster Computing

M.Z. Omar
Department of Computer Science
University of Cape Town
Cape Town
South Africa
momar@cs.uct.ac.za

H. Suleman
Department of Computer Science
University of Cape Town
Cape Town
South Africa
hussein@cs.uct.ac.za

Abstract

Most institutions make use of digital library systems (DL) to deal with the information they use. DL systems allow them to provide information management services, including the ability to search through and browse the information they have made available. The current architecture of the systems however, does not scale well when the amount of information increases dramatically.

A good example of a digital library would be a national collection of academic theses. Such a system provides anyone the service of searching, browsing and viewing the theses in its collection. These services allow users to effectively and efficiently locate and make use of the materials available. With an increase in the number of users and underlying information these systems degrade quite rapidly though.

This research is therefore aimed at creating digital library systems using components or services with the ability to migrate and replicate themselves on a collection or cluster of computers. Component based systems have proven to be more extensible and maintainable than monolithic systems. Each component encapsulates the functionality it requires and can be tested, modified and used without affecting other parts of the system. The components in this experimental system consist of Web services and are mobile, in that they have the ability to move around the cluster. Migration allows services or components to move to different locations in order to maximally use available resources. Replication serves to improve performance by improving availability as well as creating duplicates of services as an increase in the need for them arises. These two service enhancements promote a dynamic architecture with a greater degree of system efficiency and reliability.

Currently, routing and migration modules have been implemented for the proposed system. Initial performance tests have been conducted and the results from these are discussed and analysed.

Keywords: digital library, scalability, migration, web services, components

1. Introduction

Digital Library – or Information Management (IM) – systems are software systems to help with the management of metadata and data, as well as provide end-user services for activities such as submission, discovery and retrieval of digital objects. In most cases these systems have developed out of the needs of brick-and-mortar libraries to manage digital equivalents of their holdings, especially catering for the new trend towards publishing solely in digital form.

The existing generation of such software tools – systems such as Greenstone (Witten and Bainbridge, 2002), DSpace (Tansley, et al., 2003) and Eprints (University of Southampton, 2006) – have made it possible for non-programmers to easily set up and manage a digital archive. However, all of these are relatively young tools and have much room for improvement.

One often requested feature is the ability to scale to handle much larger collections of data and/or much higher numbers of requests for services. Improving scalability is unfortunately not trivial – in most cases it requires a re-engineering of the core architecture. Even then, the nature of data processing places natural bounds on the increase in performance possible (Wilkinson and Allen, 1999). In such situations the only solution is to use faster computers or more computers. This is the approach taken in this research – to address scalability issues in an information management system by using a cluster of computers.

Another often requested feature of IM systems is the ability to add on new functionality to existing systems. This has led to a number of research efforts on extensible component frameworks (Suleman and Fox, 2001) and a gradual adoption of the concept by production systems. Greenstone 3 has already incorporated a Web Services-based component model (Bainbridge, et al., 2004) and DSpace has documented an intention to consider this for its next version (Tansley, 2004).

In bridging these two requirements, this research effort has thus attempted to exploit the components being created in IM systems in order to appropriately distribute tasks within a Beowulf cluster, without the applications needing to be particularly cluster-aware. In order to support scalability at this high level, it has been necessary to design and implement a node-independent service layer using DNS-like resolution of location-independent identifiers to URLs. Then, a simple migration and replication system was added to transparently balance load across the cluster as and when needed.

The rest of this paper discusses relevant background work, the design and implementation of this high-level scalability framework and initial performance test results. Finally, an analysis of the current system and thoughts on future work are presented.

2. Background

This section covers technologies related to the system being developed. It discusses the Open Digital Library architecture, cluster computing, registry systems as well as migration and replication. All of these are incorporated into the system in one way or another.

2.1 Open Digital Library (ODL)

The Open Digital Library (ODL) framework (Suleman and Fox, 2001) was adopted for this research because of the availability of components and their simplicity of use.

The Open Digital Library suite consists solely of service-oriented Web-based components, each one performing a unique task or service. An extended version (XPMH) of the Open Archive Initiative's Protocol for Metadata Harvesting (OAI-PMH) is used by the component instances to communicate with one another. While certain components are completely independent, others rely on additional components for common functionality (such as the storage of metadata). Users can download specific components in order to customise their digital library systems 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 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. Component instances can be queried directly, using the verbs/requests in the XPMH, or indirectly if such an interface is provided. Individual ODL protocols for components are also specified as each component supports specific functionality (Suleman, et. al, 2003).

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

Table 1 contains a list of some of the ODL components available.

Table 1: List of ODL Components, descriptions and protocols (Suleman, 2002)

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

The ODL architecture provides a platform that may be extended for dynamic configuration of services given its distributed design. This feature has been exploited in this research by overlaying the ODL components on a Beowulf cluster.

2.2 Cluster Computing

Cluster computing is best characterized by combining a number of off-the-shelf computers and resources, using both hardware and software, in order to work as a single machine (Duggan, 2001).

Over the past 10 years cluster computing has grown dramatically and there is a wide variety of software and hardware choices when creating a cluster today. Currently clusters are classified according to their functionality (Burleson, 2003). These include:

- 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 cluster
  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 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.

Cluster software can be categorised 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. By making use of a cluster correctly, systems can speed up their execution time and provide better service. Having multiple computers may increase scalability of a system as more resources are available. However, it is necessary to track the location of compute resources within the cluster, especially at the high level of Web-based services.

2.3 Domain Name System (DNS)

The DNS system (Network Working Group, 1987) fulfils the role of a resource locator on the Internet – mapping generic resource (typically domain) names to specific locations.

A domain name is used to refer to resources on the Internet, in preference to a fixed IP address. The URL http://www.howstuffworks.com contains the domain name “www.howstuffworks.com”. Similarly, the email address “iknow@howstuffworks.com” contains the domain name “howstuffworks.com”. These URNs 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 (HowStuffWorks, Inc., 2006). This conversion from the human-readable address to the IP address is provided by the DNS.

The DNS has three main components (Network Working Group, 1987), 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 and a particular name server may have 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 authoritative 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.

Based on the immense size of the database to manage all online 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.
In practice, after typing in a URL into a Web browser, it is passed to the local resolver on the machine. The resolver then takes this URL and queries a name server to retrieve the IP address associated with the URL. If the name server has the address it will send it to the resolver and the resolver 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 URL. This process may continue until a name server with the IP address is located or until the domain name is considered invalid.
The DNS system has been proven to work both efficiently and effectively (Berinato, 2002). 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. The DNS system provides a model that was used to develop the registry component of the system for this research project.

2.4 Migration

Process migration allows for processes or objects to be moved from one machine to another during execution in a distributed environment. Migration is transparent if the process or object is unaware that the move has taken place (Nuttal, 1994).

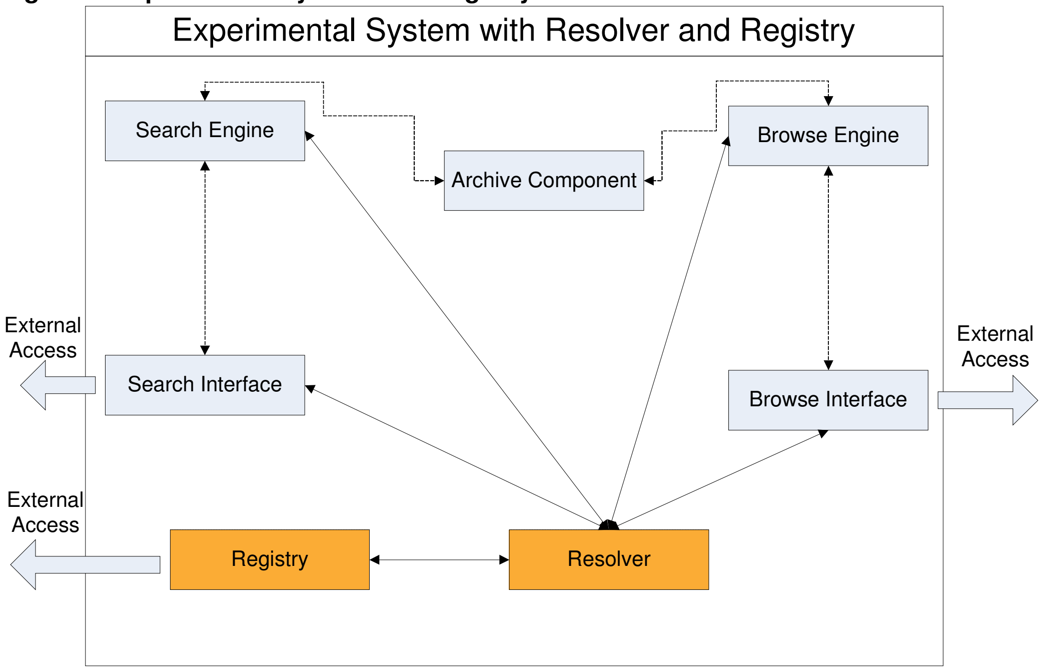
It is of primary interest due to its offering improved performance over static allocation schemes for processes and objects, where the system load is unstable or not known a priori. Migration can be used to speed up the execution time of a single task. It enables dynamic load balancing, fault resilience, ease of administration, and data access locality (Milojicic, et. Al, 2000). For the purpose of this paper the emphasis will be on object migration of component instances.

Systems which support object migration include the Persistent Object Infrastructure by Mike Olsen, the Chorus Object-Oriented Layer developed by Chorus Systémes and Emerald, to mention only a few (Nuttal, 1994). In order to improve system performance migration as well as replication of components will be implemented as part of the system being developed.

3. Analysis and design

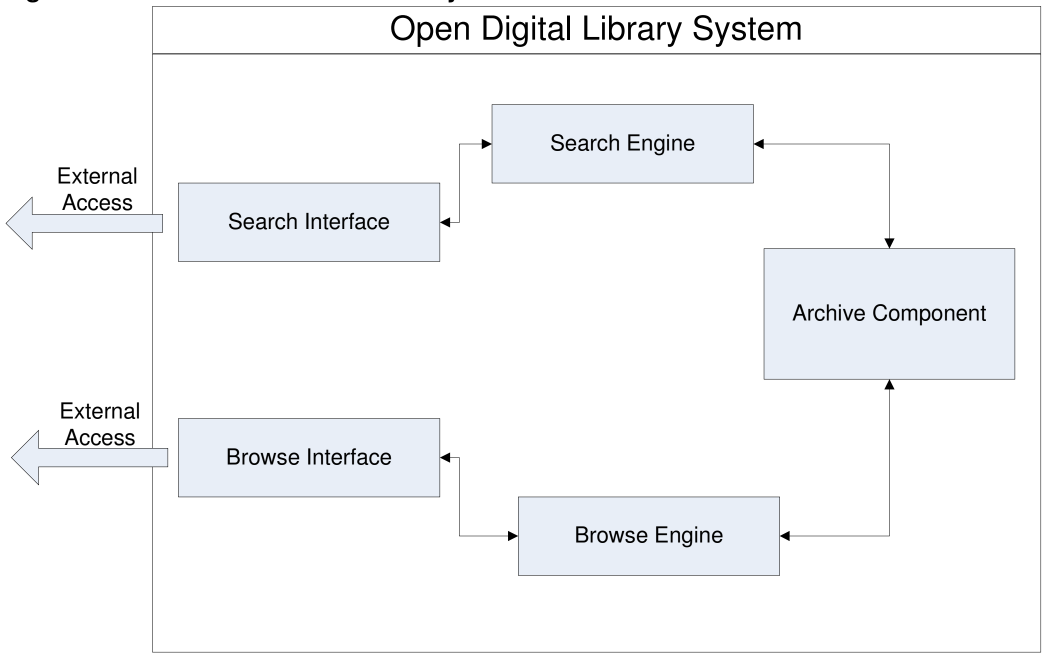
In order to develop a digital library system that manages mobile component instances, additional infrastructural services had to be introduced. These services are called the Registry and the Resolver. The Registry is responsible for tracking all component instances in the system while each Resolver would track components on its local machine. The figure below shows this system (on a single machine) and will be referred to as the experimental system.

Figure 1: Experimental System with Registry and Resolver



The system was based on an ODL architecture (as shown in Figure 2) which was extended as discussed later in this section. The system itself is spread across a cluster. In the rest of this document, machines inside the cluster will be referred to as nodes. The cluster consists of 13 3Ghz Pentium nodes connected via a Gigabit switch, of which appropriately-sized subsets were used for development and testing of this system. The modifications to the original ODL architecture are discussed below.

Figure 2: ODL architecture used for system



The first phase was the implementation of the Registry and Resolver components. The need for these components arose due to the introduction of component migration and replication. The traditional ODL architecture uses hard-coded URLs inside the component instances. Thus, in a system where components can migrate, once a component has moved the hard-coded URL would be invalidated and the system would no longer function correctly. Thus the Registry and Resolver were introduced in order to alleviate this problem and they are discussed below.

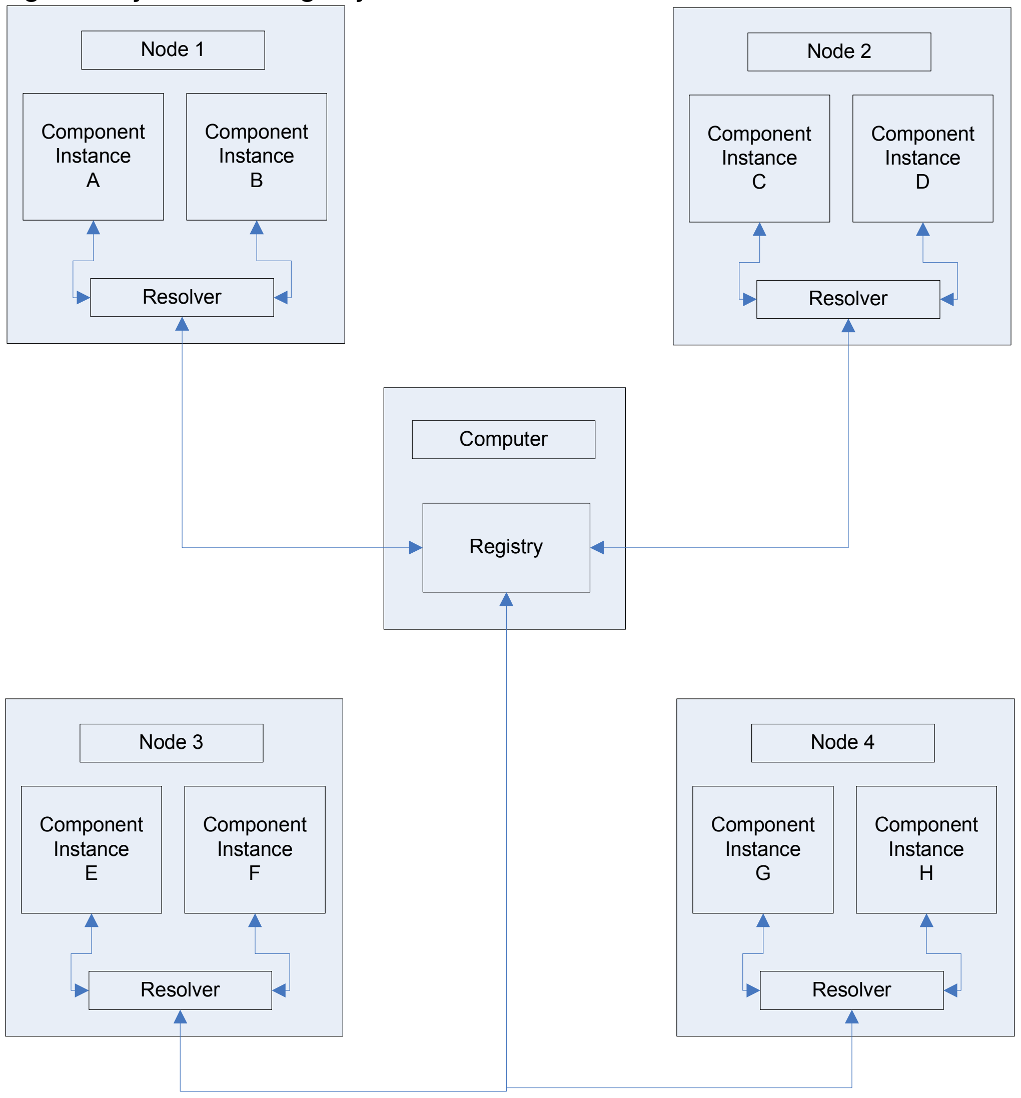
A Resolver resides on each node in the cluster. It keeps track of all component instances residing on that node and could be seen as a local registry. The Resolver acts as an intermediary between the component instance and the Registry. The Resolver's list contains the name, version, type, owner, the actual URL to the component instance and a database name if the component has one. This list is only updated when existing component instances migrate or replicate and new component instances are created or removed. An entry is removed when a component instance is removed or migrates to another node. An entry is added to the list when a component instance is created or an instance is migrated or replicated to the node. When a component instance requires a URL to another component it sends a request to the Resolver. The Resolver then forwards the request to the Registry. The Registry responds with the appropriate URL and the response is forwarded from the Resolver to the component.

The main responsibility of the Registry is to keep track of all components within the system and provide their associated URLs on demand. It stores all of this information in a database – it contains the following information for each instance:

- name of the component instance,
- type of component the instance is,
- the actual URL to the instance,
- the version of the instance,
- the status of the instance (running or migrating/replicating), and
- the owner of the instance.

When a node starts up the list of component instances residing on it is sent to the Registry and the list is updated. When a node is shut down, the Registry is informed and it removes all component instances which reside on that node from the list. The list is also updated during component instance migration or replication. After migration, the component instance URL is updated in the Registry. During replication though, a complete new entry is added. Finally, when an instance is created, the list adds a new entry and when an instance is removed, an entry in the list is removed. The Registry also sends out URLs to component instances (e.g., search interface) that depend on other component instances (e.g., search engine) for some request to be fulfilled. The Registry communicates with components via the Resolver. The entire component instance, Registry and Resolver system is seen in Figure 3.

Figure 3: System with Registry and Resolver



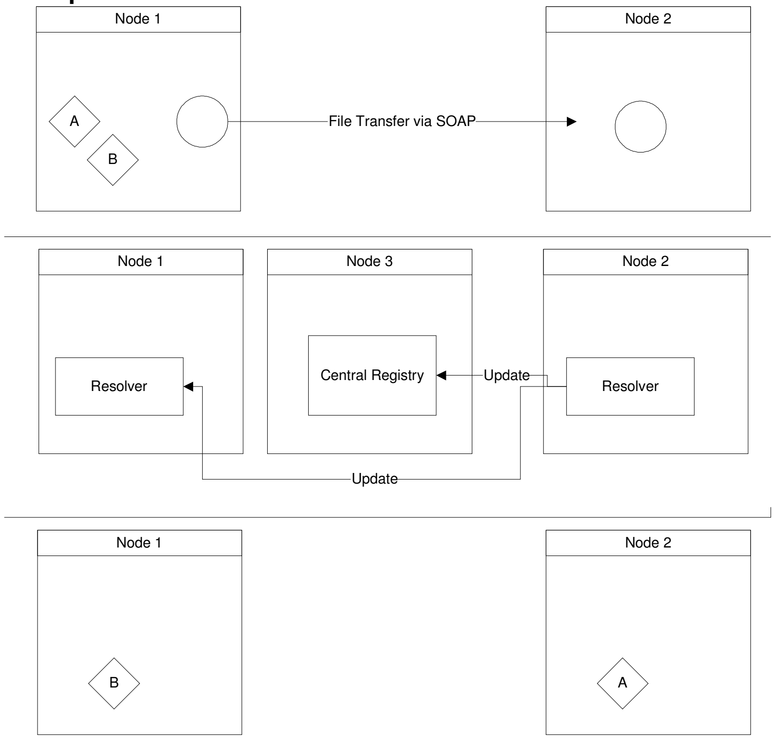
The second phase was the modification of each component so that they can migrate and replicate. Components are divided into two different categories with respect to migration and replication. These are components that have associated data (e.g., databases or flat files) and those that do not have any data associated with them (such as interface components). The system caters for both types of components. The process of migration is explained in the following paragraphs.

When a component instance is requested to migrate, the instance determines whether it may or may not migrate. Once the migration request has been accepted, the component creates a copy of itself inside a predefined directory. This is done by copying the component instance directory to the predefined one. Any databases or flat files that the component instance uses are also copied into this directory. This directory is then compressed and encoded into an XML document using base 64 encoding. This document is then transferred to a migration service on the destination machine using HTTP.

Once the migration service has received this XML document, the archive file is recreated and uncompressed. The component instance itself is then moved to the directory in which a template of the component exists. Any data dependencies are set up. Once the instance has been set up and is working correctly, the original component instance is removed from the original node. This process is illustrated graphically below.

Figure 4: Migration of a component instance

1. Component instance A is zipped up along with its dependencies and is encoded into an XML document. This is transferred to node 2 via HTTP
2. After setting up component instance A on node 2, the local Resolver, the Registry and the Resolver on node 1 are updated.
3. Component instance A is then removed from node 1. Component instance A has now migrated to node 2.



The replication of a component happens in the same way as the migration with the exception that the original component instance is not removed from the system.

These modifications to the system provide a basis for system scalability. Testing the impact of the Registry, Resolver, additional communication and other overheads is discussed in the next section.

4. Initial evaluation and testing

The aim of testing is to evaluate the various performance impacts of the introduction of a Registry and Resolver as well as the extra communication introduced to the component instances. Ultimately, the system will be tested for its ability to scale – when the implementation of a load balancer has been completed. The series of test cases outlined below have been created in order to evaluate particular aspects of the system and how it compares to the traditional system.

4.1 End-to-End communication test

The first part of the testing was aimed at evaluating the overall slow-down of the system due to the introduction of the Registry and Resolver. This test had both the experimental and normal (historical) system running on a single node (at the same time).

The architectures of both experimental (Figure 1) and normal systems (Figure 2) can be seen in the previous section. Batch jobs of 200 requests were sent to the search and browse interfaces and the results are documented below in Tables 2 and 3. The requests were sent serially. The request to the search interface was for the search term 'computer'. The browse interface was requested to display the records in descending order by title and date.

Table 2: Normal and Experimental System Overall Search Time in Seconds

Run Number	**Normal Search System**	**Experimental Search System**
1	101.402	143.537
2	101.543	143.028
3	100.542	142.985
4	100.516	143.557
5	100.493	143.262
6	101.598	143.167
7	100.545	143.611
8	100.537	143.554
9	100.668	142.997
10	100.670	143.661

Table 3: Normal System and Experimental System Overall Browse Time in Seconds

Run Number	**Normal System Browse**	**Experimental System Browse**
1	249.514	334.087
2	249.548	333.924
3	249.573	333.675
4	249.473	333.952
5	249.550	333.872
6	249.590	333.921
7	249.704	334.020
8	249.467	333.729
9	249.482	333.824
10	249.433	333.922

The results obtained from this test indicate a delay of approximately 43 seconds for a set of 200 search requests submitted to the experimental system. Therefore the delay can be seen as a 43/200 or 0.215 second delay per request. This of delay is relatively small for an end user submitting requests to the search interface. The browse interface delay is approximately 85 seconds for the entire batch of 200. The delay for an end user in this case is 85/200 or 0.425 seconds. In both cases, for search and browse, the delay is minimal and is a small cost for the introduction of the Registry and Resolver components.

It is expected that the benefit of supporting migration and replication will more than make up for this additional cost.

4.2 Distributed components test

The second test had both systems distributed over multiple nodes. With this test the performance impact as well as the additional network communication between component instances could be taken into account. The figures below show how the instances were distributed and the results for this experiment (with identical parameters to the prior test) are documented as well.

Figure 5: Distribution of experimental system for testing

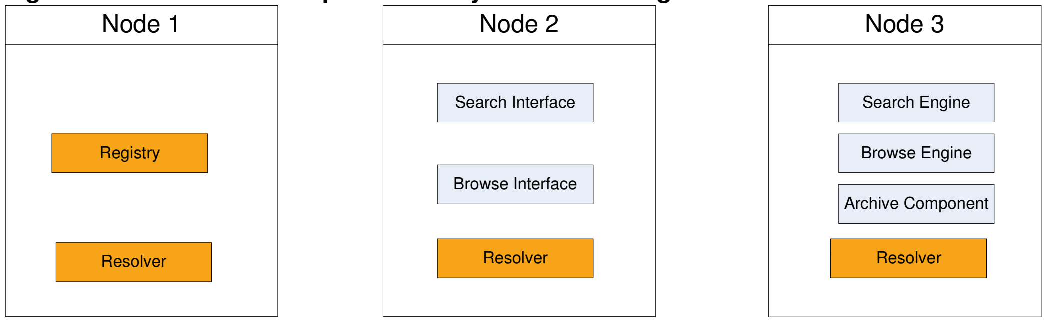


Figure 6: Distribution of normal system for testing

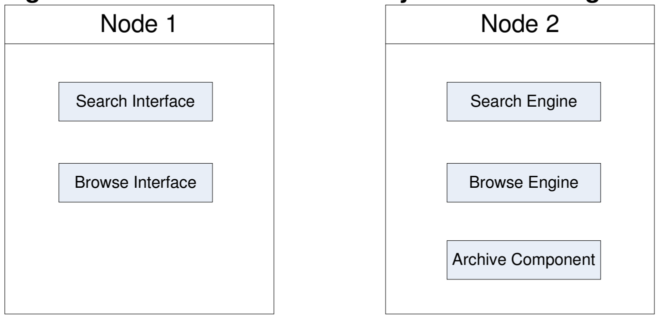


Table 4: Distributed experimental and normal system request times

Request to	Average experimental time	Average normal time
Search	1.199	0.592
Browse	8.0315	2.366

The results above show an increase in both search and browse time which was expected. The experimental search component took twice as long as the normal one. This would still be acceptable as it is only a difference of 0.5 seconds. The browse component took four times as long which was unexpected. This may be due to other network traffic on the system or the way in which the browse component was configured. We can thus ascertain that the browse component needs more attention to perform optimally.

4.3 Single instance performance test

The third part of the testing focused on the performance impacts on individual component instances, as a result of indirection through the Registry and Resolver services. The component instances in the traditional system were compared to those in the experimental one. Each instance was tested directly using the command line. The instances that were used in this test were the search and browse engines. The tests requested that both the search and browse instances retrieve a record from the archive component instance and this was done in batch jobs of 100. The performance differences between the traditional and experimental components, for the test, are as follows:

Table 5: Normal and Experimental Search Component Times in Seconds (command line)

Run Number	Normal Search Component	Experimental Search Component
1	11.793	22.479
2	11.809	22.485
3	11.789	22.536
4	11.789	22.508
5	11.817	22.499
6	13.233	22.498
7	11.766	22.477
8	11.759	22.507
9	11.787	22.604
10	11.799	22.487

Table 6: Normal and Experimental Browse Component Times in Seconds (command line)

Run Number	Normal Browse Component	Experimental Browse Component
1	18.895	21.390
2	18.901	21.399
3	18.887	21.407
4	18.931	21.439
5	18.887	21.486
6	18.889	21.417
7	18.938	21.408
8	18.910	21.401
9	18.939	21.413
10	18.890	21.488

The testing of the browse engines individually show that the extra communication introduced is very small. The time difference in the request time for 100 requests is just under 4 seconds, making the delay time for a single request 0.04 seconds, which is barely noticeable to someone using the component instance. The search engine however performed differently. The search engine in the experimental system took almost twice as long as the normal one. This makes the delay time for the search engine 0.11 seconds, which is quite large compared to the browse engine but still quite small in terms of search time.

4.4 Overheads isolation test

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. The network traffic between the following component instances was monitored on the distributed system (Figure 5):

- between the Resolver and the component instances and
- between the Registry and Resolver.

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.

Table 7: Bytes sent and received from component instance to Resolver as well as time taken

Component	Bytes sent	Bytes Received	Average Time	Request for
Search Interface	87	388	0.01911	Search Engine URL
Browse Interface	91	400	0.020237	Browse Engine URL
Search Engine	90	393	0.019506	Archive URL
Browse Engine	90	393	0.020064	Archive URL

Table 8: Bytes sent and received from Resolver to Registry as well as time taken

Component	Request from	Bytes sent	Bytes Received	Average Time	Request for
Resolver	Search Interface	42	388	0.010076	Search Engine URL
Resolver	Browse Interface	45	393	0.010124	Browse Engine URL
Resolver	Search Engine	45	393	0.010012	Archive URL
Resolver	Browse Engine	45	393	0.010061	Archive URL

As can be seen from the results in the table 6, the overall time used for networking for each component instance is quite small. In table 6 we have the time it takes a component to get a URL from the Registry. This includes the time in table 7 for each component respectively. The time for connecting an instance to the Resolver is just over 0.010 seconds and the time for the Resolver to connect to the Registry is 0.010 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.

5. Conclusions

The use of a cluster of computers shows promise as a mechanism to achieve scalability for a component-based digital library system. An overall system design for such an approach is based on existing tools and practices in Internet technology (such as DNS) and cluster computing, albeit at a high level. The initial implementation of this system, to support location independence and resolution, has been tested and the services have achieved acceptable performance levels. It is expected that with migration, replication and simple load balancing added, this experimental system will provide sufficient evidence to support the use of clusters for high-level scalability of information management systems.

6. Future work

The current system is a proof-of-concept one and provides a platform for additional work to be done. It serves to prove that this type of system could perform better than other systems not incorporating migration and replication of services as well as provide a possible solution to scalability problems. The current system has various aspects which can be improved upon. These are discussed in the following paragraphs as well as how they form part of the current system.

The introduction of a simple load-balancing component will be done as part of the research. This load-balancing component is responsible for ensuring that requests go to the appropriate component instances which have the least load. It will also decide when these instances should migrate and replicate in order for the system to perform optimally at all times.

Components may vary in the way they operate in terms of system resources. Finding combinations of components which compliment one another may increase performance of the system. Having a model for components which perform optimally when placed together would also help with migration and replication decisions.

Another improvement is the granularity of service provision. Here we need to determine which services are being used more often and make them more available through replication. Certain services may only be needed for a short period of time and these services should then be removed from the system as soon as the duplicates are no longer needed.

A better load-balancing component will need to be implemented. This could encompass some sort of artificial intelligence system which could learn when certain component instances are needed more than others and then replicate them as needed. It should also have an idea of request times for different types of components so that request routing can be improved.

With all of the above-mentioned improvements, a fully functional and working version of this experimental system could be implemented and used.

7. List of references

Bainbridge, D., K. J. Don, G. R. Buchanan, I. H. Witten, S. Jones, M. Jones and M. I. Barr. 2004. Dynamic Digital Library Construction and Configuration. In Proceedings of Research and Advanced Technology for Digital Libraries: 8th European Conference (ECDL2004), Bath, UK, 12-17 September 2004, LNCS 3232, Springer.

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

Burleson, D. 2003. RAC – Types of Clusters. [Online] Available: http://www.praetoriate.com/oracle_tips_mamt_cluster_types.htm (Accessed 15 June 2005)

Duggan, A. 2001. Technical Supplement – Beowulf Computer Clusters. [Online] Available: http://www.tessella.com/Literature/Supplements/PDF/beowulf.pdf (Accessed 14 June 2005)

Gerry McGovern Publications, 2006. Content Critical: Chapter One: Everything you know about publishing is wrong: Part Two: Two: It's an information overloaded world. [Online] Available: http://www.gerrymcgovern.com/cc_ch1_2.htm (Accessed 12 May 2006)

HowStuffWorks, Inc. 2006.Howstuffworks "How Domain Name Servers Work". [Online] Available: http://computer.howstuffworks.com/dns.htm/printable (Accessed 21 June 2005)

Kaufmann, M (ed). 2002. How to build a digital library. San Francisco, CA.

Milojicic, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S. 2000. Process Migration. ACM Computing Surveys (CSUR), 32(3):241-299. [Online] Available: http://delivery.acm.org/10.1145/370000/367728/p241-miloiic.pdf?key1=367728&key2=1102019411&coll=portal&dl=ACM&CFID=72555809&CFTOKEN=28259796 (Accessed 15 May 2006)

Network Working Group. 1987. RFC 1035 Domain Names – Implementation and Specification. [Online] Available: http://www.rfc-editor.org/rfc/rfc1035.txt (Accessed 15 June 2005)

Network Working Group. 1987. RFC 1034 Domain Names – Concepts and Facilities. [Online] Available: http://rfc.dotsrc.org/rfc/rfc1034.html (Accessed 21 June 2005)

Nuttal, M. 1994. A brief survey of systems providing process or object migration facilities. ACM SIGOPS Operating Systems Review. 28(4):64-80. [Online] Available: http://delivery.acm.org/10.1145/200000/191541/p64-nuttall.pdf?key1=191541&key2=2391019411&coll=portal&dl=ACM&CFID=72555809&CFTOKEN=28259796 (Accessed 15 May 2006)

Suleman, H., Fox, E.A., Kelapure, R., Krowne, A., Luo, M. 2003. Building digital libraries from simple building blocks. Online Information Review, 27(5): 301-310. [Online] Available: http://www.emeraldinsight.com/Insight/ViewContentServlet?Filename=Published/EmeraldFullTextArticle/Articles/2640270501.html (Accessed 12 May 2006)

Suleman, H. 2002. Open Digital Libraries, Ph.D. dissertation, Virginia Tech. [Online] Available http://www.husseinsspace.com/publications/odl.pdf (Accessed 12 May 2006)

Suleman, H. and E. A. Fox (2001). A Framework for Building Open Digital Libraries. D-Lib Magazine 7(12). [Online] Available: http://www.dlib.org/dlib/december01/suleman/12suleman.html (Accessed 15 May 2006)

Tansley, R. 2004. DSpace 2.0 Design Proposal, presented at DSpace User Group Meeting, 10-11 March, Cambridge, USA. [Online] Available: http://wiki.dspace.org/DspaceTwo (Accessed 16 May 2006)

Tansley, R., Bass, M., Stuve, D., Branchofsky, M., Chudnov, D., McClellan, G., Smith, M. 2003. The DSpace Institutional Digital Repository System: Current Functionality. In Proceedings of Joint Conference on Digital Libraries 2003, Houston, TX, May 27-31, 2003, ACM Press, New York, NY, 87-97. [Online] Available: http://delivery.acm.org/10.1145/830000/827151/p87-tansley.pdf?key1=827151&key2=3503019411&coll=GUIDE&dl=ACM&CFID=11792774&CFTOKEN=13954292 (Accessed 16 May 2006)

Witten, I.H., Boddie, S.J., Bainbridge, D., McNab, R.J. Greenstone: A Comprehensive Open-Source Digital Library Software System. *Proceedings of the Fifth ACM Conference on Digital Libraries,* San Antonio, Texas, 2000, 113 – 121. [Online] Available http://delivery.acm.org/10.1145/340000/336650/p113-witten.pdf?key1=336650&key2=9711019411&coll=portal&dl=ACM&CFID=72555809&CFTOKEN=28259796 (Accessed 14 May 2005)

Wilkinson, B., Allen, M. 1999. Parallel Programming. Prentice Hall, New Jersey.