Imperial College of Science,

Technology and Medicine

(University of London)

Department of Computing

Distributed Pagerank: Comparisons between a Simulation and Peer-to-Peer Implementation of the Algorithm

by

Lida Tsimonou (L.T.)

Submitted in partial fulfilment

of the requirements for the MSc

Degree in Computing Science of the

University of London and for the

Diploma of Imperial College of

Science, Technology and Medicine

September 2004

Acknowledgments

This thesis has been completed with the invaluable support of Dr Jeremy Bradley. His guidance at all stages of the project has been essential in producing this piece of work.

A special thank you also goes to family and friends for their support throughout the year.

Abstract

The Pagerank algorithm was first introduced by Sergey Brin and Lawrence Page. Pagerank can be described as a measure of popularity assigned to each web page. The idea behind the algorithm is to infer the popularity of a web page based on information provided by the link structure of the web graph. Except from web page ranking, the algorithm has also been applied to ranking documents in peer-to-peer file-sharing systems as a means to reduce network traffic.

The algorithm has traditionally been implemented centrally, on a set of pages obtained by a centralized web crawler. This approach suffers from performance and scalability issues and as a result distributed Pagerank algorithms have been proposed in literature. The present project presents an implementation of a peer-to-peer, distributed Pagerank algorithm, first introduced by Sankaralingam et al. in 2003, which is based on chaotic relaxation.

In order to be able to evaluate the accuracy of the Pagerank values generated by this system, an additional application has been developed which runs as a stand-alone system, simulating the multiple-peer environment, but avoiding possible peer communication issues.

The results obtained suggest that the peer-to-peer system developed can converge with considerable accuracy when compared to the simulated version. In addition, the system can deal effectively with dynamic changes like node arrivals and page insertions. Certain issues related to dealing with increased message traffic during Pagerank calculations, as well as a trend of the distributed system to underestimate Pagerank values have been identified. Possible strategies for improvement, as well as suggestions for future work are discussed.

Acknowledgments 1
Abstract.. 2
1. Introduction... 6
1.1. Ranking mechanisms 6
1.2. Aims of the project 7
1.3. Structure of the report 7
2. Theoretical Background & Derivation of the Algorithm 9
2.1. Graph Theory Terminology 9
2.2. Original Pagerank definition 10
2.3. Markov Chains 11
2.4. Derivation of the Pagerank Algorithm.. 13
3. Computation of Pagerank 16
3.1. Centralized Implementation 16
3.2. Distributed Pagerank.. 18
3.2.1. Implementation as a web server function... 18
3.2.2. Implementation within a peer-to-peer framework 19
3.2.3. Considerations in a distributed system.. 19
3.3. Chaotic Iterations Method.. 19
3.3.1. Implementation evaluation 20
3.3.2 Testing Evaluation 21
3.4. Open-System Pagerank.. 23
3.4.1. Simulation Evaluation 24
3.5. Conclusions 26
4. Peer-to-peer framework for distributed algorithms.. 27
4.1. Overview 27
4.2. The Peer-to-Peer Software Architecture 27
4.3. Structured versus unstructured P2P Overlays 28
4.3.1. Unstructured P2P Overlays 28
4.3.2. Structured P2P Overlays 29
4.4. Structured P2P Overlays: Pastry 30
4.4.1. Identifiers and identifier space 30
4.4.2. Routing... 30
4.4.3. Routing performance... 31

4.4.4. Node arrivals and departures ... 31
4.4.5. Locality ... 32
4.5. Conclusions ... 32
5. System Design ... 33
5.1. Algorithm Overview ... 33
5.2. Design issues ... 35
5.2.1. Linkage Information ... 35
5.2.2. Structure of Updates ... 35
5.2.3. Updates Frequency ... 36
5.2.4. Dynamic features ... 36
6. Implementation: General issues ... 37
6.1. Programming language & P2P Overlay choice ... 37
6.2. Web graph & Link Extraction ... 38
6.3. Format of extracted graph file ... 39
7. Implementation: Part A - Simulation of a Distributed Pagerank Calculator ... 40
7.1. Graphical user interface ... 40
7.2. Package Structure ... 41
7.3. Web page representation ... 42
7.4. Web graph handling ... 42
7.5. Simulation of the distributed environment ... 44
7.5.1. Host Manager: Creating hosts ... 44
7.5.2. Performing Pagerank calculations ... 45
7.5.3. Managing retransmissions ... 46
8. Implementation: Part B - Distributed Pagerank Calculator ... 47
8.1. Distributed Pagerank Calculator: Package Structure ... 47
8.2. Application development on top of Pastry ... 48
8.3. Communication between nodes ... 49
8.3.1. Message Interface & Message Types ... 49
8.3.2. Continuation (Asynchronous communication) ... 50
8.4. Representation of web pages ... 52
8.5. Inserting new web pages into the system ... 53
8.6. Dealing with update message traffic ... 54
8.6.1 Using threshold updates ... 54
8.6.2. Distinguishing between update messages ... 55

8.7. Confirming delivery of messages ... 57
8.8. Saving the state of the system ... 58
9. System Testing & Evaluation ... 59
9.1. Experimental measures ... 59
9.1.1. Simulation measures ... 59
9.1.2. P2P performance measures ... 59
9.1.3. Comparison measures ... 60
9.2. Data sets & Setup procedures ... 60
9.3. Pagerank Value and Indegree of Pages ... 61
9.4. Convergence in the Simulated Environment ... 61
9.5. Convergence in the P2P Environment ... 64
9.6. Comparing the Simulation & P2P system accuracy ... 66
9.6.1. Accuracy in a single-peer system ... 66
9.6.2. Accuracy in a multiple-peer system ... 67
9.7. Dynamic Page Insertions ... 70
9.7.1. Simulation System ... 71
9.7.2. Peer-to-peer System ... 72
9.7.3. Accuracy of Pagerank Values ... 72
9.8. Node arrivals & departures ... 73
9.8.1. Simulation System ... 73
9.8.2. Peer-to-peer System ... 74
9.8.3. Accuracy of Pagerank Values ... 74
10. Conclusions ... 76
10.1. Outcomes of the current project ... 76
10.2. Limitations & Future work ... 76
10.3. Concluding comments ... 78
References ... 79

1. Introduction

The Pagerank algorithm was first introduced by Sergey Brin and Lawrence Page – the founders of Google - in a paper called "The Pagerank Citation Ranking: Bringing Order to the Web" in 1998. In that paper, parallels were drawn between academic publications and web pages. The idea behind these parallels was that just like high quality academic papers are more often cited in other academic papers, popular web pages are linked more often from other web pages. In very broad terms then, this paper, looked at the link structure of the web graph in order to infer some information about the quality and popularity of a given web page. This concept was quantified in a variable called Pagerank, which can be thought of as a measure of popularity assigned to each web page.

1.1. Ranking mechanisms

Considering the size of the World Wide Web, it is essential for a successful search engine to develop efficient ranking mechanisms, if users are to make any sense and use of the results returned. Pagerank is the ranking method that has been associated with Google and is the focus of this project. It should be noted that Pagerank is typically used in conjunction with other methods in order to achieve optimum ranking of results returned for a given web search. Among those other measures are relative font size of the specified keywords as well as their frequency of occurrence. These measures will not be further discussed in this context, as they are not directly related to Pagerank.

At the same time, as the popularity of the Internet has grown and peer-to-peer file sharing systems have gained prevalence among increasing numbers of users, the importance of ranking mechanisms in those systems has also been realized. As the content shared by users increases, it is important to be able to rank this content in some way so as to make user queries more effective. In addition, in those systems, efficient ranking mechanisms can significantly improve performance by limiting traffic generated by a query.

It is known that Google had been using until recently a centralized version of the algorithm, the idea being that web pages are crawled by a cluster of crawlers, linkage information of the crawled pages is extracted and subsequently the Pagerank calculations are applied to those pages. Considering the size of the crawled web (Google currently holds 4 billions of crawled paged), these calculations take a significant amount of time (in the magnitude of days) to complete. In addition, such a system is not flexible, as it does not allow for dynamic incorporation of new pages into the calculations. Rather, new pages are only considered

during the next crawl, thus not providing a system that is responsive to dynamic changes in the World Wide Web.

As these issues of performance become even more prevalent with the increasing size of the web, and as the concept of a ranking algorithm expands from web search engines to peer-to-peer file sharing systems, the idea of a distributed implementation of the algorithm, which is responsive to dynamic changes, gains increasing popularity.

During the last two years, a number of papers have been published, which present possible implementations of a distributed Pagerank algorithm (e.g. Sankaralingam et al., 2003; Shi et al., 2003). An idea proposed has been to implement a distributed version of the algorithm on an Internet scale as a service provided by web servers, thus spreading the burden of calculations (e.g. Wang & DeWitt, 2004). An alternative solution has been to incorporate the calculations in peer-to-peer systems, in which each peer is responsible for a set of pages (e.g. Sankaralingam et al., 2003) and peers exchange Pagerank related information in order to complete the calculations.

1.2. Aims of the project

The present project aimed at developing a distributed version of the Pagerank algorithm in a peer-to-peer context. The idea was to evaluate the convergence properties of a distributed Pagerank algorithm, experiment with dynamic properties of the system (e.g. how it reacts to host failures or new page insertions) as well identify potential issues that could arise in a larger-scale implementation of such an application.

Before implementing the peer-to-peer based application, a simulated version of the system – providing the same functionalities as the distributed version - was developed in order to get a feeling of the structure and possible issues in such a system. The simulated system has been used as a control for evaluating the accuracy of the results returned by the distributed Pagerank system.

1.3. Structure of the report

Before proceeding, the structure of the report will be briefly outlined. The following two sections provide an overview of the Pagerank algorithm and its derivation, as well as a discussion of existing implementations of a distributed version of the algorithm. Next, the

main features of peer-to-peer systems are introduced, and available technologies are presented, with an emphasis on the technologies used in this project.

After this theoretical overview, the design of the system developed is briefly outlined. This is followed by a detailed discussion of the implementation, which is divided in three parts. As two applications – a simulation system and a peer-to-peer based system - were developed, each is discussed in a separate section. These two sections are preceded by a general implementation section, in which issues that are overlapping between the two applications are discussed.

The implementation sections are followed by the testing and evaluation section, which presents and discusses the results obtained during the testing phase of the project. Finally, the outcomes of this project together with limitations and suggestions for future work are summarized in the conclusions chapter.

2. Theoretical Background & Derivation of the Algorithm

Before proceeding to the specifics of the algorithm, the basic terminology relating to graph theory will be outlined so that the reader is familiar with the terms that will be used throughout this document.

2.1. Graph Theory Terminology

A graph can be thought of as a set of points called vertices or nodes, which are connected by a set of edges or arcs. Graphs can be undirected or directed. In the former case, the edges connecting the vertices have no direction, while in the latter case, as the name implies, edges point from the originating vertex to the target vertex. Vertices can also be connected to themselves by an edge, thus forming a loop, while it is also possible to have multiple edges between two vertices [3].

Adjacent vertices are vertices sharing a common edge. Another property of vertices is their degree, which is defined for an undirected graph as "the number of edge ends at that vertex" [3]. In a directed graph it is possible to distinguish between the in-degree and the out-degree of a vertex, the former being the number of edges that terminate at a given vertex, and the latter being the number of edges that originate from a given vertex.

In this context the web is seen as a directed graph whose nodes correspond to web pages and whose directed edges, connecting two nodes, correspond to outgoing links from the originating web page to the target page. A schematic depiction of such a graph can be seen in Figure 1. Thus, for example, there is an outgoing link (forward link) from page 1 to pages 2 and 4, while page 4 has three incoming links (backlinks) from pages 1, 4 and 5.

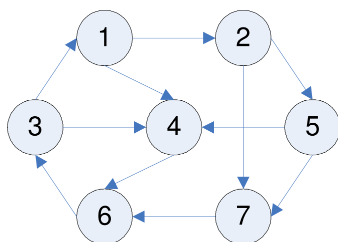

Figure 1. Small-scale depiction of the directed web-graph; nodes represent web pages; edges represent links from one page to another.

2.2. Original Pagerank definition

In this section the original definition and equations proposed in the Page and Brin [17] paper will be discussed, while in later sections modified Pagerank definitions proposed by other researchers will be presented.

There are a number of assumptions behind this Pagerank definition, which have been summarized recently by Kamvar et al. (2003) as follows. First of all, every incoming link from a given page *v* to another page *u* is regarded as a vote for the latter pager.

In addition the weight of this vote is stronger the less outgoing links page *v* has – i.e. the less pages it casts its vote for. Specifically, if $R(v)$ is the Pagerank value of page *v*, and $o(v)$ the number of pages it links to (outdegree), then it contributes to each of those links $R(v)/o(v)$ of its Pagerank value. This means that generally, the higher the Pagerank value of a given page, the higher the contribution to its outgoing links will be.

This implies that the ranking of a page will not only depend on the number of its incoming links but also on the quality of these links. Finally, as in the beginning, none of the Pagerank values are known an iterative process is required in order to calculate the Pagerank values of a set of pages.

Broadly speaking, in order to calculate the Pagerank value $R(u)$ for a given page *u*, we would look at the sum of the Pagerank contributions that each page linking to page *u* makes. This can be expressed as:

$$R(u) = \sum_{\forall(inlinks\ (v))} \frac{R(v)}{o(v)}$$

Equation 1

$R(v)$ being the Pagerank value of a given page v that links to page u, and $o(v)$ being the out-degree of this page v.

This is only one component of the Pagerank equation and represents the basic idea behind the algorithm. From now on, we will look at the technical details relating to the algorithm and will discuss all the issues that need to be handled so that the algorithm satisfies certain properties and can converge to a consistent solution.

2.3. Markov Chains

An intuition behind the Pagerank equation is that of viewing the user as a random surfer, which in turn has been influenced by the notion of random walks in graph theory. A random walk through a graph is a set of random steps from one vertex to a randomly selected adjacent vertex [10]. Similarly, the random surfer can be thought of as following random links from one page to another. If a loop is encountered, the user will jump to a new page, instead of looping through the same web pages again and again.

Random walks are a special case of discrete time Markov chains. Markov chains describe systems in which it is not possible to predict future states based on the past, but provided that the current state is known, it is possible to know the probability with which the system will be at a given state at the next observation time [1].

The probability of moving from a given state to another state in such a system is called transition probability. Transition probabilities can be expressed as a transition matrix. Assuming that P is a square matrix in which rows represent the source states and columns represent destination states (or web pages in this case), then the element P_{ij} in the matrix represents the probability of moving to state j, when one is currently in state i.

The probabilities of moving from one state to another at any given time will obey to the following rule: if there is a link from i to j, then $p_{(i,j)} = \frac{1}{o(i)}$, otherwise $p_{(i,j)} = 0$, where $o(i)$ is the out-degree of page i. Elements in each row in the transition matrix add up to 1 and for this reason this transition matrix is called stochastic. It should be noted that this matrix is sparse – i.e. most entries in each row are equal to zero, as most web pages are not linked to each other.

However, it should be noted that the matrix derived directly from the web graph is not necessarily stochastic as there are pages with no outgoing links, in which cases the sum of the elements in a row is zero.

In Markov chains, at any one time there exists a state vector, each element in the vector indicating the probability that the system is at the corresponding state at that specific time [1]. This state vector is derived from the equation

$\pi^{T+1} = P\pi^{T}$ **Equation 2**

The property of Markov chains that is of interest here is that of ergodicity. First we need to explain the conditions under which a discrete-time Markov chain is ergodic. There are two conditions – one is that the Markov chain is irreducible and the other that it is aperiodic. Each of these terms will be explained in turn.

A Markov chain is irreducible if any possible state is reachable from every other state in the system. Concerning periodicity, "a state i has period d if, given that $X_0 = i$, we can have $X_N = i$ when n is a multiple of d" (Gerrard, 2003). A state is called periodic, if $d > 1$.

According to the ergodicity property, "if the discrete-time Markov chain X is ergodic, with transition matrix P, then there is exactly one probability vector π, which satisfies $\pi^T = P\pi^T$" [11].

Specifically, as $T \to \infty$ (i.e. the time component tends to infinity) it holds that $\pi^T \to \pi$ (π being the stationary distribution of the chain or the steady state vector) and this is independent of the values of the original vector π^0. The stationary distribution can be calculated using an iterative procedure, based on equation 2 stated above, by repeatedly substituting π^{T} with the latest vector obtained.

Another issue that needs to be qualified is that of the equation $\pi = P\pi$ that results as $T \to \infty$. In this equation, π is really the eigenvector of the matrix P. As for stochastic transition matrices it holds that the eigenvalue $\lambda = 1$, the equation takes the form $P\pi = \pi$ and can be solved using an iterative solution as mentioned above.

Based on what has been discussed in this section then, the solution to the Pagerank algorithm is to use some iterative method in order to find the stable state Pagerank vector, which will be the eigenvector of the matrix derived from the web graph.

However, certain procedures need to be followed before this method can be applied to the web graph. This is because the matrix deriving from the web graph is neither stochastic in contrast to proper transition matrices, as some pages in the web graph don't have any outgoing links, nor irreducible as not all pages in the web graph are linked with each other.

In the next section, the derivation of the actual Pagerank equation will be discussed and linked to the issues discussed in the present section regarding matrix properties.

2.4. Derivation of the Pagerank Algorithm

Before proceeding it should be noted that the notation used throughout this has been based on the notation that Haveliwala et al. used in the paper 'Computing Pagerank using Power Extrapolation'. In addition, even though the idea is to present the Pagerank as originally proposed by Page & Brin (1998), certain explanatory elements have been borrowed from other papers, which have also presented the algorithm, as the original Page & Brin paper is not always clear as to the exact procedure followed to derive the algorithm.

One problem that needs to be handled in the original web graph is that of dangling links. Dangling links are defined as "links that point to any page with no outgoing links" [17; p.6]. The problem with dangling links is that the system does not know how their weight should be distributed and Pagerank might be overestimated for these links at the expense of other pages. Even though the exact procedure followed in the original paper concerning dangling links is not very clear, the idea is that they remove these links from the system until the calculations have converged and then add them back to the system.

However, as Langville & Meyer (2004) discuss, this approach is not necessarily the ideal one, as certain dangling links do actually deserve to get a high ranking, and also because they consider that taking these pages temporarily out of the system introduces bias into the system (p.7).

What has been proposed as a solution to the dangling link problem is introducing artificial links within the graph, so that all states in the matrix have at least one outgoing transition – i.e. each row in the matrix has at least one non-zero element [15].

For this to be achieved a new matrix D is first defined as follows. If n is the number of pages in the web graph, then d is an n-dimensional vector, in which if $o(i) = 0$, then $d_{(i)} = 1$, and in any other case $d_{(i)} = 0$.

Based on this vector, the matrix D is defined as $D = dv^T$, where v^T (the transpose of the n-dimensional column vector v) is in its simplified form a uniform probability vector whose elements are equal to $v_{(i)} = \frac{1}{n}$.

The new transition matrix is subsequently defined as

$P' = P + D$ **Equation 3**

In other words, the effect of matrix D is to allow the random surfer to jump to a new page every time a dangling page (i.e. a page with no outgoing links) is visited using the distribution of v^T [12].

It should be noted that vector v^T does not need to have a uniform distribution, and could vary depending on the type of page or some other criterion. For this reason this vector is also referred to as personalization vector, and is one of the parameters in the Pagerank algorithm that can be manipulated.

Another important problem, in some respects parallel to the problem of dangling links, is that of rank sink [17]. Assume that there is a group of pages that contains several links pointing to pages within that group, but no links to other pages outside the group. In this case, once someone enters this group of pages, it will be impossible to escape and so the result will be looping through this set of pages repeatedly. The effect of such groups of pages on calculations is that these pages receive higher Pagerank values than expected. Rank sink can be dealt with by actually forcing the matrix to be irreducible as will be shown below.

A further matrix (E) is constructed for this purpose, which is defined as

$E = [1]_{n*1} v^T$ **Equation 4**

and is a fully connected matrix (every state can be reached from every other state in the matrix).

The final vector P' is defined based on E as

$P'' = cP' + (1-c)E$ **Equation 5**

P'' being at this stage both irreducible and aperiodic (as self-loops are also added).

In this equation c is a damping factor, which is typically set to 0.85 and indicates the probabilities with which the random surfer will follow a link from the page he's currently at, rather than jump to a random page within the system.

Concerning matrix E, what it signifies in terms of the random surfer model, is the probabilities with which at any arbitrary time the user will make a random jump to a new page instead of following the links in the page he is currently in [12]. It should be noted that this matrix is referred to as the teleportation matrix.

To sum up then, after the modifications made to the original simplified Pagerank equation so that it is can converge to a single solution, we have derived equation 5, where P'' is at this stage a dense matrix.

Having shown in detail how the Pagerank equation is derived, a summary of the outcomes of this section will be presented before proceeding to implementation issues regarding Pagerank.

Basically for any page i that belongs to the web graph, the Pagerank value $R(i)$ will be:

$$R(i) = (1-c)*\frac{1}{n} + c*\sum_{\forall(inlinks(j))} \frac{R(j)}{o(j)}$$

Equation 6

It should be noted that in their paper 'Anatomy of a Large-Scale Hypertextual Web Search Engine', Page and Brin also present the following formula

$$R(i) = (1-c) + c*\sum_{\forall(inlinks(j))} \frac{R(j)}{o(j)}$$

Equation 7

which according to [19], also converges to a single solution and most importantly, both equations preserve the same order of Pagerank values. So in this context, these are both presented as acceptable definitions of Pagerank.

3. Computation of Pagerank

3.1. Centralized Implementation

In the introductory section on the algorithm it was demonstrated that in order to calculate the Pagerank values for a given web graph, one needs to basically find the principal eigenvector of the transition matrix resulting from that web graph.

The basic procedure for finding the principal eigenvector, also followed by Page and Brin in their seminal paper, is based on the Power Method, and is formally defined as follows:

$R_0 \leftarrow S$

loop:

$$R_{i+1} \leftarrow AR_i$$
$$d \leftarrow \|R_i\|_1 - \|R_{i+1}\|_1$$
$$R_{i+1} \leftarrow R_{i+1} + dE$$
$$\delta \leftarrow \|R_{i+1} - R_i\|_1$$

while $\delta > \varepsilon$

where A is the transition matrix, and d is used for normalization (maintaining the value of $\|R_i\|$), which won't be discussed in this context [17].

The Power method has the advantage that is does not require the matrix to be formed explicitly, as it can be performed on the sparse matrix (as demonstrated by [15]). This is important as Pagerank calculations are performed on web graphs that contain billions of nodes. Storing a dense matrix of this magnitude as well as manipulating it during calculations would impose constraints both in terms of storage and memory capacity.

In addition, as can be seen above, with the Power Method it is necessary to only store the current vector at any one time thus again limiting the storage requirements [15]. Finally, as Page & Brin showed in their original experiments, the Power Method converges quickly and also scales well as the size of the web graph increases. Specifically, they based their original paper on a web graph consisting of 24 million pages (322 million links). The Pagerank values converged after 52 iterations, and what's more important, when only half of the data were used, convergence was reached after 45 iterations (see Figure 2). This was considered as strong evidence that the system is potentially scalable to even larger datasets.

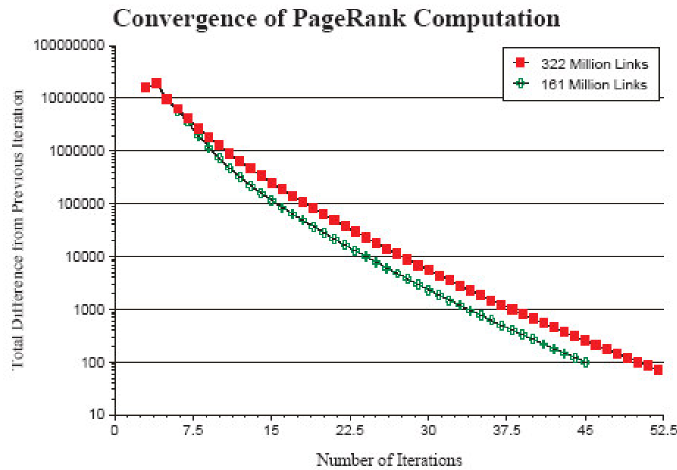

Figure 2. Diagram illustrating rate of convergence of Pagerank values with datasets of different size (Page & Brin, 1998, p.8).

The implementation proposed by [17] and which has been used at the Google search engine at least until some time ago is centralized. Basically, the web is crawled at regular time intervals (e.g. once per month), and then Pagerank calculations are performed from scratch on the new web graph resulting from the latest crawl.

These calculations take a long time (in the magnitude of days) to complete, as the web graph currently considered by Google has an estimated size of 4 billion pages. The major problem with this approach is that it is not possible to incorporate new pages while calculations are taking place and so the system is not responsive to dynamic changes in the web. In addition, the computations require huge clusters of machines, as they are both memory and CPU intensive and these requirements increase with the size of the crawled web.

This has led a number of researchers to look for an alternative implementation of the algorithm that will be able to accommodate dynamic changes in the web graph, while it will also distribute the load of the computations in some way that will reduce the need for these huge clusters. A number of distributed implementations of the Pagerank algorithm have been proposed which will be discussed in detail in the following sections.

3.2. Distributed Pagerank

Two alternative proposals concerning the implementation of a distributed Pagerank algorithm have been proposed in literature. One option is to implement the algorithm as a service within web servers on an Internet scale, while the other option is to implement it within a peer-to-peer framework.

3.2.1. Implementation as a web server function

A simple implementation of the algorithm as a web server function has been briefly discussed by Sankaralingam et al. (2003). The idea is that each web server calculates the Pagerank values of the pages it is responsible for, and then passes on the updated values to the appropriate web servers, depending on the outgoing links of the local pages.

Wang & DeWitt (2004) proposed a more complex system, which consists of two separate measures. Each web server produces a vector of its local Pagerank values, called Local Pagerank vector, which is based on the intra-server linkage information. According to the authors, this measure is likely to be very similar to the standard Pagerank values, as the majority of links in a web page are typically intra-server links.

In addition, there is a measure of the relative importance of each web server, called ServerRank, which is obtained by exchanging inter-server linkage information between web servers. The final ranking, which is calculated by the server handling the user request, is obtained by ranking the results according to both Local Pagerank values and ServerRank values.

The advantage of the web server approach is that the calculations are incorporated into the existing web servers, and are spread throughout the Internet, rather than being catered by dedicated clusters of servers. In addition, the web server approach naturally takes advantage of the fact that the majority of links from a page point to pages within the same site [6]. This means that communication overhead is reduced, as most updates happen locally, within the web server.

3.2.2. Implementation within a peer-to-peer framework

As already mentioned an alternative implementation of a distributed Pagerank algorithm is within a peer-to-peer framework, where each peer is responsible for a set of pages, and update messages are routed between peers using the underlying framework.

For example, Sankaralingam et al. (2003) discussed the integration of the algorithm in existing peer-to-peer systems as a means of ranking documents in those systems and improving the efficiency of queries. Specifically, they focused on keyword search in DHT (Distributed Hash Table) based systems. The authors suggested storing along with the pointers to documents containing a given keyword, the Pagerank values of those documents, in order to return the most popular entries during a query and thus reduce network traffic.

3.2.3. Considerations in a distributed system

Irrespective of the implementation chosen, there is a need to adjust the standard Pagerank algorithm presented in previous sections for use in a distributed system. Most importantly, in a distributed system, the web graph won't be located at a single machine. In that respect, to perform matrix-vector multiplication would require passing down to other machines partial multiplication results or requesting the required elements from other nodes. Such an approach would incur great communication overheads between nodes. In addition, in the standard algorithm, for each iteration to take place, information from the previous iteration is required. As a result, some kind of global synchronization between participating nodes would be necessary. For these reasons alternative solutions have been proposed, which are discussed in the following sections.

3.3. Chaotic Iterations Method

Sankaralingam et al. (2003) proposed an implementation of a distributed Pagerank algorithm based on the chaotic (asynchronous) iterative solution of linear systems [5]. As Strikwerda [24; p.1] states, "in chaotic relaxation the order in which components of the solution are updated is arbitrary and the past values of components that are used in the updates are also selected arbitrarily".

In such an implementation, there is no need for global synchronization of the different processors and no need for central control. Sankaralingam et al. (2003) implemented this

algorithm in a peer-to-peer context, though they did also discuss its implementation as a function of web servers.

One of the important strengths of this approach is that it allows incremental computation of Pagerank values as new pages enter the system. This way it deals with the main limitation of the centralized implementation, where updates occur only in specified time intervals and require re-computation of all Pagerank values in the system.

In simple terms, the idea of the Sankaralingam et al. (2003) algorithm is that after an initialization stage, each peer starts calculating the Pagerank value for every document they hold. If the difference between previous and current Pagerank values is bigger than some threshold error value, then update messages are sent to the outgoing links of that page, thus cascading the update process. Every time a new update message for a given page is received by a peer, the Pagerank value for that page is recalculated and depending on whether the convergence condition has been met or not, further updates may be generated. This process continues as long as more update messages are received, causing more cascading updates.

The system developed by Sankaralingam et al. (2003) has also been designed in such a way as to handle dynamic features such as peer arrivals and departures, as well as page inserts and deletions. Specifically, when a peer goes offline, any update messages that were targeted at it are stored at the sender and resent at regular time intervals until they are successfully delivered. Concerning new page insertions, new pages are initialized with some Pagerank value, and they start sending update message to their outgoing links, thus being incorporated into the existing system. Finally, concerning page deletions, update messages with the respective Pagerank values that should be subtracted are sent out to the outgoing links of the pages to be removed.

3.3.1. Implementation evaluation

The algorithm by Sankaralingam et al. (2003) has a discrepancy with the Page and Brin algorithm that has to do with the convergence criterion used. In contrast to Page and Brin who used absolute error as a convergence criterion (i.e. $(oldvalue - newvalue)$), Sankaralingam used relative error (i.e. $abs(oldrank - newrank) / newrank)$ as their convergence criterion. The issue with the latter is that it is will cause the algorithm to converge much faster than it will using absolute error. This discrepancy should be kept in mind when evaluating the simulation results from the Sankaralingam et al. paper.

In addition, in the Sankaralingam et al. (2003) implementation, there seems to be some kind of synchronization going on between peers. Specifically, the authors mention that computation of Pagerank values occurs concurrently by all peers. This if followed by the exchange of update messages, after the completion of which the next iteration is initiated concurrently at all peers. The authors describe this exchange of update messages as a 'pass'. The computation is said to converge once all documents within a peer have a Pagerank value that does not exceed the error threshold when compared to the value at the previous iteration.

This description is against the idea of asynchronous computation. First of all, when using the asynchronous method, there is no need to make sure that all peers compute or send messages concurrently. In addition, there is no such thing as a clearly defined iteration, as updates can occur at any time, and for only a small subset of the pages held within a peer. In other words, in such a system, each page may converge at a different time, in which case its Pagerank value won't be recalculated anymore, unless it receives a new update message. In that sense, it is not possible to speak about iterations when a method like chaotic relaxation is used.

It could be the case that the authors were trying to make a close comparison between a centralized version of the algorithm and the distributed version they proposed, and for this reason they decided to follow such strictly controlled procedures even though this is not clearly stated within the paper.

3.3.2 Testing Evaluation

In their testing procedure Sankaralingam used 500 peers and four different web graph sizes (10000, 100000, 500000 and 5 million nodes each).

In Table 1 a general idea of the number of passes (as defined above) required for convergence, with varying graph size and peer availability can be seen. The picture from this table is that as the increase in number of passes is not dramatic as the graph size increases. For example, the difference in number of passes for a graph size of 10000 and a graph size of 5 million is 50, which was considered by the authors as evidence of the efficiency and scalability of the system.

Graph size (in 1000s)	# of passes
10	74
100	88
500	118
5000	120

Table 1. Convergence rate (measured in number of passes) for 500-peer simulation, with an error threshold $\varepsilon = 10^{-4}$ (adapted from Sankaralingam et al., 2003, p. 5).

A synchronous centralized implementation of the Pagerank algorithm was used in parallel in order to compare accuracy of the values obtained. It appears that the values obtained using the distributed algorithm were highly accurate even for higher threshold values (the highest threshold used was 0.2; average relative error values were in all cases in the magnitude of 10^{-3} or less).

Other measures taken during the simulation include number of update messages generated during the computations, message traffic generated during document insertion, as well as execution time. These results will not be documented in detail here, but rather will be discussed together with the results obtained in the present implementation. The reason is that the implementation in this project has been largely based on this paper, and thus it will be more informative to present all findings together in the results section of the report as there will be greater scope for comparisons and discussion.

The authors estimated the execution time in a web server environment, assuming a T3 line network connection between web servers, to be approximately one month for a web graph consisting of 3 billion nodes, with a convergence threshold of 10^{-3}. Even though this estimation is very similar to the centralized version of the algorithm, the distributed implementation has the advantage that it does not require re-calculation of values from scratch when new pages are added to the system, or pages are removed from it [22].

Overall, the experimental results suggest that the algorithm is potentially scalable to large data sets. It has the advantage that it does not require global re-computation of Pagerank values when new documents are inserted as these are naturally incorporated into the system. In addition, except from handling new page insertions, the system can handle other dynamic features as well, including peer arrivals and departures, which are commonplace in any large-scale system. Overall, this paper forms a very good basis for further investigations in the area of distributed Pagerank algorithm implementations.

3.4. Open-System Pagerank

Shi et al. (2003) proposed the idea of open-system Pagerank. According to them the standard Pagerank algorithm considers pages in the web graph as a closed system. In order for distributed computations to take place, the pages in the web graph need to be seen as an open system in which pages in different machines can communicate with each other.

The difference from the traditional algorithm [17] is that Shi et al. (2003) also incorporate a vector Y, which is the vector of ranks to be the sent to other nodes. Y is defined as $Y = BR$, where $B_{uv} = \beta / d(u)$ (β being the equivalent of $(1-c)$) if $d(u) > 0$ and $B_{u,v} = 0$ otherwise. The altered Pagerank algorithm can be seen is Table 2.

Pagerank for Open-Systems
$GroupPageRank(R_o, X)$ repeat $R_{i+1} = AR_i + \beta E + X$ $\delta = \lVert R_{i+1} - R_i \rVert_i$ until $\delta > \varepsilon$ Return R_i
Notation used: A is the original sparse matrix, E is the personalization vector, and X is the vector of ranks received from afferent links.

Table 2. Pagerank definition for open systems by Shi et al. (2003).

Based on this algorithm, Shi et al. (2003) proposed two versions of a distributed Pagerank algorithm, which are outlined in the following table. Both algorithms share the same basic idea. Before the new local Pagerank vector is calculated, vector X is refreshed as other nodes in the system may have sent updates. Subsequently, the new local Pagerank vector R is calculated, and an updated vector Y is generated which is transmitted to the other nodes in the system.

Distributed Algorithm 1	Distributed Algorithm 2

$R_0 = S$
$X = 0$
loop
    $X_{i+1} = refresh$ X
    $R_{i+1} = GroupPageRank(R_1, X_{i+1})$
    Compute Y_{i+1} and send to other nodes
    Wait for some time
while true;
	$R_0 = S$
	$X = 0$
	loop
	    $X_{i+1} = refresh$ X
	    $R_{i+1} = AR_i + \beta E + X_{i+1}$
	    Compute Y_{i+1} and send to other nodes
	    Wait for some time
	while true;

Table 3. Distributed Algorithms proposed by Shi et al. (2003).

Both of these algorithms are designed to be executed asynchronously by the different nodes participating in the system, so again there is no need for synchronization. The difference between the two algorithms lies in the frequency with which a node receives updates from other nodes and sends updates to other nodes within the system.

In the first algorithm, every time a recalculation of vector R takes place, it iterates using the algorithm in Table 2 until it converges, at which point updates are sent, and the updates received are incorporated. In the second algorithm, the node always uses the most recent vector received from external nodes, and sends the updated vector Y as soon as a single iteration has taken place. Thus this algorithm always uses up-to-date values but it achieves this at the expense of increased network traffic.

These algorithms seem follow a different strategy than the algorithm presented in the previous section. First of all, they are not based on chaotic relaxation. Each page is not considered as an independent entity, in the sense that calculation does not occur on a page scale when updates are received; rather all calculations seem to take place in a vector scale (where a vector contains all the entries in the local node). In that sense, there is actually a notion of iteration involved, as re-calculations occur on a node scale.

3.4.1. Simulation Evaluation

For simulating the system, Shi et al. (2003) used a graph with 1 million pages and 15 million links. Asynchrony was simulated by having nodes wait for $T_w(u,m)$ time units before they start a new iteration (i.e. loop). In addition, in order to consider node or network failure, some

nodes failed to send their update vector with probability p. Simulations were run with different values of T , p and different numbers of page rankers.

They also included a comparison between centralized & distributed versions, the results of which can be seen in Figure 3. DPR1 and DPR2 refer to the two versions of the distributed algorithm proposed. As the figure shows, the second distributed algorithm was the most efficient in terms of number of iterations. This makes intuitive sense, as this version of the algorithm always incorporates updated information from external node before proceeding with a new calculation.

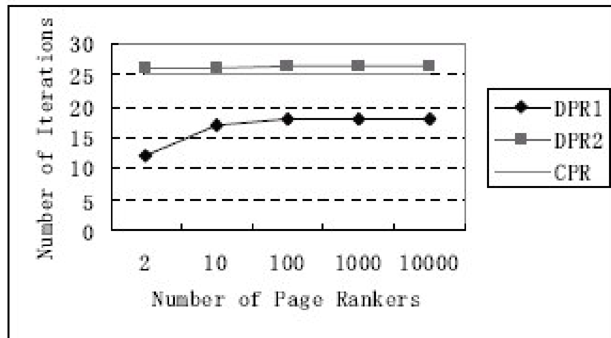

Figure 3. Shi et al. (2003): Number of iterations required for the three different algorithms used.

What is not clear in this figure is how number of iterations is defined. As mentioned, an iteration is defined as a loop in the algorithm. However, this is a local rather than a global measure. So it is not clear in this graph, what exactly the number of iterations refers to.

In general, the description of the implementation of this distributed system is quite brief, and does not always provide a clear definition of the terms used. Specifically, the notion of iteration is not very clearly defined, and thus when it comes to results analysis there is a question about whether the measure of iterations is global, and if so how exactly it is generated.

3.5. Conclusions

In this section, implementation issues concerning the Pagerank algorithm have been discussed. The centralized implementation has been discussed and contrasted with possible distributed implementations of the algorithm. Between the two distributed implementations presented, the one based on chaotic relaxation seems to be defined in more detail, while it is also better grounded on experimental research. This is the implementation that the present project has been based on, and thus will be encountered again in later sections.

As the implementation in this project has been based on a peer-to-peer framework, we will now proceed to discuss the main features of peer-to-peer systems, and existing technologies in this area.

4. Peer-to-peer framework for distributed algorithms

4.1. Overview

Peer-to-peer (P2P) computing has had its share in the field for a number of years, though it has only been popularized as a term since the last 1990s when Napster, a P2P music sharing application was released, which made P2P technology accessible to the wider public.

In P2P networks (figure 4), there is no distinction between client and server. All nodes can act at different times either as a client or as a server, receiving and serving requests, or sending themselves requests to other nodes. In pure P2P architectures, communication and data exchange occurs directly between the different nodes, without any server intervening.

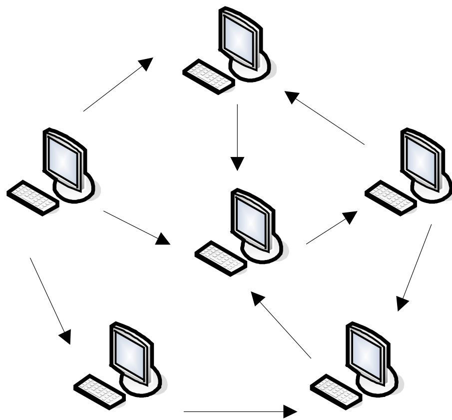

Figure 4. Diagram of the basic peer-to-peer architecture.

4.2. The Peer-to-Peer Software Architecture

It is possible to distinguish broadly between three layers: base overlay layer, middleware layer and application layer [25].

The base overlay layer is the most basic of all three layers, and is the common denominator of all peer-to-peer applications. This layer is responsible for ensuring that nodes running in the system are aware of each other and of any joining nodes and that communication between them runs smoothly.

The middleware layer is not as general as the base overlay layer, though the functionality it provides can be used by different types of applications. For example, distributed hash tables, which are an example of a middleware layer service, are shared by diverse P2P applications.

Finally, the application layer, as its name suggests, is the actual P2P application that provides the user-targeted services. Examples of P2P applications include distributed file storage systems, or file sharing systems. The distributed Pagerank calculator developed for this project operates at this layer.

4.3. Structured versus unstructured P2P Overlays

A useful description of the base overlay layer has been provided by Yang & Molina (p.4), who describes P2P overlays as an undirected graph, in which vertices are the nodes in the network and edges correspond to open connections between these nodes. Broadly speaking there are two alternative implementations of the base overlay layer: unstructured and structured overlays.

4.3.1. Unstructured P2P Overlays

Unstructured overlays pose no constraints in the location of data within the P2P network, nodes are organized in a random graph and search methods used include keeping a centralized directory, flooding or random walks. With flooding, a node receiving a query will forward it to all its neighbors, which in turn forwards it to their neighbors, until some expiry factor (e.g. time-to-live of the query) is reached. A random walk involves forwarding the query to a random neighbor or node at each stage until the expiry factor is reached.

Napster is the most popular example of a P2P network, in which queries for files are performed centrally, and is therefore often referred to as hybrid P2P. Napster makes use of a centralized server, to which users upload automatically their file lists, and which users query in order to determine the existence and location of a specific file. Once they know where they can download a file from, direct communication is then established between these two peers.

On the other hand, Gnutella is an example of a pure P2P, with fully decentralized search facilities. The search method used in its original versions is flooding, with a time-to-live component in the message that ensures that a query won't circulate in the network forever.

Once a file is located, the exchange between the requesting node and the node containing the file is performed directly.

Unstructured overlays make queries involving more than a single keyword possible. If centralized search is used, there is there problem, however, that there is a single point of failure – the server holding the file information, and in addition the scalability of the system is eventually limited by this centralized component.

If flooding or random walks are used, extensive network traffic is generated, especially as the number of nodes increase. In addition, these two techniques don't guarantee that items stored in the network will always be returned in a query, as the original query message may not reach the right nodes due to the time-to-live constraints. It should also be mentioned that the technique of flooding can only work in networks up to some limited size, so the design is not as scalable as it would be desirable.

4.3.2. Structured P2P Overlays

Structured overlays, on the other hand, which were developed to deal with the shortcomings of unstructured overlays, especially in terms of data querying, "impose constraints both on the node graph and on data placement to enable efficient discovery of data" [4]. This makes structured overlays especially efficient and reliable for exact-match queries, though complex queries are currently not so well supported in structured overlays. In their majority structured overlays are self-organizing, fault-tolerant and load balanced [16].

Structured P2P overlays make use of key-based routing (KBR; [8]). Generally speaking, in this type of implementation nodes are assigned unique IDs from a predefined identifier space. Objects are also assigned with unique IDs from the same identifier space, and are mapped according to some overlay-specific algorithm to a single node, which is called the root for that key. This node is required to keep either a copy of this object or a pointer to the location in which this object is stored.

Each node maintains a routing table in order to achieve routing of a key to its root. Every time key is received by a node, the node checks whether it is the root for that key, and if not forwards the key to a node that is progressively closer to the root node, based on its routing table. This process is repeated until the root node is reached [8].

Different structured overlays differ in how they organize their identifier space and also in what algorithm they use for mapping keys to nodes. For example Chord [8] and Pastry [18] use a circular identifier space, while CAN (Content Addressable Network; [20]) uses a d-dimensional identifier space. The majority of these implementations provide mechanisms for dealing with node arrivals and departures and also provide performance guarantees in terms of number of hops for locating a given object. Of course maintenance costs are higher for structured overlays as they need to keep the tables of nodes updated as the system changes dynamically.

4.4. Structured P2P Overlays: Pastry

In this section, a detailed overview of the features of Pastry – a structured P2P Overlay developed by Microsoft Research in Cambridge and Rice University, will be provided. It was decided to focus more exhaustively on one implementation rather than present a rather superficial summary of multiple implementations, in order to cover as many of the available features as possible. As Pastry is the system used for the purposes of this project, it is the one chosen to be analyzed in depth.

4.4.1. Identifiers and identifier space

Pastry relies on a circular identifier space, which in the latest version is also 160-bit long (originally it was 128-bit long). Nodes and objects are identified using 160-bit nodeIDs and objectsIDs respectively. Identifiers can be thought of as numerical sequences in the base of $2*b$ (b is a configuration parameter, typically equal to 4).

4.4.2. Routing

An object is assigned to the node whose identifier is numerically closest to that object's identifier. Routing relies on two main types of information that each node holds: a routing table and a leaf set. The routing table consists of a number of rows, each consecutive row containing node identifiers (as well as IP addresses) sharing one more digit in common with the local node. Each row consists of (2b-1) entries, one for each possible digit in the next position of the identifier.

The leaf set can be though of as a smaller version of the routing table, which hold information about the nodes with the numerically closest smaller and larger identifiers compared to the

local node's identifier. The aim is to increase the efficiency of routing, by trying to by-pass the routing table and is the first table that is used during routing.

Every time a node receives a key, it checks whether it is the root for its identifier. If not, it checks whether the key can be routed based on the entries in the leaf set. If so the key is forwarded to the node corresponding to the relevant leaf set entry, in which case a single hop is required.

If not, the routing table needs to be consulted for forwarding the message. At every hop, the key is forwarded to a node that either shares a longer prefix with it, or shares a prefix of the same length with it, but is numerically closer. This guarantees that routing will eventually converge.

4.4.3. Routing performance

According to Rowstron & Druschel (2001), "the expected number of routing steps is [$\log_{2b}N$] steps, assuming accurate routing tables and no recent node failures" (p.6). In addition, assuming that many nodes fail at the same time, "the number of routing steps required may be at worse linear in N, while the nodes are updating their state" (p.6). Pastry then, provides highly satisfactory routing performance, even at adverse conditions.

4.4.4. Node arrivals and departures

Pastry allows nodes to self-organize and adapt to node arrivals and departures, thus being a system that is responsive to dynamic changes, which are a major part of P2P networks.

When a new node joins the system a join request is routed. Based on the outcome of this request, the node initializes its state tables, i.e. the routing table and the leaf set. This information is sent to the nodes contained in its state tables. In this way, both the new node and the existing nodes have up-to-date information in their state tables.

In a P2P system, nodes don't always depart in an orderly manner informing other nodes of their departure. Different procedures are used for keeping the state tables up-to-date. These procedures won't be described here in detail, as they are beyond the scope of this paper.

However what should be mentioned is that the likelihood of failure in the updates in considerably low.

4.4.5. Locality

An important feature of Pastry is that it takes into account physical locality of nodes based on some proximity metric, such as number of IP routing hops or geographic distance.

Proximity is considered when building the routing table for a given node, the aim being to maintain entries in the table that not only satisfy the relevant prefix requirements but are also closer to that node according to the chosen proximity metric. In addition, at each routing step, care is taken to forward the message to the destination node trying to minimize the distance traveled.

4.5. Conclusions

Overall, Pastry provides all the basic functionalities of a structured overlay. Locality awareness is one of the features that distinguish Pastry from other overlays such as Chord, and gains increasing importance with the size of the system considered.

More generally, structured overlays provide a highly satisfactory framework for building peer-to-peer applications. They provide efficient and scalable routing and lookup facilities, are self-organizing - dealing effectively with peer arrivals and departures, and are particularly well suited for performing reliable searching as they conform to a specific graph structure. Of course these features require a higher maintenance costs than unstructured overlays

Structured overlays seem particularly suited as an infrastructure for the distributed Pagerank calculator. Assuming that pages are distributed among peers based on their hash value, the underlying structured overlay can then take care of determining which peer is responsible for a given page and route messages to the appropriate peer.

Having provided a general overview of peer-to-peer technologies and structured overlays, which are the technology of choice in this project for the reasons discussed, we now move on to present the design of the system developed.

5. System Design

5.1. Algorithm Overview

The distributed algorithm implemented in this project has been largely based on the algorithm presented in the Sankaralingam et al. (2003) paper. The main reason for choosing this implementation over the implementation suggested by Shi et al. (2003) was that the latter, while being distributed, still had some synchronization requirements. In contrast the Sankaralingam et al. (2003) proposal based on chaotic relaxation can be implemented in a completely asynchronous system, even though as discussed previously, the authors still used some form of synchronization in their implementation, which will not be included in the present system.

In this section, the algorithm used will be outlined and the points of divergence from the original version will be outlined. In addition the basic issues that have been considered during the design stage will be discussed.

Details concerning how the web graph was obtained, and how pages were distributed among nodes will be dealt with in the implementation section. For now, it will be assumed that a web graph exists, which has been partitioned among the available peers using the underlying structured overlay, so that each participating peer is responsible for a subsection of this web graph.

The formula used in the present system is the one defined in equation 7, which is restated below.

$$R(i) = (1 - c) + c * \sum_{\forall (inlinks(j))} \frac{R(j)}{o(j)}$$

This formula was chosen as it has been used more extensively in previous papers, and also it is the formula that the Sankaralingam et al. (2003) paper has used.

In contrast to [22], in this context, the convergence condition used is absolute rather than relative difference, so to restate

$$absdiff = abs(oldrank - newrank)$$ **Equation 8**

In order to determine whether the Pagerank value for a given page has converged, the *absdiff* value is compared to the error threshold value epsilon. If $absdiff < epsilon$ then the Pagerank value is considered to have converged. Epsilon can be set to different values (e.g. 10^{-2}, 10^{-4} and so on) depending on the desired accuracy. It should be noted that there is a trade-off between the epsilon value chosen and the message traffic generated until convergence.

1. Each peer runs a Pagerank Calculator client. 2. Once the application is started, the peer goes through each of the pages it is responsible for, so that the process of Pagerank calculation is initiated. 3. If the Pagerank value for a given page has not converged, any local outgoing links are updated, and update messages are sent to outgoing links that reside in other nodes. 4. Once an update is received (locally or remotely), the Pagerank value for that page is recalculated. If the convergence condition is not met, further updates are generated, again both locally and remotely, depending on where the outgoing links from that page are residing. 5. Recalculation of Pagerank and generation of further update messages continues until the Pagerank value for all pages has converged and no more messages are generated in the system.

Table 4. Outline of distributed Pagerank algorithm.

Step 2 is necessary in order to ensure that the system is initialized in some way, and that all pages residing in a node get the chance to propagate their Pagerank value (e.g. in the case of pages with no incoming links).

In contrast to the algorithm outlined by Sankaralingam et al. (2003), the algorithm presented here does not introduce any type of synchronization in terms of when nodes start sending update messages, and there is no concept of iteration involved.

5.2. Design issues

5.2.1. Linkage Information

An important decision is what information should be held about each page in a node. Concerning linkage information, it was decided to only hold a vector of outgoing links per page, so that it is possible to keep track of where update messages need to be sent. Incoming links for a given page are not stored, as this would be very memory intensive especially for very popular pages with large numbers of incoming links. In addition, considering that the web graph will potentially be split among a considerable number of nodes, it is not possible to know with certainty all of the incoming links for a given page.

5.2.2. Structure of Updates

Having considered what linkage information is saved for a given page, we now need to consider what update information will be transmitted from the source page to its outgoing links. As mentioned in the section analyzing the Pagerank algorithm, the information sent from a given page to each its outgoing links is $PRToPass = PRValue(u)/o(u)$ where $o(u)$ is the out-degree for an arbitrary page u.

As equation 7 indicates, the part of the equation that changes between consecutive calculations is the sum part. As no information is saved about the incoming links of each page in the system, it is not feasible to send the entire *PRToPass* every time a new update message is sent, as the receiving page does not keep track of the pages that have sent updates, so as to cancel out a previous update and add the new value received. Such an approach would require extra memory (for storing these values) and computation (for finding the previous value received, and swapping it with the new one) and would thus end up being very inefficient.

Instead the approach followed is to keep within the object holding information about a given URL a variable corresponding to the Pagerank value at the previous calculation. *PRToPass* is redefined then as the difference between the newly calculated Pagerank value and the Pagerank value at the previous calculation time, divided by the number of outgoing links.

$$PRToPass(u) = (PRValue(u) - OldPRValue(u)) / o(u) \qquad \textbf{Equation 9}$$

When a page receives an update message then, what it does is add the value received to the sum part of equation 7 and recalculate its Pagerank value.

5.2.3. Updates Frequency

Another design issue is related to controlling the traffic that will be generated by update messages. As in a system dealing with a large web graph, significant traffic will inevitably be generated, it is important to develop techniques for controlling this traffic to some extent.

In the present context, the procedure followed is to wait until the difference between the newly calculated *PRValue* and the *OldPRValue* is bigger than some threshold value, before an update message is sent. In addition, in order to avoid a deadlock situation, where all pages are waiting for each other to send update messages, an expiry mechanism is incorporated to ensure that update messages are eventually sent.

5.2.4. Dynamic features

Finally, the system has been designed to handle page insertions, as well as node arrivals and departures. Whenever an update message is sent that is not successfully delivered, it is stored at the sender, which retransmits it at regular time intervals until it is eventually delivered. This approach enables dealing with updates aimed at peers that are temporarily offline, as well as updates aimed at pages that have not yet been incorporated into the system, without causing any distortion to the resulting Pagerank values. So, when a new page is inserted into the system, it starts sending updates to its outgoing links, and also receives any retransmissions from existing pages linking to it. In this way it is incorporated into the existing system.

6. Implementation: General issues

The implementation phase of this project was divided in two stages. The first stage involved the development of a simulated distributed Pagerank calculator, which works as a standalone application, while the second stage involved the development of the actual distributed application using a peer-to-peer framework. As the simulation serves as a baseline for comparing accuracy of results in the distributed version, the basic design has been kept as similar as possible in both applications, with some inevitable application-specific differences.

The implementation discussion will be divided in three chapters. This section deals with basic implementation issues such as programming language and P2P overlay system choice. The specifics of each implementation are discussed in the next two chapters.

6.1. Programming language & P2P Overlay choice

The choice of programming language was largely based on the choice of P2P overlay for the peer-to-peer part of the project. The P2P overlay chosen for this project was Pastry, which has already been discussed in the background section. Pastry has been developed in Java and makes use of Java RMI (Remote Method Invocation) for invoking methods between nodes.

Chord was the alternative P2P overlay considered, which offers similar functionality to Pastry and has been written in C. It was decided to choose Pastry rather than Chord, because at the time that the research was being conducted there was no official release of Chord. As an official release of Pastry was available it was considered a safer options. In addition, the documentation that was supplied with Pastry provided a good basis for understanding the hierarchy of and the message passing mechanisms within the system.

As Pastry was developed in Java, it was decided to develop the simulation program in Java as well, in order to keep the two applications as similar as possible. It should also be acknowledged that an existing file-sharing application (PAST; [9]) that has been built on top of Pastry was used in the initial stages of development as a guideline for implementing the message structure and hierarchies within the Pagerank application.

6.2. Web graph & Link Extraction

The web graphs used for testing the two applications developed were taken from the WebGraph framework for studying the web [27], developed at the University of Milan.

This project has made publicly available data sets from a few major crawls of the .uk and .it domains, stored in the form of compressed graphs. Within this framework, .graph files contain the compressed web graph, while .urls files contain the corresponding URLs for the entries in the .graph file.

As the data sets provided consisted of millions of nodes, it was decided to extract from those graphs specific domains (e.g. imperial.ac.uk) of different sizes and test the system on those domains.

Within the WebGraph framework, certain classes are provided that can be used to extract specific information from existing data sets, and create new web graphs. The way specific domains were extracted was by determining the node ranges corresponding to a given domain within the .graph file based on the .urls file (where the ith line in a .urls file corresponds to the ith node in the .graph file), and then extracting the information relating to those nodes to some predefined file format.

The linkage information for any page belonging to the selected domain was stored in a .txt file format, rather than a smaller .graph file type, in order to avoid dependencies on the WebGraph framework libraries in the peer-to-peer context. This was achieved at the expense of compression provided by .graph files, but it was decided that considering the scale of this project, this was a reasonable compromise.

6.3. Format of extracted graph file

The format of the .txt files containing the graph information is space delimited, the first entry within a row containing the code of the source page, while the remaining entries within that row representing the codes of the outgoing links from that page.

5001	5002	5003	
5002			
5003	5001	5004	
5004	5008	5002	5006
…			

Table 5. Format of .txt file for storing web graph linkage information

7. Implementation: Part A - Simulation of a Distributed Pagerank Calculator

The idea behind developing the simulation application was to experiment with the algorithm and its convergence behaviour in a simplified environment, without dealing with communication issues involved in a real peer-to-peer environment. In addition it was considered as a baseline measure against which Pagerank values obtained in the distributed system could be compared.

The simulation environment was kept as simple as possible. Most importantly, it was decided not to have different nodes running concurrently in a multi-threaded fashion. Instead it was decided to have nodes perform Pagerank calculations serially. As nodes do not exist in such a system as independent entities, some kind of central manager passing control to different nodes was introduced, in order to ensure proper flow of control.

In the simulation version there is still a concept of iteration or pass – an iteration being defined as a single loop through all active nodes in the system.

7.1. Graphical user interface

A front end was developed for the simulation application, in order to allow users to choose the input file, as well as manipulate factors such the epsilon value, the as number of hosts in the system, as well as activate and inactivate hosts according to their preferences. The main page of the interface can be seen in figure 5.

There are options for choosing the number of nodes (or hosts), as well as setting the error threshold value (epsilon), which are defined before dividing pages among the hosts and setting up each host. The *Single Iteration* option at the bottom right hand side allows the user to step through the calculation pass-by-pass, while the *Full Iterations* option performs the whole set of calculations until convergence. Finally, there is an option (*Add pages*) for adding pages to the existing web graph at a later time, which simulates dynamic page insertion into the system.

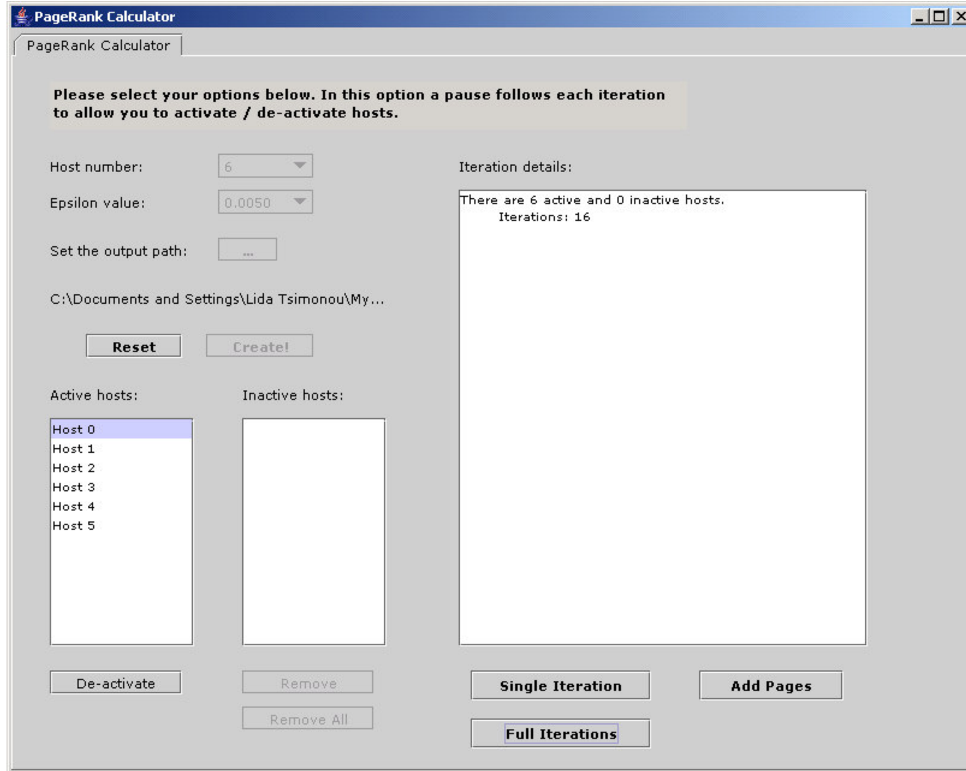

Figure 5. Interface of the application simulating the distributed Pagerank algorithm.

7.2. Package Structure

The classes comprising the simulation application have been outlined in the table below and will each be discussed in turn. It should be briefly noted that the MainFrame class is the class controlling the front-end as well as handling user input and program output and will not be discussed any further in this context.

Package		Principal Classes
graph		MainFrame.class
	operators	GraphCreator.class HostManager.class NodesOperator.class
	parser	Parser.class
	support	Host.class PRPassing.class URLInfo.class URLLinks.class

Table 6. Package Structure for the Simulation Application

7.3. Web page representation

Before going into details about how the application has been implemented, it should be briefly explained how a web page is represented within the system. The URLInfo class is responsible for holding information about a given web site. Its main structure can be seen in the table below.

URLInfo
int url_code int host Vector outlinks boolean converged double PRSum double prValue double oldPass

Table 7. Structure of the URLInfo class holding web page information.

The url_code is the unique identifier for a URL and corresponds to the code that was assigned to that page in the original graph crawled by the WebGraph group. Each page is assigned to a host (or node), which also has a unique integer identifier (*host*). The outgoing links from that URL are stored in a vector (*outlinks*), while the boolean *converged* indicates whether the Pagerank value for that page has converged. Finally, the three last variables in the table are related to the actual calculations and will be discussed in the next sections.

7.4. Web graph handling

The application is initialized with a graph file (in the format discussed in section 6.3), which forms the basis of any subsequent handling. In order to represent the information in a graph form, JDSL (Java Data Structures Library; [13]) was used, which provides a package of classes implementing the functionalities of graphs.

The libraries used provide methods for obtaining iterators over the outgoing or incoming links to a given vertex in the graph. Originally, the system was designed to insert directed edges into the graph, and then use this information to extract the linkage information from the graph. However, after implementing this feature at a testing stage it was considered that it is not the optimal solution for one main reason. Considering that the application developed is a simulation of a distributed system, it was thought that having all the information in the form of a large web graph containing all the linkage information would not be consistent with simulating a distributed system.

It was therefore decided to parse linkage information from the input file straight into the *outlinks* vector within a given URLInfo object, rather than first input it into the graph structure. The graph structure was only used then to hold the URLInfo object corresponding to each node, and no linkage information was inserted. In that respect this could be replaced by a vector, but was kept as a graph as it made intuitively more sense.

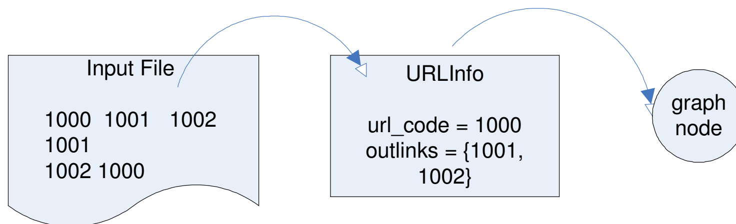

Figure 6. Parsing page information from the input file.

Figure 6 shows the exact procedure followed when parsing information from the input file. The corresponding method is defined within the class graph.parser.Parser. The first element in a row is assigned to the url_code variable, while the remaining elements are added to the outgoing links vector for that page. When a new URLInfo object is created, a new node is added to the graph, which holds this URLInfo object.

It should be noted that the graph.operators.GraphCreator class is basically a wrapper class for the Parser. It is within GraphCreator that the parser is created and executed.

The resulting web graph is then split among the specified number of hosts. A simple heuristic was developed for dividing pages among nodes, and was chosen so that it is possible to determine the node responsible for a given page, knowing only its code and the total number of nodes in the system. This was important for passing information between hosts during the calculations, as there are no routing algorithms in a simulation environment. The short algorithm followed can be seen in the following table.

It should be noted that the same procedure is followed when new pages are added to an existing system, the only difference being that new hosts are not created again, but rather existing hosts are updated by being provided with additional pages.

```
findHost(int value, int numHosts)
{
   while (value%10 > 10)
      {
         value %= 10;
      }
   while (value > numHosts)
   {
      value = value%numHosts;
   }
   return value;
}
```

Table 8. Routine for dividing pages among nodes, where value is the URL code.

This routine achieves relatively even distribution of pages among the existing nodes. Although there are certainly other ways to distribute pages (e.g. using dome digits of the hash code of a URL code), this method was chosen because of its simplicity. This method has been implemented within the graph.operators.NodeOperator class, which is the class responsible for manipulating the nodes within the graph. The resulting value, which corresponds to the host to which a given URL has been assigned to, is stored within the URLInfo object for that URL.

7.5. Simulation of the distributed environment

The graph.operators.HostManager class acts as the glue holding together active and inactive hosts in the system. As there is no way to achieve exchange of updates directly between source and destination host, all updates are done via the HostManager class, which holds a vector of all hosts in the system. The structure of this environment and the form of updates are the topic of this section.

7.5.1. Host Manager: Creating hosts

The basic structure of the host manager class is depicted in the table below. When the user has provided all the input (number of hosts and value of epsilon), an instance of the HostManager class is created, and the setup information is stored within that instance.

HostManager
int numHosts int activeHosts int iterations Vector hostVector double epsilon

Table 9. Structure of the HostManager class.

Once the HostManager instance of the application is created, the web graph is setup as described in section 7.4. Next the HostManager creates the number of active and inactive hosts specified, and divides the pages among those hosts (see section 7.4). Instances of the GraphCreator and NodeOperator classes are created and called from within the HostManager instance accordingly.

When a new host is created, it needs to get hold of the web pages it is responsible for, as it is assumed that there is no global access to the web graph in a real distributed environment. This information is passed in the form of a hash table, in which the URL code acts as the key, and the corresponding URLInfo object as the value. So, for a given host, all URLInfo objects, whose host number is equal to that host's number are added to the hash table. In addition, a flag is passed to the host that indicates whether it is currently active or inactive.

7.5.2. Performing Pagerank calculations

The simplest way to explain how calculations are performed in the simulation environment is to describe the lifecycle of a single iteration or pass.

During a single pass, the HostManager loops through every host in its host vector, and if a host is marked as active, then it is passed control for performing its part of the calculations. Once a host has gained control, it starts going through each of the entries in its URLInfo hash table (i.e. the hash table containing the information of the pages it is responsible for). For each URLInfo object within the hash table, the new Pagerank value is calculated.

The method performing the calculation is actually called on the URLInfo object, as it is within the URLInfo object that all the necessary information in stored (most importantly the

PRSum variable, which holds the sum part of the Pagerank equation). If after calculating the new Pagerank value, it is determined that an update is required (i.e. there is not convergence at that point), then the updated PRToPass (equation 9) needs to be passed to the outgoing links for that URL.

First the local URLInfo hash table is checked to determine if the outgoing link is stored locally. If so the PRToPass value is added to the PRSum for that link, and the entry in the hash table is updated.

If the outgoing link is not stored locally, then the HostManager is called, which is responsible for finding the host responsible for the corresponding page, and passing to that page the PRToPass value.

It should be noted that when a new PRToPass value is received by a URLInfo object, the only action performed is to add the value to the PRSum. The actual Pagerank calculation is not performed at that stage, as the URLInfo object can only propagate its new PRToPass (if it Pagerank value has not converged) when the host it is located at has been given control by the HostManager.

7.5.3. Managing retransmissions

As it is possible to inactivate hosts during the calculations, there must exist a way for retransmitting the values not delivered at pages located at inactive hosts. This issue is dealt with within each host. Specifically, when a host performs a 'remote' update operation by passing the PRToPass values to the HostManager for delivery to the appropriate host, it monitors whether the values were actually delivered or not by means of a boolean which indicates whether the operation was successful or not.

In this simulation environment, the operation can be unsuccessful if the host responsible for the specific page is inactive, or if the page in question is not within the graph parsed originally, and is thus not assigned to any host. In either case, the value that was not passed on successfully is saved in a special hash table, in which the code of the target page acts as the key, and a variable of type double comprising the sum of all the values missed so far acts as the value. At the end of the normal transmissions for a given host, all the values accumulated at that hash table are retransmitted, and for the transmissions that were successful the corresponding entries are removed from the hash table.

8. Implementation: Part B - Distributed Pagerank Calculator

8.1. Distributed Pagerank Calculator: Package Structure

In this section, the main building blocks of the distributed Pagerank application will be discussed together with Pastry-related implementation features, which are relevant for understanding how the peer-to-peer framework enables the message passing necessary for the distributed calculations.

The basic package structure of the distributed application can be seen in table 10. It should be noted that only the most important classes are included in this table, for keeping the implementation description as clear as possible. The DistPageRank class is the main class of the application that is called whenever a new instance of the calculator is to be initialized (i.e. a new node).

Package		Principal Classes
rice.p2p.pararank		
		CHashPagerankContent.class Confirmation.class ExpiredUpdates.class Pagerank.class PagerankContent.class PagerankImpl.class
	messaging	CircularsMessage.class ContinuationMessage.class InsertMessage.class LookupHandleMessage.class PagerankMessage.class RetransmitMessage.class UpdateMessage.class
	testing	DistPageRank.class

Table 10. Package structure of the distributed Pagerank Calculator built on top of Pastry.

Pagerank.class is the basic interface exported by the distributed Pagerank calculator application. This interface is implemented by the PagerankImpl.class, which will be discussed in detail in the following subsections. Whenever an instance of the Pagerank application is

initiated, it is assigned a unique identifier, which is determined by the IP address and the port number in which the instance is to be bound.

In addition, each instance of the Pagerank.class has an associated persistence storage space in some predefined directory, in which the objects it is responsible for are stored. Within a given machine, a different directory is created for each port where an instance of Pagerank.class is currently running and data stored in this directory are used for resuming the state of this instance after the node has been offline (assuming that it will bind again to the same port).

8.2. Application development on top of Pastry

Pastry provides an API for building applications that take advantage of its routing and lookup services. The most important interfaces of this API are outlined in the table below. Their meaning will be discussed in parallel with the description of the current implementation, as it is necessary to show how they link with the application developed.

Package			Principal Classes
rice			Continuation.class
	p2p	commonapi	Application.class Endpoint.class Id.class Message.class Node.class NodeHandle.class
	pastry	commonapi	PastryEndpoint.class PastryEndpointMessage.class

Table 11. Package structure of Pastry – only major classes included.

As mentioned, PagerankImpl.class is the class implementing the basic interface of the application. PagerankImpl also extends the rice.p2p.commonapi.Application interface, which defines methods for message exchange between the underlying node and the application layer.

The Pagerank interface contains the declarations of the methods that are required for controlling the calculations (e.g. handling updates or insertion of new pages into the system).

When a PagerankImpl instance is created, it registers with the local Pastry node (PastryNode). The PastryNode class is the implementation of the Node interface, and represents a peer in the system. By registering with a PastryNode, the PagerankImpl instance gets associated with an Endpoint, which acts as a virtual node which is instance specific and "which the application can then use in order to send and receive messages" ([18] Javadoc for PastryNode). It is on the Endpoint that methods relating to the local node are called. PastryEndpoint provides the implementation of the Endpoint interface.

8.3. Communication between nodes

There are two basic interfaces that need to be discussed in order to understand the way in which communication is achieved in Pastry – Message and Continuation. The former is the interface that application-specific messages need to extend, while the latter is the interface that serves as the basis of asynchronous communication between nodes. Both of these interfaces will be explained in relation to the Pagerank application.

8.3.1. Message Interface & Message Types

The class hierarchy of messages in Pastry can be seen in Figure 7. The most general message type associated with the Pagerank application is the PagerankMessage, which implements the Message interface. The Message interface is "an abstraction of a message which is sent through the … system" ([18] JavaDoc). The unique message identifier, as well as the source and destination of the message, are stored within PagerankMessage objects.

Any application built on top of Pastry needs to be able to distinguish between different types of requests. For example, in the case of the Pagerank application, the system needs to be able to handle among others, insert requests, which deal with page additions and update requests, which deal with the process of updating the Pagerank value of different pages. The way the system distinguishes between different requests and acts accordingly is by defining different types of messages (e.g. InsertMessage, UpdateMessage etc.), which are basically subclasses of the more general PagerankMessage class.

The ContinuationMessage class in figure 7, which any specific type of message needs to extend, forms the basis of asynchronous communication and is the topic of the following subsection.

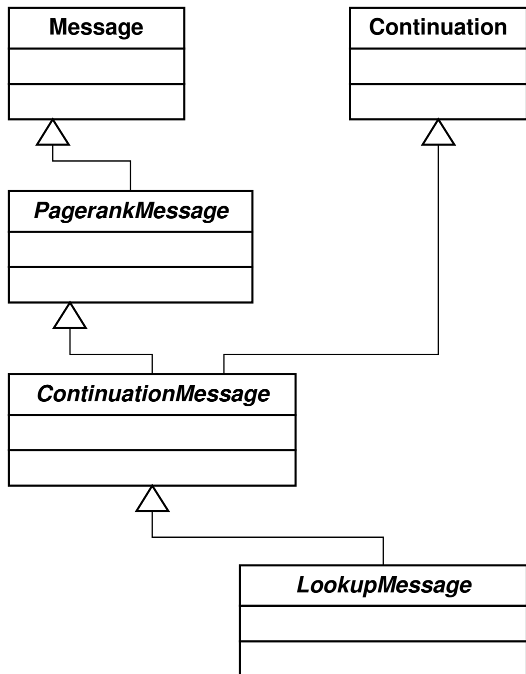

Figure 7. UML Diagram showing hierarchy of messages in Pastry

8.3.2. Continuation (Asynchronous communication)

A Continuation is an object that "asynchronously receives the result to a given method call" ([18]; JavaDoc) so that there are no blocking methods in the system. This can be achieved by passing an object implementing the Continuation interface as an argument to the method in question.

The basic structure of the Continuation interface is outlined in the table below.

Continuation

{
receiveResult(Object o) { … }
 	
receiveException(Exception e) { … }
}

Table 12. Basic methods of a Continuation interface.

Once the result of a given method becomes available, the *receiveResult* method is invoked on the Continuation, and the result is then returned to the source of the original request. If an exception has occurred during method execution, then the method invoked on the Continuation is *receiveException*.

Depending on the circumstances in which Continuation is used, the Object argument in the receiveResult method can be of any object type and the action to be taken upon receiving that Object can differ. For example, it could be a Boolean indicating whether a given operation was successful or not or a NodeHandle – i.e. an object holding the address/port information of a given node, which would be useful in a lookup context.

In order to be able to send requests to remote nodes and receive the result of those requests asynchronously, a slightly more complicated procedure needs to be followed.

In those cases, the Continuation object that specifies what the local node should do once the result of a remote operation is received, is stored locally in a hash table. The unique identifier of the message to be sent out serves as the key, and the Continuation object as the value.

In order for the original message to be able to return the result back to the requesting node, a special message hierarchy has been created. Specifically, every type of message within the system (e.g. LookupMessage) extends an object called ContinuationMessage. The latter is basically an implementation of the Continuation interface, while at the same time it subclasses the PagerankMessage class (see figure 7). This means, that every type of message is able to receive asynchronously the result of a method executed at the destination node, and then use the information provided in the PagerankMessage it subclasses in order to be routed back to its source node.

Once the message now containing the response to the original request is received back at the source node, the local Continuation that was stored initially in the hash table can be used to receive the result contained within the message and perform any locally defined operations. This way, asynchronous communication between remote nodes is achieved.

Table 13 provides an example lookup request that is dealt asynchronously with the Continuation method. This example can be seen as the general blueprint for any type of request that is handled asynchronously.

1	A new LookupMessage for id is created together with some associated Continuation for receiving the result of the lookup locally once this becomes available.
2	The unique message identifier of the LookupMessage and the Continuation are inserted into a hash table so that the system can track them down until a response is received or the request expires.
3	The LookupMessage is routed to the appropriate node.
4	When this is reached, the message is delivered to the application it is aimed at, and the type of message is determined. Casting from PagerankMessage (pmsg) to the specific message type subclass is performed (in this case LookupMessage – lmsg), and the message is handled according some predefined way.
5	In the case of a LookupMessage, the procedure is to check whether the given Id exists in local storage. The result of this lookup is received by the ContinuationMessage. The ContinuationMessage is obtained by casting the PagerankMessage (pmsg) into a ContinuationMessage (cmsg) – the latter being a subclass of the former class.
6	The ContinuationMessage is sent back to the source node.
7	When the response message is received by the original node, the original Continuation from the pending hash table is retrieved. The response contained within the ContinuationMessage is received by the Continuation, which performs some locally defined action.

Table 13. Asynchronous request handling with Continuation. Full details not provided for simplification reasons.

8.4. Representation of web pages

Objects stored in an instance of Pagerank.class implement the PagerankContent.class interface (implementation provided by the CHashPagerankContent.class), which defines the basic features of objects associated with the current application.

Each web page is assigned with a unique identifier, which is obtained by applying the SHA-1 hash function to its URL code. The resulting identifier is assigned to an Id object, which implements the rice.p2p.commonapi.Id interface (i.e. the standard interface for Pastry identifiers). The identifiers used for Pastry nodes are also represented as Id objects. In this way, there is a common denominator between Pastry nodes and PagerankContent objects, and so routing of PagerankContent objects to Pastry nodes can be achieved.

Similarly to the URLInfo object in the simulation application, the CHashPagerankContent class holds general information about a page's code, its Id and a vector containing the codes of its outgoing links. It also holds calculation-specific information, which is outlined in table 14.

ChashPagerankContent
double PRValue double PRSum boolean converged double lastPRSent

Table 14. CHashPagerankContent – variables related to Pagerank Calculations.

The *PRValue* variable corresponds to the Pagerank value of that page at any given time, *PRSum* holds the sum part of Equation 7 – i.e. the sum of the weighted Pagerank values of that page's incoming links. Finally, *lastPRSent* holds the Pagerank value at which the last update message was sent. As mentioned in the design section, this is required in order to keep track of the last update propagate and thus only propagate the difference from that update. Finally, converged is a flag indicating whether the Pagerank value of a given page has converged at a given time or not.

Like in the simulation, methods relating to the actual Pagerank calculations are called on a CHashPagerankContent instance (i.e. on a given web page). In contrast to the simulation application, here every time an update value is received, PRValue is recalculated and if necessary any update messages are propagated. This is because in the peer-to-peer system, each node is a separate entity, and thus updates can occur at any time, rather than whenever a node has control – as is the case in the simulation application.

8.5. Inserting new web pages into the system

The process of inserting new pages into the system consists of two stages. The first stage involves parsing an entry from the file storing the web graph information and creating a new PagerankContent object, which is similar to the way the simulation application handles inserts. Parsing basically involves determining the code of the next URL in the file, and creating a vector consisting of the outgoing links of that URL.

The second stage involves routing the object to the appropriate node in which it will be stored, which is undertaken by the underlying peer-to-peer framework. It should be noted that the PagerankContent class extends the Serializable interface. Serialization allows writing objects to some form of persistent storage for later invocation. This means that it is not necessary to re-insert the data sets every time the application is run, which can be a time consuming process, especially as the datasets used increase in size.

In order to handle inserts, a special type of message has been defined, called InsertMessage, which is routed to the node whose identifier is numerically closest to the identifier of the PagerankContent object to be inserted. When the remote node receives an InsertMessage, it first checks whether an object with the same identifier already exists in local storage. If so, the insert operation does not take place. Otherwise, a copy of the PagerankContent is stored locally in persistent storage in serialized form, with its identifier code as the file name. As soon as the object is stored, it is loaded to main memory, and added to a hash table (localPages), in which the Id of the PagerankContent object is the key, and the object is the value.

Finally, the outcome of the insert operation is sent asynchronously back to the original node.

8.6. Dealing with update message traffic

Whenever a page recalculates its Pagerank value, it needs to check whether it needs to propagate the update to its outgoing links. Generally speaking, if after calculating a page's Pagerank, the value has converged based on the error threshold defined, then no further updates are propagated from that page. Otherwise the new Pagerank value of that page needs to be propagated accordingly.

Potentially a substantial number of updates can occur concurrently, which means that the traffic between the nodes can be quite significant. For this reason, the way propagation of Pagerank values was designed and implemented aimed at reducing this type of traffic as much as possible.

8.6.1 Using threshold updates

As mentioned in the design section, one of the ways to reduce network traffic was to use threshold updates. The idea is that a page performing an update, won't propagate the difference immediately (assuming it has not converged), but rather will wait for more update

messages and propagate the difference once this reaches a predefined threshold. In the current implementation, the update threshold was set equal to 0.6, though further testing would be required to determine the optimal threshold value.

The identifier of the CHashPagerankContent objects that have not yet propagated their updated Pagerank value is kept in a queue (implemented as a linked list), so that it is possible to keep track of pending updates. A timer task checks this pending list at regular time intervals and sends expiry updates for the least recent entries in the queues. This is done in order to prevent a deadlock situation, where no web page sends any more updates, as it is waiting to reach the threshold value.

The way the timer task was implemented was by using the Timer class (Java 1.4.2), which allows tasks to be scheduled for execution at regular time intervals. Tasks are executed by a background thread and so classes defining timer tasks should implement the Runnable interface that allows execution as a thread. It was decided to run the timer every 3 minutes, in order to limit the total running time of the application, though it should be noted that as the size of the web graph used increases, the time intervals between the executions of the specified task should also become larger, in order to prevent cancelling out the effects of threshold updates.

An abstract class called TimerTask is provided in Java 1.4.2, which implements the Runnable interface, and can be extended by any class wishing to define timer tasks. Since TimerTask is implementing the Runnable interface, any instance of this class will be executed as a thread, thus not blocking the normal execution of the application.

For the purposes of defining the timer tasks, the ExpiryUpdates class was developed, which extends TimerTask. Every time it is executed by the Timer, it checks whether there are any items pending to send their updates (waiting for the threshold to be reached), and if so sends update messages for up to a maximum specified number of items, starting from the least recently added.

8.6.2. Distinguishing between update messages

As mentioned, Pastry nodes communicate via RMI. Messages received are added in a queue – typically in a first-come-first-served basis, and are then handled by a message handler thread, which delivers them to the PastryNode for processing.

A problem encountered at an early stage of the implementation was that the queue of received messages quickly reached its upper message limit and started dropping messages. The reason for this was that messages were processed within the application at a slower rate than the rate they were arriving at, and thus messages were removed from the queue slower than they were added. In addition, this issue was more pronounced at the beginning of the calculations when all nodes were sending messages to each other.

An approach that was attempted to solve this problem – though it also did not provide a working solution, was introducing a new type of message CongestionMessage that would be send back to the sender, every time a full buffer was encountered at the receiver end, so that retransmissions could be scheduled.

This issue was dealt with by splitting the update messages circulating in the system into three different types and implementing a separate message handler thread for each of these message types within the actual application, rather than at the Pastry layer. The three types of update messages created were CircularMessage, UpdateMessage and RetransmitMessage.

A CircularMessage is a message that is generated at the initialization stage, when each node goes through every page that is stored locally and propagates its Pagerank value. An UpdateMessage is a message generated at any other stage – e.g. during a threshold update. Finally RetransmitMessage concerns retransmission of updates in case of unsuccessful delivery and is further discussed in section 8.7.

Both the CircularMessage and the UpdateMessage have the same internal structure, which consists of the Id of the target PagerankContent and the value that needs to be added to the PRSum part of that page before re-calculating its PRValue.

Thus, increased message traffic causing messages to be dropped at the receiver end was dealt with using the two approaches outlined – threshold updates and division of messages into different categories. Even though this solution worked satisfactorily for the web graph sizes used, it is acknowledged that problems could be encountered when testing the system with a larger web graph, as it was found that the use of multiple threads for receiving messages only worked in conjunction with the threshold updates solution. This suggests that an alternative solution needs to be considered in the future. The CongestionMessage approach discussed above is a more viable solution in conjunction with threshold updates, as it is more sensitive to the state of the system at any given time, rather than following a generalized approach.

8.7. Confirming delivery of messages

As mentioned above, if a message is not successfully delivered at the destination node, the source node is responsible for re-transmitting the message until it is eventually delivered. It should be noted that this functionality has only been implemented for update messages as missed updates can cause diversions in the final Pagerank values, which are not possible to change at a later stage.

Delivery notification occurs asynchronously by means of a Continuation object, whose methods have been overridden in order to act accordingly depending on whether the message was delivered or not.

If the node responsible for a given PagerankContent is offline, then an update message aimed at that object is delivered to the node that is now responsible for the id range originally covered by the node currently offline. However, the specific PagerankContent is not found within that node, and so delivery is not successful in the sense that the target PagerankContent was not reached.

After sending an update message, the source node receives asynchronously an object of type Confirmation, whose structure can be seen below, through the local Continuation object. The *pageId* field corresponds to the PagerankContent the Confirmation was originally aimed at, the *delivered* flag signifies whether delivery was successful or not, and the *values* variable contains the update value sent with the original update message.

The idea is that whenever an update message is not delivered, the source node needs to store the id of the PagerankContent the update was aimed at, together with the value that was to be transmitted. This way no updates are lost and they can be retransmitted at some later time.

Confirmation
Id pageId boolean delivered double values

Table 15. Structure of the Confirmation class.

The way this information is stored locally is by adding it in a hash table (retransmitHash), in which the target PagerankContent is the key, and the sum of missed updates the value.

Every time then an update message (of any type) is not delivered successfully, the source node checks whether the given sourceId already exists in the retransmitHash, and if so adds

the new missed value to the relevant entry's vector. Otherwise, a new entry is added to the hash table.

When the RetransmitMessage is received successfully by the target PagerankContent, it is handled the same way update messages are handled. Retransmissions take place whenever the TimerTask described in section 8.6.1 is scheduled to run, thus avoiding the need for an additional scheduler.

8.8. Saving the state of the system

As in a peer-to-peer system, peers can go offline at any time, it is important to preserve the state of peers, so that the next time they come online they can continue with their calculations and the propagation of update messages from the point that they had stopped. Otherwise, if every node starts again from scratch after going offline for a while, Pagerank values will not be accurate, as they will appear to have different values depending on the dynamics of the system.

For these reasons, it was decided to back up the state of the system, every time the Timer expires. Specifically, three objects are stored every time, so that the state of the system can be restored. These are the hash table containing the PagerankContent objects assigned to the local node, the queue of scheduled updates, as well as the hash table containing the identifiers of the pages for which retransmissions exist, and the corresponding value that needs to be retransmitted.

9. System Testing & Evaluation

In order to evaluate the system developed, a number of testing procedures were setup. Before discussing in detail the tests conducted and datasets used, a brief description of the measures used for evaluating the system will be provided.

9.1. Experimental measures

A number of measurements were taken during the testing in order to determine performance of the system under different conditions. These measures varied between the simulation and the P2P implementation.

9.1.1. Simulation measures

In the simulated system, it does not make sense to measure number of messages generated, as the way calculations are performed is not message-based, but rather relies on switching control from one host to another. As a result the measure chosen was number of passes until convergence. As already mentioned in the implementation section, a pass is defined as a single loop through each of the active hosts in the system. A pass takes place as long as there is at least one active host in the system that contains pages whose values have not yet converged.

In order to get a better picture about the rate of convergence, an additional measure was used. This measure looks at the percentage of pages whose Pagerank value has converged at the end of each iteration.

9.1.2. P2P performance measures

For the P2P system, the main measure of performance is the number of messages exchanged between peers until convergence. Local updates were not counted in this case, as the focus is on the network traffic generated between peers during the calculations. It should be reminded that the total number of messages sent can be divided into two types: update messages and confirmation messages. For every update message sent, a confirmation message is generated to let the sending peer know whether delivery was successful or not. So the total number of messages is double the number of update messages required for convergence.

9.1.3. Comparison measures

As the core measures of the simulated and the P2P system are fundamentally different, the way to evaluate the two implementations in parallel was by comparing the actual Pagerank values obtained using each method.

The comparison was based on a measure of relative error, defined as

$$abs(P_{dist} - P_{sim}) / P_{sim} \qquad \textbf{Equation 10}$$

Sankaralingam et al. (2003) used an equivalent measure in their evaluation of the quality of distributed Pagerank values, using a centralized implementation as a point of comparison. In the present setting, the results from the simulation were used as a point of comparison, as they do provide a well-controlled version of the algorithm.

9.2. Data sets & Setup procedures

Three web graphs consisting of (approximately) 100, 400 and 1000 nodes each were used throughout the testing procedures. All web graphs were taken from academic domains – psychol.cam.ac.uk, psych.ox.ac.uk and bio.ic.ac.uk respectively. Any outgoing links pointing to pages outside the given web graph were removed in order to keep the system as controlled as possible.

The P2P system was tested with 1, 6 and 12 peers. The standard procedure followed in the peer-to-peer environment was to initialize all peers by inserting the respective web graph into the system and then initiate the calculations within each peer. It was decided that for the sizes of web graphs used, these numbers of peers would produce a representative picture of the performance of the system. Sankaralingam et al. (2003) tested their system with 500 peers using a minimum web graph size of 10000 nodes, which is an equivalent of 20 nodes per peer. In our system, with a minimum graph size of 100, and a minimum of 6 peers, this is an equivalent of approximately 15 nodes per peer (assuming even distribution of pages among peers) for the smallest web graph used.

9.3. Pagerank Value and Indegree of Pages

According to the assumptions on which the original algorithm is based, the Pagerank value of a given page should be proportional to the number of incoming links pointing to that page. At the same time, the quality and outdegree of those links should affect their contribution to that page's Pagerank value.

In that respect, one should expect a positive correlation between the indegree of a given page and its Pagerank value, but this correlation should not be perfect, as the incoming link quality and the outdegree of incoming links also affect the weight of this contribution.

This is the general trend was obtained both at the simulation and the peer-to-peer test data, as can be seen in figure 8. There is a clear positive correlation both for the simulation and the distributed data, which is actually considerably strong (with correlation values of 0.97 and 0.96 respectively). At the same time, it is possible to see from the trend, that indegree is not the only factor affecting pagerank. For example, there is a flattening of the lines between the indegree values of 100 and 150, which confirms that other factors are at play as well when determining a page's value.

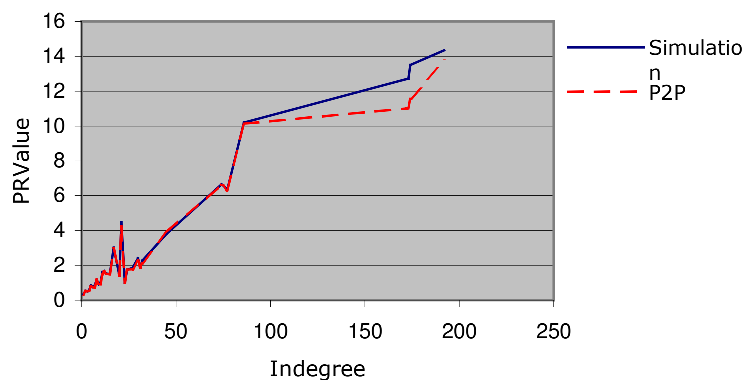

Figure 8. Correlation between the indegree of a given page and its Pagerank value

9.4. Convergence in the Simulated Environment

The number of passes required for Pagerank values to converge with different sizes of web graphs and with different epsilon values (i.e. error threshold values) under standard

calculation conditions (i.e. all hosts active at any one time and all pages inserted before calculations commence) can be seen in table 16.

pages	epsilon	1 node	6 nodes	12 nodes
100	0.01	10	10	10
100	0.001	14	15	14
100	0.0001	17	18	18
400	0.01	14	14	14
400	0.001	22	22	22
400	0.0001	30	30	30
1000	0.01	14	13	13
1000	0.001	22	21	21
1000	0.0001	30	29	29

Table 16. Simulation: Number of passes for convergence under standard calculation conditions.

There are a few points that need to be discussed concerning this table. First of all, the number of passes is not affected by the number of nodes present in the system. This is expected in the simulation application when all hosts are present, as the same calculations need to be performed independently of how pages are distributed among the hosts.
The increase in number of passes from the 100-page graph to the 400- and 1000-page graph is approximately in the order of 1/3. At the same time, the same number of passes was required for both the 400- and 1000-node graph. This is an indication that the system is potentially scalable to larger web graphs, though more extensive testing would be required to determine the relation between web graph size and number of passes.

Even though these tables provide a general idea of the performance of the simulation program, they don't provide information about the proportion of pages that have converged at different stages of the iterations. For this reason, an additional measure was introduced. This measure was taken at the end of every iteration, and provided the percentage of pages that had converged at that stage. As each web graph required a different total number of iterations until convergence, iterations were converted to percentage values. The figure below shows the percentage of iterations passed before 50%, 80% and 95% of the total pages in a graph had converged.

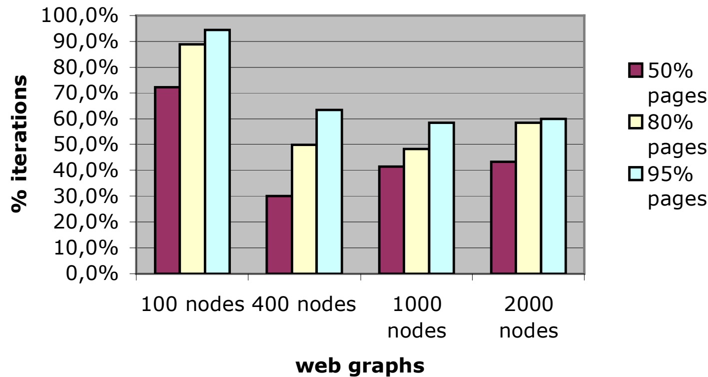

Figure 9. Graph relating percentage of iterations to percentage of pages converged.

This graph shows that, with the exception of the 100-node graph, for the other two graphs, 95% of pages had converged after 60% of the total iterations had been performed, which is similar to the convergence rate obtained in previous work and serves as evidence for the efficiency and correctness of the system. Sankaralingam et al. (2003) found that the distributed Pagerank values they obtained converged within 1% of the expected values after approximately 50% of the passes had taken place.

It is worth noting though that the 100-node graph seems to have poor convergence performance both when looking at the total number of passes it required (compared to larger web graphs), and when looking at the rate at which the majority of its nodes converged. What is suggested is that there is possibly a minimum number of passes that the system needs to perform, and that the system starts exhibiting its scaling behaviour only with larger web graphs.

Overall, the trends obtained from these initial findings serve as an indication that the design of the simulation application is potentially scalable to even larger web graphs, without a dramatic increase in the number of passes before convergence. The results are generally consistent, with the exception of the 100-node graph, which demonstrated poor convergence behaviour, in contrast to what would be expected. This could be related to the specific graph chosen, though it was rather suggested that there is a minimum of passes that need to take place before values converge.

9.5. Convergence in the P2P Environment

The number of messages exchanged between peers in the 6 peer testing procedure (again under the standard calculation procedures) until convergence can be seen in table 17 for each of the web graphs used, and for three different threshold values. The *per peer* measure, which is obtained by dividing the total number of messages by the number peers, assumes even distribution of pages and messages among nodes, and is only an indication of the average traffic generated by each peer.

pages	threshold	update msgs	total msgs	per peer
100	0,01	1386	2772	462
100	0,001	2092	4184	697
100	0,0001	2968	5936	989
400	0,01	6227	12452	2075
400	0,001	10483	20966	3494
400	0,0001	15237	30474	5079
1000	0,01	16228	32456	5409
1000	0,001	31809	63618	10603
1000	0,0001	51431	102862	17144

Table 17. Number of messages generated during calculations with 6 active peers under the standard calculation procedures.

The traffic generated when having 12 instead of 6 active peers can be seen in table 18. The same notation applies to this table.

pages	threshold	Update msgs	total msgs	per peer
100	0,01	1405	2810	234
100	0,001	2075	4150	346
100	0,0001	3332	6664	555
400	0,01	6596	13192	1099
400	0,001	9582	19164	1597
400	0,0001	14150	28300	2358
1000	0,01	19606	39212	3268
1000	0,001	34046	68092	5674
1000	0,0001	54362	108724	9060

Table 18. Number of messages generated during calculations with 12 active peers under the standard calculation procedures.

As expected, the number of messages generated by the 6-peer and 12-peer systems do not vary to a significant extent, as the same calculations need to be performed independently of the number of peers. In the majority of cases, the total number of messages generated by the 12-peer system is slightly higher, which would be expected, as in the latter case pages are

more widely distributed among the available peers, and it is more likely that outgoing links from a given page will be located on a different peer.

The number of messages per peer is reduced by a factor of 2 in the 12-peer system compared to the 6-peer system. Again, this is expected, as pages are now more widely distributed, so each peer is responsible for a smaller number of pages. This draws attention to the fact that as larger web graphs are introduced, more peers should take part in the calculations in order to balance traffic as much as possible, and avoid congesting peers.

In both the 6- and the 12-peer system, the increase of message traffic as the web graph size increases seems to follow a roughly linear trend. In addition there is an increase in total messages in the factor of 3, as the error threshold decreases from 0.01 to 0.0001. This is consistent though less prevalent with the trend obtained by Sankaralingam et al. (2003) who found a threefold increase in messages as the error threshold decreased form 10^{-1} to 10^{-6}.

Even though, as already discussed, there are certain differences between the present implementation and the one by Sankaralingam et al. (2003), it is still interesting to compare the message traffic generated by the two implementations. The comparison is based on the 10000-node web graph condition in the Sankaralingam et al. paper. The number of messages generated in that condition can be seen in table 19.

pages	threshold	total msgs
10000	0,01	510000
10000	0,001	630000
10000	0,0001	750000

Table 19. Sankaralingam et al. (2003). Message traffic for 10000-node web graph with 500 peers.

In order to allow for better comparison, the ratio of message traffic in the 10000-node condition, and the other conditions, for all epsilon values was calculated and the results can be seen in table 20.

	Increase in number of pages		
Epsilon	10-fold	25-fold	100-fold
0,01	16	41	184
0,001	10	30	151
0,0001	7	25	126

Table 20. Increase in message number as a function of web graph size increase.

What this table indicates, is that for an epsilon value of 10^{-4}, the present system and the Sankaralingam et al. implementation had roughly equivalent performance in terms of message

traffic (especially for the 1000- and 400-node graphs – columns 2 and 3). For a tenfold increase in the number of pages (i.e. 1000 versus 10000), there was a sevenfold increase in the number of messages generated, and for a 25-fold increase in the number of pages (400 versus 10000), there was a 25-fold increase in message traffic. In general, with the exception of the 100-node graph – which again did not follow the trends of the other graphs, the present system seems to generate similar message traffic patters as the Sankaralingam et al. (2003) paper. However, it should be noted that the trends for higher error thresholds suggest that there is further potential for reducing message traffic.

There are a few suggestions for decreasing the message traffic. One method would be grouping messages in the sender according to the destination address, and only sending them out as a single message once a given number of updates has accumulated. An additional strategy would be to reduce the frequency of updates by further increasing the update threshold (i.e. the difference reached between old new values before an update is sent), so that multiple updates concerning a specific page are coalesced within a single update message.

9.6. Comparing the Simulation & P2P system accuracy

As the basic measures of system performance differ between the simulation and the peer-to-peer implementations, the way to compare the two is by using a Pagerank accuracy measure. The Pagerank values obtained in the simulation version are used throughout this section as the baseline measure, against which results in the distributed version are compared for accuracy.

As already mentioned, the measure used to determine accuracy of the peer-to-peer system is relative error as defined in section 9.1.3. It should be mentioned that in order to allow comparisons between the present implementation, and the one developed by Sankaralingam et al. (2003), the results have been structured in the same manner as in the corresponding paper.

9.6.1. Accuracy in a single-peer system

The calculations were first run using a single peer in both the distributed and the simulated version in order to see how the peer-to-peer implementation performs when there is no between-peer message passing involved. The relative errors of the values obtained can be seen in table 21. The '*% pages*' measure in the first column corresponds to the percentage of pages that have a relative error equal or less than the corresponding value.

% Pages	Threshold (epsilon) 0,01	0,001	0,0001
		100 pages	
90%	0,004	0,001	$5,2*10^{-5}$
Max.	0,002	0,001	$8.8*10^{-5}$
Avg.	0,003	0,0002	$2,7*10^{-5}$
		400 pages	
90%	0,006	0,001	0,001
Max.	0,015	0,002	0,022
Avg.	0,001	0,0002	0,002
		1000 pages	
90%	0,004	0,001	$4,4*10^{-5}$
Max.	0,002	0,002	0,0002
Avg.	0,01	0,0002	$1.8*10^{-5}$

Table 21. Relative error values for the peer-to-peer system with one active node compared to a 1-node simulation condition.

It can be said that the single peer distributed system produces overall highly accurate results compared to the one-node simulated system, the average relative error being mostly in the magnitude of 10^{-3} or less.

This finding is important for evaluating the multiple-peer distributed system, as it suggests that the main source of any divergence in Pagerank values will be related to the way message passing and updates are handled rather than to some inherent problem in the algorithm.

9.6.2. Accuracy in a multiple-peer system

Having determined that a single-peer distributed system converges with high accuracy to the values produced by the single-node simulation, we now proceed to evaluate the accuracy of the Pagerank values produced by the peer-to-peer system with multiple active peers.

Tables 22 and 23 summarize the relative error values for the 6- and 12- peer system, for all three we graph sizes used, for each of three error thresholds. The relative error values have been calculated using both the single-node and 6-node simulation results. As the table shows, there is no difference between the relative error values based on the 1-node simulation and those based on the 6-node simulation. This again shows that any divergence in the values in the peer-to-peer implementation is related to message passing between active peers in the system, as in the simulated system, where there is no message passing involved, there is no divergence the values as the number of active nodes varies.

Simulated Nodes	Threshold (epsilon) 0,01		0,001		0,0001	
	1	6	1	6	1	6
% Pages	100 pages					
50%	0,047	0,043	0,047	0,045	0,047	0,045
90%	0,373	0,372	0,377	0,377	0,378	0,378
Max.	3,362	3,364	3,402	3,402	1,991	1,991
Avg.	0,281	0,280	0,283	0,283	0,206	0,206
	400 pages					
50%	0,004	0,003	0,002	0,002	0,002	0,002
90%	0,053	0,051	0,056	0,060	0,060	0,060
Max.	0,569	0,569	0,589	0,589	0,59	0,590
Avg.	0,023	0,022	0,021	0,021	0,021	0,021
	1000 pages					
50%	0,009	0,009	0,003	0,003	0,003	0,003
90%	0,159	0,158	0,084	0,084	0,084	0,084
Max.	0,842	0,842	2,898	2,898	2,906	2,906
Avg.	0,054	0,055	0,040	0,040	0,040	0,040

Table 22. Relative error values for the peer-to-peer system with 6 active peers.

Simulated Nodes	Threshold (epsilon) 0,01		0,001		0,0001	
	1	12	1	12	1	12
% Pages	100 pages					
50%	0,060	0,057	0,059	0,059	0,059	0,059
90%	0,576	0,573	0,576	0,576	0,576	0,580
Max.	1,799	1,800	1,804	1,805	1,807	1,807
Avg.	0,210	0,210	0,210	0,210	0,210	0,210
	400 pages					
50%	0,003	0,004	0,001	0,001	0,002	0,001
90%	0,044	0,041	0,039	0,039	0,038	0,038
Max.	0,744	0,746	0,749	0,749	0,750	0,750
Avg.	0,022	0,022	0,021	0,021	0,021	0,021
	1000 pages					
50%	0,005	0,005	0,004	0,004	0,004	0,004
90%	0,111	0,108	0,109	0,108	0,109	0,109
Max.	2,964	2,947	3,005	3,005	3,009	3,009
Avg.	0,049	0,049	0,048	0,048	0,048	0,048

Table 23. Relative error values for the peer-to-peer system with 12 active peers.

The 6- and 12-peer systems produce Pagerank values of comparable accuracy and thus it can be inferred that varying the number of peers in the system will not have any effect on the values obtained. Rather it will only affect the number of messages per peer, as section 9.5 suggested. Thus adding more peers to the system should only be seen as a means of reducing traffic per node.

The relative error measure seems to perform the worse for the 100-page web graph. It should be reminded that the performance of this web graph in the initial simulation tests was also not very satisfactory in terms of number of iterations until convergence.

The average relative error values between the simulation and peer-to-peer Pagerank values are between 10^{-1} and 10^{-2}, which indicates that there is some issue in the implementation causing the Pagerank values in the distributed version to diverge to a certain extent from the values obtained in the simulation. This is further confirmed by the fact that the maximum relative error value obtained in as high as $3{,}402$ (table 22; 100 pages; $epsilon = 0.01$), which indicates that there must be some problem in the way update messages are exchanged between peers.

By looking at the raw data, what became evident was that for a number of pages, the Pagerank value was equal to the initialization value. Considering that according to the indegree information extracted, all pages had at least one incoming link, it should be the case that all pages end up with a Pagerank value that is even slightly higher than the initialization value.

This observation suggests that in some cases Pagerank update messages were not sent out, and so certain pages' values diverged from the expected Pagerank values. It has not been possible to determine the exact reason why certain update messages were not actually sent out, but what is suggested is that it is possibly related with the way updates were propagated depending on whether the update threshold was met or not.

In order to further investigate this issue, the direction of error in the calculations was looked at, as a means to determine whether there was any consistent trend to over- or underestimate values. For this purpose, the relative error calculations were performed again. This time, instead of dividing the absolute of the difference between the two values by the simulation value, the raw value was divided by the simulation value.

This showed that in the peer-to-peer implementation, Pagerank values were consistently lower than in the simulated implementation, which is consistent with the suggestion that a number of update message failed to be sent out to their outgoing links. Unfortunately, due to time limitations it has not been possible to deal with this issue effectively.

It should finally be mentioned that for the 10000-page web graph condition, Sankaralingam et al. (2003) obtained considerably lower relative error values. Specifically, 90% of Pagerank values had relative error in the magnitude of 10^{-3} to 10^{-5} at all error thresholds.

9.7. Dynamic Page Insertions

The effect of new page insertions into the system was tested by breaking down the three web graphs into a number of smaller graphs. These smaller graphs were inserted gradually into the system. After a new part was inserted, the values were left to re-converge before a further part was inserted. The Pagerank values were not reset between the insertion of two web graph parts, but rather pages were allowed to continue the calculations from the values they had converged at up to that point. This feature was tested only in a 6-peer system, with an error threshold of 0,001 due to time constraints.

The process of gradual convergence of values has been modelled in figures 10 and 11. The line on top represents the final Pagerank values, while the line at the bottom represents the converged Pagerank values after only a fraction of the web graph has been inserted. Figure 10 shows the divergence from the final values after only 10% of the pages of the 400-page web graph have been inserted and left to converge, while figure 11, shows the corresponding divergence after 40% of the pages have been inserted and left to converge.

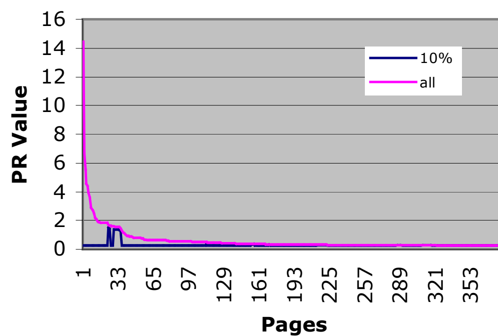

Figure 10. Pagerank value convergence after 10% of pages have been inserted, compared to final values (for 400-page web graph).

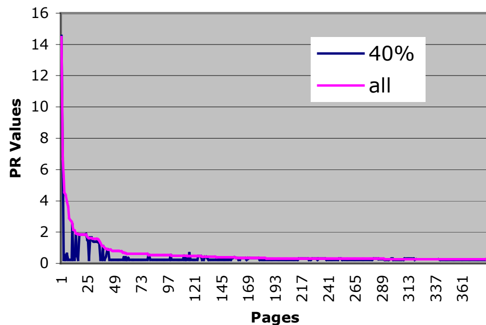

Figure 11. Pagerank value convergence after 40% of pages have been inserted, compared to final values (for 400-page web graph).

9.7.1. Simulation System

Table 24 displays the number of passes required for convergence of all Pagerank values in the system. The increase in number of passes between this condition and the standard calculation condition is almost three-fold. What should be kept in mind is that there results are dependent on how the web graph is partitioned, as it how linked pages are spread that affects how quickly the algorithm will converge.

Pages	Epsilon	# passes
100	0,001	34
400	0,001	60
1000	0,001	90

Table 24. Number of passes required until convergence for the page insertion condition.

The accuracy of the results obtained in the simulation under the page insertion condition was evaluated against the results obtained from the standard calculation procedure (in the simulated version). As table 25 indicates, the divergence in the values between these two conditions was very low – between 10^{-3} and 10^{-5}, thus indicating that in a well-controlled system, gradual page insertions do not cause any divergence in the final Pagerank values.

	100 pages	400 pages	1000 pages
% Pages			
50%	0,0003	$1{,}7*10^{-5}$	0,0001
90%	0,0004	0,0002	0,001
Max.	0,0007	0,0006	0,065
Avg.	0,0002	$6{,}2*10^{-5}$	0,001

Table 25. Relative error of Pagerank values in the dynamic page insertion condition.

9.7.2. Peer-to-peer System

As in previous sections, the message traffic generated under the dynamic page insertion condition is first considered. The number of messages that were exchanged until convergence can be seen in table 26, for all three web graphs used.

pages	threshold	update msgs	total msgs	per peer
100	0,001	3664	7328	1221
400	0,001	9544	19088	3181
1000	0,001	32307	64614	10769

Table 26. Number of messages generated by gradually inserting fractions of the web graph.

With the exception of the 100-page web graph, the number of messages required until convergence under the dynamic page insertion condition is approximately the same as the number of messages required under the standard condition described initially. This is an interesting finding, as it implies that the system responds well to dynamic updates. This is important, as one of the reasons for proposing a distributed algorithm in the first place, was to allow dynamic updates.

9.7.3. Accuracy of Pagerank Values

The Pagerank values obtained in this condition were compared against the values obtained in the 6-node simulation condition (nodes being active throughout), as section 9.6.2. showed that the 1-node and 6-node simulation results produce nearly identical relative error values, and it was considered more intuitive to compare systems running with the same number of nodes. The relative error values obtained can be seen in table 27.

	100 pages	400 pages	1000 pages
% Pages			
50%	0,057	0,004	0,032
90%	0,371	0,079	0,282
Max.	2,683	0,383	1,540
Avg.	0,252	0,026	0,087

Table 27. Relative error values for pages inserted gradually into the system.

The average relative error values between the simulation and peer-to-peer Pagerank values again indicate that there are some issues concerning the exchange of update messages between peers, which causes the Pagerank values to diverge to a certain extent. In this condition, this issue could have been attenuated by the gradual insertion of pages, though the

results suggest that the divergence is within the same range obtained in the standard calculation condition, thus suggesting that the problem is not related to the dynamic feature introduced, but rather to some more basic update propagation issue.

9.8. Node arrivals & departures

The way node arrivals and departures were modelled was by activating one peer or host at a time, performing the calculations until convergence and then activating a further peer and following the same approach. In other words, assuming a 6-host system, initially only *host 1* was active. After calculations within that host had converged, host 2 was activated, thus now having two active hosts in the system. This process was repeated until all peers in the system had been activated.

This approach was followed in both the simulation and the peer-to-peer environment. The reason why this method was devised is because that it provides a well-controlled routine for testing how the system handles node arrivals. Even though node departures are not performed directly as nodes start from an offline state, as long as retransmission of messages works correctly – something that can be tested in the present context, arbitrary node departures should present no complications.

Like page insertions, this test was only conducted in a 6-host system with an epsilon value of 0,001 due to time constraints.

9.8.1. Simulation System

Table 28 displays the number of passes required for convergence of all Pagerank values in the system. Again, as in table 20 where all hosts were active throughout, the number of passes for the 400- and 1000-node web graphs are very similar. This suggests that the system performs similarly for given ranges of web graph sizes, though this needs to be confirmed using larger and more varied datasets.

Pages	epsilon	# passes
100	0,001	42
400	0,001	97
1000	0,001	95

Table 28. Number of passes for required during simulation of node arrivals.

The increase in passes for the 100-page graph, when compared to the standard procedure (all hosts active) is approximately threefold, while for the other two web graphs there is a roughly fourfold increase in the number of passes required for convergence. This increase was expected, given that the conditions simulated are extreme, in the sense that new hosts are activated only when calculations of existing hosts have converged and a considerable number of pages are unavailable for a large part of the calculations.

9.8.2. Peer-to-peer System

The number of messages sent under the equivalent conditions in the peer-to-peer framework can be seen in table 29.

Pages	threshold	update msgs	total msgs	per node
100	0,001	2457	4914	819
400	0,001	19325	38650	6442
1000	0,001	45459	90918	15153

Table 29. Number of messages generated during simulation of node arrivals.

By comparing these values with the results obtained under the condition where all peers are active throughout (see table 22), it can be seen that the increase in message traffic is between 20% (for the 100-page graph) and 85% (for the 400-page graph), while for the 1000-page graph there is a 45% increase in message traffic. Taking nodes offline then seems to produce more pronounced effects in the rate of convergence than gradually inserting pages, as in the former case, certain pages are not reachable at all, and thus the balance of system is more radically changed when a new host comes online.

9.8.3. Accuracy of Pagerank Values

Finally, it is interesting to look at the accuracy of Pagerank values under these conditions. Initially, the accuracy of the simulation values under dynamic conditions was evaluated against the results obtained in the standard calculation condition. As table 30 shows, the simulation system produced highly accurate results when node dynamics were introduced as compared to the standard simulation condition, for all three web-graph sizes.

	100 pages	400 pages	1000 pages
% Pages			
90%	$1,4*10^{-5}$	$4,8*10^{-6}$	$1,9 * 10^{-6}$
Max.	0,0005	0,0005	0,001
Avg.	0,0001	0,0001	0,0001

Table 30. Accuracy of the simulation condition with node arrivals against standard simulation.

As a result, it was decided to compare the accuracy of the results obtained in the peer-to-peer system under node arrival condition, against the results obtained in the equivalent simulation condition. The results can be seen in table 31.

	100 pages	400 pages	1000 pages
% Pages			
50%	0,093	0,005	0,011
75%	0,162	0,050	0,055
90%	0,395	0,157	0,161
Max.	1,883	0,639	1,071
Avg.	0,173	0,042	0,048

Table 31. Relative error for node arrival condition with 6 nodes and 0,001 error threshold.

These relative error values are in the same magnitude as the relative error values obtained during the standard calculation procedure (with all peer present; see table 25), which once again suggests that any divergence is not due to failure of the system to deal with new node arrivals, but rather with the way update messages are passed between nodes. Thus the system is responsive to new node arrivals and can converge to the expected Pagerank values obtained under standard conditions.

Finally, concerning execution time, this was not considered in the present system, as the way the system was implemented due to the problems encountered, ended up introducing states at which no updates where sent between nodes waiting for expired updates to be sent – which are in a way artificial delays. Under these conditions it is not possible to produce accurate execution time results, and as a consequence it was decided not to include such a measure.

10. Conclusions

10.1. Outcomes of the current project

The aim of the present project was to develop and evaluate a distributed version of Google's Pagerank algorithm within a peer-to-peer framework. The accuracy of the peer-to-peer implementation was compared with a simulated version of the distributed algorithm running on a single machine. Features like dynamic page insertions and new node arrivals into the system were implemented and tested.

Overall, the results obtained during the testing phase of the project suggest that under standard conditions, where all peers are active throughout the calculations and all pages are inserted before calculations commence, the peer-to-peer system can converge with considerable accuracy when compared to the simulated version. This is achieved without any need for synchronization, in contrast to previous work imposing certain synchronization constraints. The message traffic generated is comparable to message traffic obtained in previous implementations of the distributed algorithm, though for higher error threshold values, the system tends to produce proportionately higher message traffic than expected, an issue worth further investigation.

Considering the ability of the distributed system to deal with dynamic features like gradual page insertions, and new node arrivals, it appears that these conditions produce similar accuracy ratings with the standard calculation procedure. In addition, the message traffic generated under these conditions generally suggests that the system can cope effectively with these dynamic changes.

10.2. Limitations & Future work

Even though results suggest that the distributed system converges with considerable accuracy, the relative error values obtained indicate that there is some issue in the implementation causing the values to diverge to some extent from the target Pagerank values. What was suggested in the testing section is that there is some problem with the way updates are propagated between peers, which tends to cause an underestimation of values in the distributed system.

An important problem with the current peer-to-peer implementation concerns the issue of dealing with increased message traffic generated by update messages. This was handled by

introducing threshold updates and implementing separate message receiver threads for receiving different types of messages. Even though the combination of these two approaches worked satisfactorily in this context, it is acknowledged that it could potentially not be sufficient if larger web graphs are used – something that should be considered in future work.

In addition, in order to avoid queue size problems, the update threshold had to be kept above a certain limit, which introduced artificial delays in the system, thus not allowing the present implementation to provide any execution time estimates. This is an area that should be further explored, as execution time is an important factor in a system that could potentially be used to perform calculations on graphs in the magnitude of billion nodes.

As mentioned in the implementation section, the objects holding information about web pages are loaded in main memory and accessed from there throughout the calculations. With larger web graphs this approach will not be feasible. What is suggested is that a cache of the most recently accessed pages should be kept in memory. The optimal size of this cache will vary with the graph size and needs to be explored in future implementations.

Future work should also consider handling page deletions from the system, as well as changes in linkage information of a given page. For example, the number of outgoing links from a given page could change, leading to changes in the Pagerank contribution of that page to its outgoing links. Sankaralingam et al. (2003) suggested handling page deletions by sending a message to the outgoing links of the page to be removed, requesting them to negate its Pagerank value. This approach can potentially deal with the issue of changes in linkage information within a page as well, by initially negating the corresponding Pagerank value, recalculating the weight that should be passed to each of the outgoing links and then propagating this new value.

Finally, the present system assigns pages to a peer at start-up, and this allocation does not change after this point. This means that if new peers are added to the system at a later stage, they don't contribute to the calculations, except if a new set of pages is inserted at a later point. Implementing a more flexible system that would allow for redistribution of pages as new peers are added to the system would be an interesting extension of the current project.

10.3. Concluding comments

This project has investigated the properties of an asynchronous implementation of a distributed Pagerank algorithm within a peer-to-peer framework. The system was found to perform satisfactorily both in terms of accuracy of the Pagerank values obtained and in terms of the message traffic generated. The system also dealt effectively with dynamic features like page insertions and node arrivals. The trends obtained during the testing phase of the project are consistent with previous work, and in contrast to previous implementations have not involved any type of synchronization between peers taking part in the calculations.

As the need for efficient distributed content ranking algorithms is becoming more and more evident - given the growing size of the World Wide Web, and the increasing popularity of peer-to-peer file sharing systems - special consideration should be given in future research to dynamic features of the algorithm, as these are its competitive advantage compared to the standard centralized Pagerank algorithm.

References

[1] S. Breen. Stochastic Matrices & Markov Chains. Available at www.spd.dcu.ie/staff/breens/202notes/202sub5,4.pdf.

[2] S. Brin, L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. WWW7/Computer Networks 30 (1-7): 107-117. 1998.

[3] C. Caldwell. Graph Theory Glossary. Available at http://www.utm.edu/departments/math/graph/glossary.html

[4] M. Castro, M. Costa, A. Rowstron. Peer-to-peer overlays: structured, unstructured or both? MSR-TR-2004-72. Microsoft Research Cambridge. 2003

[5] D. Chazan, W. Miranker. Chaotic Relaxation. In Linear Algebra and its applications 2(1969): 199-222.

[6] J. Cho, H. Garcia-Molina. Parallel crawlers. In *11th International World Wide Web Conference*. 2002.

[7] The Chord Project. Available at www.pdos.lcs.mit.edu/chord/

[8] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In Proceedings of the 2nd International Workshop on Peer-to-Peer Systems. 2003.

[9] P. Druschel, A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. HotOS VIII, Schoss Elmau, Germany. 2001

[10] B. Hescott. Random Walks. Available at www.cs.bu.edu/~steng/teaching/Fall2002/lecs/ben.ppt. November, 2003; Boston University.

[11] R. J. Gerrard. Discrete Stochastic Modelling; Markov Chains (Notes). Available at http://www.staff.city.ac.uk/r.j.gerrard/courses/ 2dsm/dsm03_4.htm

[12] T. Haveliwala, S. Kamvar, D. Klein, C. Manning, G. Golub. Computing Pagerank using Power Extrapolation. Stanford University Technical Report, 2003.

[13] JDSL: The Data Structures Library in Java. Brown University. Available at http://www.jdsl.org/.

[14] S. D. Kamvar, T. J. Haveliwala, C. D. Manning, G. H. Golub. Extrapolation Methods for Accelerating Pagerank Computations. In *Proceedings of the Twelfth International World Wide Web Conference*, May, 2003.

[15] A. N. Langville, C. D. Meyer. Deeper Inside Pagerank. Department of Mathematics, N. Carolina State University, 2003.

[16] V. Muthusamy. An Introduction to Peer-to-Peer Networks. Presentation for MIE456 – Information Systems Infrastructure II. 2003.

[17] L. Page, S. Brin, R. Motwani, T. Winograd. The Pagerank Citation Ranking: Bringing Order to the Web. S*tanford Digital Libraries Working Paper*, 1998.

[18] Pastry: A substrate for peer-to-peer applications. Available at http://research.microsoft.com/~antr/Pastry/

[19] L. Pretto. A Theoretical Analysis of Google's Pagerank. In SPIRE 2002: 131-144. Lecture Notes in Computer Science. 2002.

[20] S. Ratnasamy. A Scalable Content-Addressable Network. Ph.D. Thesis. 2002.

[21] A. Rowstron, P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350. 2001.

[22] K. Sankaralingam, S. Sethumadhavan, J. C. Brown. Distributed Pagerank for P2P Systems. *12th International Symposium on High Performance Distributed Computing (HPDC)*. 2003.

[23] S. M. Shi, J. Yu, G. W. Yang, D.X. Wang. Distributed Page Ranking in Structured P2P Networks. In *Proceedings of the International Conference on Parallel Processing.* IEEE, 2003.

[24] J. C. Strikwerda. A Convergence Theorem for Chaotic Asynchronous Relaxation. In Linear Algebra and its Applications, 253(1997): 15-24.

[25] D. C. Verma. Legitimate Applications of Peer-to-Peer Networks. John Wiley & Sons, 2004.

[26] Y. Wang, D. J. DeWitt. Computing Pagerank in a Distributed Internet Search System. In *Proceedings of the 30th VLDB Conference,* Toronto, Canada. 2004.

[27] WebGraph Framework. Available at http://webgraph.dsi.unimi.it (University of Milan).

[28] B. Yang, H. Garcia-Molina. Efficient Search in Peer-to-Peer Networks. In Proceedings of the International Conference on Distributed Computing Systems. IEEE Press. 2002.