ΔΗΜΙΟΥΡΓΙΑ ΑΝΤΙΓΡΑΦΩΝ XML ΑΡΧΕΙΩΝ ΜΕ ΧΡΗΣΗ ΕΥΡΕΤΗΡΙΩΝ ΑΝΤΙΓΡΑΦΗΣ ΣΕ ΑΔΟΜΗΤΑ ΣΥΣΤΗΜΑΤΑ ΟΜΟΤΙΜΩΝ ΚΟΜΒΩΝ

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Παναγιώτη Σκυβαλίδα

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Απρίλιος 2007

DEDICATION

To my parents Giorgo and Evaggelia…

ACKNOWLEDGMENTS

I would like to thank my supervisor professor Evaggelia Pitoura for the help, support and patience she showed during the elaboration of this thesis. I would also like to thank my parents and my sister for the moral and financial support they provided me during the whole time period of my studies. Finally, I would like to thank all my friends for making these years in Ioannina unforgettable.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Skyvalidas, Panagiotis. MSc, Computer Science Department, University of Ioannina, Greece. April, 2007. Replication of XML Documents in Unstructured P2P Systems. Thesis Supervisor: Pitoura Evaggelia.

Peer-to-peer (p2p) systems have attracted considerable attention as a means of sharing content among large and dynamic communities of nodes. A central issue in p2p systems is locating the nodes that hold data of interest. There have been various proposals towards building overlays to support efficient content location. Such proposals vary from building rigid topologies and placing data on specific nodes in the overlay to unstructured networks with no correlation between the node content and its position in the overlay. In all types of overlays, content replication results in reducing the latency of lookups.

Motivated by the fact that XML is increasingly being used in data intensive applications, in this work, we study replication in unstructured p2p systems where participating nodes share content stored in XML. We consider XML replication for both passive and proactive protocols. XML documents have a hierarchical structure and thus, different fragments of an XML document can have different access frequencies. We show that replicating items at the *fragment level* is preferable to replicating whole documents.

For proactive replication, we introduce a new data structure that we call replication routing index. For a peer *p*, a *Replication Routing Index (RepRI)* has one entry for each file that *p* has processed queries for. Each entry keeps statistics about the requests that *p* has received for the specific file through its adjacent edges. Our replication strategy uses these indexes to decide whether to maintain a copy locally or forward it along a path. A Replication Routing Index for XML, termed RepRIX,

maintains statistics for fragments. RepRIX allows us to fine-tune the unit of replication, so that fragments of the same document can have different numbers of replicas. Further, it allows us to push fragments closer to their requesters. RepRIX entries are also used as hints during lookup to direct nodes towards paths that most probably hold replicas of the requested items. We also present experimental results of the deployment of our indexes in a dynamic unstructured peer-to-peer system.

ΠΕΡΙΛΗΨΗ

Παναγιώτης Σκυβαλίδας του Γεωργίου και της Ευαγγελίας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Απρίλιος, 2007. Δημιουργία Αντιγράφων XML Αρχείων σε Αδόμητα Συστήματα Ομότιμων Κόμβων. Επιβλέπουσα: Ευαγγελία Πιτουρά.

Τα τελευταία χρόνια, στο χώρο του διαδικτύου, έχει παρατηρηθεί αυξημένο ενδιαφέρον, γύρω από τα *Συστήματα Ομότιμων Κόμβων* (*p2p*). Τα p2p συστήματα, αποτελούν εφαρμογές που επιτρέπουν και διευκολύνουν το διαμοιρασμό δεδομένων μεταξύ μεγάλων και δυναμικών κοινοτήτων από συμμετέχοντες κόμβους. Το βασικό πρόβλημα γύρω από τα p2p συστήματα είναι ο αποδοτικός εντοπισμός των δεδομένων για τα οποία ένας κόμβος ενδιαφέρεται. Οι λύσεις που έχουν προταθεί περιλαμβάνουν τη δημιουργία δομημένων συστημάτων, στα οποία τα δεδομένα τοποθετούνται σε συγκεκριμένους κόμβους, καθώς επίσης και αδόμητων συστημάτων στα οποία δεν υπάρχει συσχετισμός ανάμεσα στα δεδομένα ενός κόμβου και στη θέση του στο δίκτυο. Ανεξαρτήτως της τοπολογίας του δικτύου, αυτό που έχει αποδειχθεί είναι ότι η δημιουργία αντιγράφων των δεδομένων και η διανομή τους στο δίκτυο συμβάλει σημαντικά στη βελτίωση της απόδοσης του συστήματος.

Στην παρούσα εργασία και έχοντας ως κίνητρο την σταδιακή καθιέρωση της XML ως πρότυπο για την αναπαράσταση και διακίνηση των δεδομένων στο διαδίκτυο, μελετούμε την δημιουργία αντιγράφων σε αδόμητα p2p συστήματα στα οποία οι συμμετέχοντες κόμβοι διαμοιράζονται XML αρχεία. Τα XML αρχεία ακολουθούν μία ιεραρχική δομή με συνέπεια διαφορετικά τμήματα ενός αρχείου να έχουν διαφορετικές συχνότητες προσπέλασης. Αυτό που ισχυριζόμαστε είναι ότι η δημιουργία αντιγράφων τμημάτων ενός αρχείου είναι προτιμότερη από την αντιγραφή ολόκληρου του αρχείου στις περιπτώσεις κατά τις οποίες κάποια από τα

τμήματά του δεν ενδιαφέρουν τους συμμετέχοντες κόμβους. Στην παρούσα εργασία παρουσιάζουμε μία νέα δομή δεδομένων την οποία ονομάζουμε *Replication Routing Index (RepRI)*. Ένα Replication Routing Index ενός κόμβου έχει μία είσοδο για κάθε αρχείο για το οποίο έχει επεξεργαστεί ερώτηση με στατιστικά σχετικά με τις αιτήσεις που έχει δεχθεί γι αυτό από κάθε προσκείμενη ακμή του. Ένας κόμβος χρησιμοποιεί αυτή τη δομή για να αποφασίσει αν θα πρέπει να δημιουργήσει αντίγραφο κάποιου αρχείου του και προς σε ποια κατεύθυνση να το προωθήσει. Ένα Replication Routing Index για XML αρχεία καλείται *RepRIX* και διατηρεί στατιστικά για τμήματα αρχείων. Το RepRIX μας επιτρέπει να ρυθμίσουμε τη μονάδα αντιγραφής, έτσι ώστε διαφορετικά τμήματα ενός αρχείου να μπορούν να έχουν διαφορετικό αριθμό αντιγράφων. Επίσης, μας επιτρέπει να προωθήσουμε τα αντίγραφα πιο κοντά στις πηγές ενδιαφέροντος. Το RepRIX μπορεί επιπλέον να συμβάλει στον αποδοτικότερο εντοπισμό ενός αρχείου, καθώς μπορεί να οδηγήσει ένα κόμβο στο να προωθήσει μία ερώτηση προς μία κατεύθυνση η οποία έχει πολλές πιθανότητες να οδηγήσει σε επιτυχημένη αναζήτηση.

CHAPTER 1. INTRODUCTION

1.1. Introduction

Peer-to-peer (p2p) systems have attracted a lot of attention as a means of data sharing among a large and dynamic population of nodes. P2p overlay networks are distributed systems in nature, without any hierarchical organization or centralized control. Peers form self-organizing networks that are overlaid on the Internet Protocol (IP) networks, offering a mix of various features such as robust wide-area routing architecture, efficient search of data items, selection of nearby peers, redundant storage, permanence, trust and authentication, anonymity, massive scalability and fault tolerance. P2p overlay systems go beyond services offered by client-server systems by having symmetry in roles where a client may also be a server. It allows access to its resources by other systems and supports resource-sharing, which requires fault-tolerance, self-organization and massive scalability properties. Here, we focus on unstructured p2p systems. These are systems in which there is neither a centralized directory nor any precise control over the network topology or data placement. The network is formed by nodes joining the network following some loose rules. The resultant topology has certain properties, but the placement of data is not based on any knowledge of the topology (as it is in structured designs). To find an item, a node queries its neighbors. The most typical query method is flooding, where the query is propagated to all neighbors within a certain radius. These unstructured designs are extremely resilient to nodes entering and leaving the system.

A central issue in p2p systems is locating the nodes that hold data of interest. Since knowing all other peers and their content is not feasible, each peer connects (knows about) a small number of other peers, thus forming an overlay network. There have been various proposals towards building overlays that support efficient content location. Such proposals vary from building rigid topologies and placing data on specific nodes in the overlay to unstructured networks with no correlation between the nodes content and its position in the overlay.

In all types of overlays, content replication results in reducing the latency of search. Various replication techniques have been proposed that can be roughly categorized as passive or proactive. With *passive replication*, items are replicated after they are successfully located after a request. A commonly used passive replication scheme, path replication, has been proven to produce the optimal number of copies under specific conditions [1, 2]. With path replication, data items are cached along the search path after an item is located. With *proactive replication*, holders of data items initiate the creation of replicas not necessarily after a request. Many issues regarding replication in p2p systems remain open. One such issue is where to place copies, since path replication tends to cluster copies on search paths.

We focus on p2p systems where participating nodes share content stored in XML documents. XML [3] has evolved as the new standard for the representation and exchange of semistructured data on the Internet. Several application domains for XML already show that XML is inherently distributed on the Web, for example, Web services that use XML-based descriptions in WSDL and exchange XML messages with SOAP, e-commerce and e-business, collaborative authoring of large electronic documents and management of large-scale network directories. All these applications demonstrate that much of the traffic and data available in the Internet are already represented in XML format. Thus, it is natural to assume that much of the data in a p2p system is already represented in XML format. XML documents have a hierarchical structure. Query languages on XML documents, such as XQuery [4] and XPath, exploit this structure through path-based expressions. Thus different fragments of an XML document may have different access frequencies.

1.2. Scope of Thesis

In this work, we show that replicating items at the fragment level is preferable than replicating whole XML documents. To achieve this and still maintain good response time for queries at different fragments of a document, we replicate the content of popular fragments and maintain links to the original document for the rest. We consider fragment replication for both passive and proactive protocols. For proactive replication, we introduce a new data structure that we call replication routing index. For a peer *p*, a replication routing index (RepRI) has one entry for each file that *p* has processed queries for. Each entry keeps statistics about the requests that *p* has received for the specific file through its adjacent edges. Our replication strategy uses these indexes to decide whether to maintain a copy locally or forward it along a path. For XML documents, the replication routing indexes, termed REpRIX, maintain statistics for fragments. REpRIX allows us to fine-tune the unit of replication, so that fragments of the same document may have different number of replicas. Further, it allows us to push fragments closer to their requesters. REpRIX entries are also used as hints during search to direct nodes towards paths that most probably hold replicas of the requested items.

We experimentally compare both proactive and passive variations of fragment replication. Both types of fragment replication outperform whole document replication resulting in increasing the percentage of items located and reducing the required steps for doing so. Proactive replication with hints is shown to work better than passive replication. We also present experimental results of the deployment of our indexes in a dynamic unstructured peer-to-peer system.

1.3. Thesis Outline

The remainder of this work is structured as follows. Chapter 2 summarizes in brief the main issues about content replication in p2p systems and introduces proactive replication using replication indexes. Chapter 3 extends replication indexes for XML documents. It presents our approach for fragment replication and describes the two implemented techniques *Skeleton replication* and *Subtree replication*. In chapter 4 we present our experimental results from the evaluation of our approach and its

comparison with existing replication strategies. Chapter 5 presents related research on content replication in p2p systems and fragmentation of XML documents. At last, chapter 6 concludes this work and presents the open issues for future work.

CHAPTER 2. PROACTIVE REPLICATION

2.1. Replication in Unstructured p2p Systems

P2p content distribution systems rely on the replication of content on more than one peer for improving the availability of content, enhancing performance, and resisting censorship attempts. Replication is traditionally understood as a static configuration for the placement of copies of data items, for the purpose of increased reliability and availability as well as better load sharing. In large-scale distributed systems that rely more on self-organization rather than carefully planned administration, such as p2p systems, replication is seen as a dynamic mechanism. A new copy may be created when an existing copy fails (transiently or permanently) or when some peer becomes overloaded, copies may be migrated, or replicas may simply be the result of cached copies being kept at peers for a longer time period. The critical issues in dynamic replication are:

- Determining the number of replicas that we want to have for a given data item, based on goals for reliability, availability, and performance.
- Determining on which peers we should place these replicas.
- Designing a strategy for adjusting the replica placement upon certain events such as peer failures or load bursts.
- Designing a mechanism and a strategy for keeping replicas updated and consistent.

Various replication techniques have been proposed that can be roughly categorized as passive or proactive. Passive replication occurs naturally in p2p systems as peers request and copy content from one another. Two easily implementable, passive replication strategies are owner and path replication. With *owner replication*, when a search is successful, the requested data is stored at the requester peer only. When *path replication* is used, information is kept along the paths that a query traverses. After a successful search, the requested data is replicated and stored at all peers on the path from the provider peer to the requester peer. On the other hand, with proactive replication, peers can create replicas of their data items without an explicit request for them. The reasons for disseminating data in a proactive manner are the following:

- To improve the response time of search requests, by means of additional replicas. In blind search situations with limited request flooding in unstructured networks, the additional replicas may even be needed to improve the probability of a successful search.
- To improve the load balance in the network and thus increase the overall throughput of the entire system. This assumes that additional replicas can effectively be considered in the request routing.
- To improve the availability of data items, in the presence of frequent peer outages and churn.
- To improve the reliability of the system, in the sense that it guarantees higher probability of data durability, i.e., not losing a data item regardless what permanent peer failures may occur.

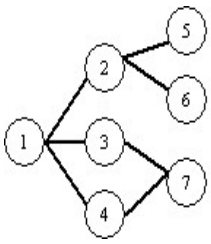
2.2. Replication Routing Indexes (REpRI)

Here, we propose a new replication strategy that creates and places replicas on the network, in a proactive manner. We consider first the case where peers store simple data files without any specified structure. Searching is done using keyword-based queries that refer to the file names. We introduce a new data structure that we call *Replication Routing Index (RepRI)*. The *RepRI(p)* of a peer *p* has one entry for each file for which peer *p* has processed requests. Our replication strategy uses these indexes to decide whether a peer should replicate some of its files and forward them

Table 2.1 RepRI for Simple Data Files

Filename	LReq	Req_1	Hop_1	…	…	Req_k	Hop_k	Ownership

to certain directions. With RepRI the decision about creation and placement of replicas on the network is based on the use of distributed information. Each peer in the system keeps some statistics on the queries it processes. Using this information it dynamically decides if replicas of its local files should be created and sent to another peer in the network. Our approach tends to create and move replicas towards the direction that are actually "needed".



Filename	LReq	Req_1	Hop_1	Req_2	Hop_2	Req_3	Hop_3	Ownership
Talk.mp3	0	8	4	2	6	4	4	1
Sun.jpg	4	3	2	0	7	1	3	0
Precious.mp3	0	5	1	2	5	4	5	1

Figure 2.1 Instance of RepRI for Peer 1

For a peer *p* with *k* neighbors, each entry is of the form (*Filename*, *LReq*, Req_1, Hop_1, …, Req_k, Hop_k, *Ownership*), shown in table 2.1, where *Filename* is the name of the file, *LReq* is the number of queries for this file initiated by peer *p*, Req_i *and* Hop_i with $i = 1,\ldots, k$, are respectively the number of queries that were forwarded to peer *p* by its neighbor *i* and the corresponding average number of hops required for the queries to reach *p*. At last, the field *Ownership* takes the values 1 and 0 depending on whether the file is stored locally at *p* or not. In Figure 2.1, we show the RepRI maintained by peer with id 1, which has three neighbors, 2, 3 and 4. Peer 1 has processed queries for two local files, "Talk.mp3" and "Sun.jpg" and for one file "Precious.mp3" stored at some other peer. When *p* receives a query, forwarded by its neighbor *i*, RepRI(p) is scanned and when the relative entry is found (*p* has processed queries for that file before), the value of the field Req_i is incremented by 1 and the field Hop_i is updated. If there is not an entry for the file, a new one is inserted. Req_i takes the value 1, Hop_i the number of hops required for the query to reach *p*, the field Ownership takes the

value 1 or 0 depending on whether the file is stored locally at *p* or not, while the other fields are set to 0.

The decision which file to replicate and towards which direction is based on both the popularity of the file and the cost for locating it. In particular, each peer *p* with *k* neighbors calculates the *replication utility*, *ru(f)* of a file *f* as follows:

$$ru(f) = \alpha * popularity\_factor + (1 - \alpha) * distance\_factor$$

where popularity_factor is defined as:

$$popularity\_factor = \sum_{i=1,\ldots,k} Req_i \,/\, Max(\sum_{i=1,\ldots,k} Req_i)$$

and distance_factor is defined as:

$$distance\_factor = Avg(Hop_i) / TTL, \text{ with } i = 1,\ldots,k$$

$Max(\sum_{i=1,\ldots,k} Req_i)$ corresponds to the total number of requests that *p* received from its neighbors for the most popular file in RepRI(p) and $Avg(Hop_i) = \sum_{i=1,\ldots,k} Hop_i / k$. The weight α, $0 \leq \alpha \leq 1$, is a tuning parameter that determines how much each of these two factors affects the replication decision. When all files have similar query probabilities, a small value for α favors the files with high average search size. On the contrary, when we have different query probabilities, some files become popular. In order to favor these files, we increase the value of α. This way popular files are replicated more easily speeding up the search process. The larger the value of α, the more efficient the search for popular files.

Periodically, a peer *p* decides to create replicas of all files *f* stored locally (field Ownership has value 1) that have replication utility greater than or equal to the average replication utility (aru). For each such file, peer *p* sends a replication message to its neighbor *m* having the maximum corresponding Req_m. After completing this replication phase, the entries for all files *f* in the RepRI are reset to 0. Before resetting them to 0 though, the values of the entries are copied in an auxiliary structure. The

reason for doing this is because we don't want to lose information about files, for the last period of time, since this information is used in the replacement policy, as we will describe below.

The duration of the period of the replication procedure depends on the query workload. A large value leads to making more informed decisions based on sufficiently large samples of requests and ignores popularity fluctuations that may be caused by random variations in the query workload. Furthermore, it reduces the associated network overhead. On the other hand, the system adapts to workload changes less promptly. Also, note that resetting the RepRI entries of the peer that initiated the replication procedure provides a simple yet effective aging scheme.

When *p* receives a replication message for a file *f*, it executes the following tasks. It scans RepRI(p) and locates the entry that corresponds to the file the name of which is included in the replication message. If the value of the field LReq is greater than 0, then the peer decides to store the replica. If LReq is equal to 0, then *p* calculates ru(f) and compares it with aru. If ru(f) is greater than or equal to aru then the replica is stored. After storing a replica, the corresponding field Ownership is set to 1. We see that a peer decides to store a replica in two cases. When it has initiated queries for it and when the file is considered *hot* based on our replication criterion. Finally, the peer checks if it should further forward the replication message to any of its neighbors, by checking the fields Req_i. The neighbor *m* having the maximum corresponding Req_m receives the replication message. If all fields Req_i have values equal to 0 then the forwarding procedure is terminated. In this point we have to mention that the values that are checked are the values acquired during the last period.

2.3. RepRI-Based Replacement and Routing

Besides replication decisions, RepRI affects the replacement policy we use for replicas, when we consider the case of limited storage capacity. When a peer *p* that has used all its available space decides to store a new replica, it has to replace it with an older one. Common replacement policies that can be used in this case include randomly choosing an entry for replacement or following a fifo (first in first out)

strategy. In our approach, we use RepRI to decide which item gathers the prerequisites to be replaced. More specifically, *p* scans RepRI(p) and calculates for each existing replica *r* the replication utility ru(r). The replica with the smallest replication utility is the one that is replaced. The statistics for each replica refer to the previous and not the current period, so the values that we use to calculate each ru(r) are the ones kept at the auxiliary structure. The reason for keeping these values is to avoid evicting a replica which was considered hot during the previous period, after resetting RepRI.

RepRI can also be used to further improve the routing process by maintaining the *direction* of the source of a file. When a peer forwards a replication message to one of its neighbors, without storing the replica, it can set the corresponding field Req_i to a hint value that indicates that the replica has been copied along this direction. Thus, the next time the peer receives a query for that file it knows where to forward it. Networks that use random walkers or flooding as their search mechanism can significantly benefit from this approach.

2.4. Discussion

Previous work on replication [1, 2] has shown that square-root replication is the optimal way to allocate replicas so that the average search size is minimized. If a p2p system uses the k-walker random walk as the search algorithm, then on average, the number of peers between the requester peer and the provider peer is 1/k of the total peers visited. Path replication in this system should result in square-root distribution. However, path replication tends to replicate files to peers that are topologically along the same path. In p2p networks with random topologies, where peers are randomly chosen to initiate queries, we would like to avoid the topological impact of path replication. Proactive replication using RepRI allow us for a more random distribution of replicas.

CHAPTER 3. XML MODEL

In this chapter we extend Replication Routing Indexes for XML documents. We present our approach for fragment replication and describe the two implemented techniques *Skeleton replication* and *Subtree replication*.

3.1. Motivation

In most p2p systems, different users and applications employ various formats and schemas to describe their data. A user is usually unaware of the schemas remote peers use. Moreover, some application domains use sensitive data that are required not to be exposed to all users for privacy reasons. Therefore, there is a need for a query language that can work with incomplete or no-schema knowledge but also capture whatever semantic knowledge is available. The flexibility of XML in representing heterogeneous data that follow different schemas makes it suitable for distributed applications where the data are either native XML documents or XML descriptions of data or services that are represented in various formats in the underlying sources.

With regards to the query language, in most p2p systems, users specify the data they are interested in through simple keyword-based queries. These keywords are matched

against the names of the shared files and results are returned to the user. Often, most results returned are not relevant to what the user is interested in. Thus, new more expressive languages are needed to describe and query the shared data. XML seems to be a promising candidate in this direction, since it enables more precise search based on context.

3.2. Data and Query Model

In our data model, an XML document is represented by a rooted labeled tree. Labels correspond to XML tags, tree nodes correspond to document elements, while edges represent direct element-subelement relationships. Figure 3.1 shows an XML description of a library catalog provided by a node and the corresponding XML tree.

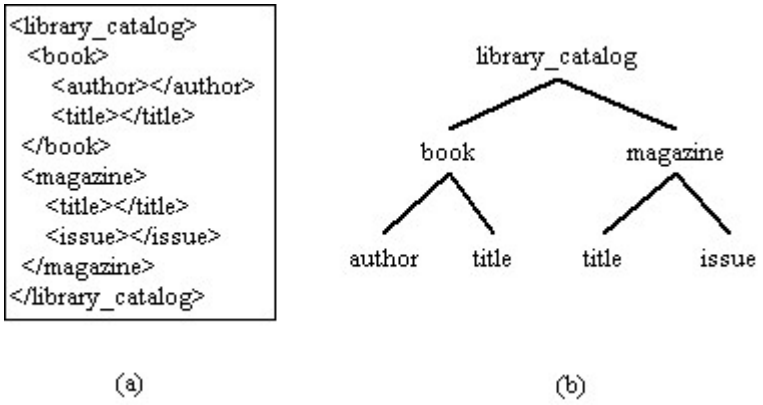


Figure 3.1 (a) Example of an XML Document (b) The Corresponding Tree

In our distribution scheme, we assume that a query can be issued at any peer and query results are delivered to that peer. Since we are interested in querying the structure of documents, we do not use simple keyword queries but path patterns, called *path queries,* to be matched against XML documents. Path queries are simple path expressions in an XPath-like query language.

Definition 1. (path query) A path query of length n has the form "$p_1\ e_1\ p_2\ e_2\ \ldots\ p_n\ e_n$" where each e_i is an element name or the wildcard operator * and each p_i is either / or // denoting respectively parent-child and ancestor-descendant traversal.

A path query q is evaluated at a node u in an XML tree T and its result is the set of nodes of T reachable via q from u.

/ **(child operator):** When used at the beginning of the path expression it refers to the root of the XML document. The child operator is used to specify the next child to select. For example, the evaluation of the path query /library_catolg/book (Figure 3.1) starts at the root of the XML document, selects the library_catalog node and then the book node. This will return all the book child nodes of the library_catalog node, which is just one in our example.
// **(descendant operator):** The descendant operator indicates to include all descendant nodes in the search. Using the operator at the beginning of the path expression means you start from the root of the XML document. The path query /library_catalog//title (Figure 3.1) returns all the title nodes.
*** (wildcard operator):** The wildcard operator finds any node. The expression "/*" finds any node under the root. The path query /library_catalog/* (Figure 3.1) returns all nodes under the library_catalog node, which in our example are the book node and the magazine node.

Definition 2. (path subsumption) A path query $q_1 = p_1\ a_1\ p_2\ a_2\ \ldots\ p_n\ a_n$ is subsumed by the path query $q_2 = p_1\ b_1\ p_2\ b_2\ \ldots\ p_m\ b_m$ if for each path expression $/a_1\ /a_2\ /\ldots/a_k$, extracted from q_1 exists path expression $/b_1\ /b_2\ /\ldots/b_l$, extracted by q_2 such that $l \leq k$ it holds $a_i = b_i$ for each i, $i = 1,\ldots,l$.

For a query *q* and a document *d*, we say that *q* is satisfied by *d* if the path expression forming the query exists in the document. Peers that store documents that match the query are called the *matching peers*.

3.3. Fragmentation of XML Documents

We allow an XML tree T to be decomposed into a collection of trees, called *fragments*, which can be distributed and stored at several peers. An XML fragment F_i is a subtree of T rooted at some node f of T. Each fragment is represented by a simple

path expression starting from the root node r of T and leading to f. In Figure 3.2, we see an example of an XML fragment that is represented by the path "*/A/C*".

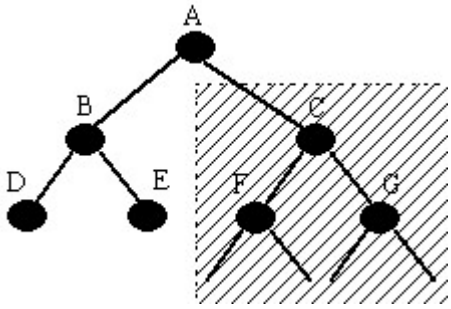


Figure 3.2 An XML Fragment Represented by the Path "/A/C"

Using path queries, peers can query a subset of the information included in an XML document. Thus, different fragments of a document may have different access frequencies. In our approach, we try to exploit the fragmentation of XML documents and replicate only the minimum required information from them. In particular, there are cases where peers are only interested in a specific fragment of a document. Thus, it is better for these peers to receive a replica of that fragment instead of the whole document. This approach is also useful when the storage space of peers is limited. A specific fragment of a document corresponds to a specific amount of data. If we traverse the tree of an XML document following the path that represents a fragment of it, we reach at the fragment-root node. The leaf nodes of the subtree rooted at that node contain all the data we are interested in.

3.4. Fragment Replication

In order to support data replication, we allow peers to replicate their documents or just fragments of them depending on the queries they have processed. We assume that the documents stored initially at peers are not fragmented. Fragmentation of documents is the result of the replication strategy. In order to be able to discern the origin of a replicated fragment, all elements of the initial documents have unique identifiers. This is done especially for handling possible updates. When part of the data, in the original document, is updated, having these ids help us distinguish the fragments originated from that specific document and update them as well. For brevity in our examples we

have omitted most of these ids. The replicas that are created correspond to fragments defined by path queries. We have implemented two techniques for replica creation, called *skeleton replication* and *subtree replication* that are described below.

3.4.1. Skeleton Replication

In skeleton replication, replicas have similar structure with the original documents. The only difference is that only part of the data of the original document is replicated. The approach we follow when we create a replica is the following. For a path expression, $/a_1\ /a_2\ /\ldots\ /a_n$, we copy the skeleton (element hierarchy) of the original document from the root node a_1 to the fragment-root node a_n. For each element node a_i of the original document that has siblings, then the corresponding replica contains along with the sibling elements an *external link* to the original document. Finally, we copy the data contained at the subtree defined by the path query. Data that corresponds to elements that are not included in the related path is not replicated.

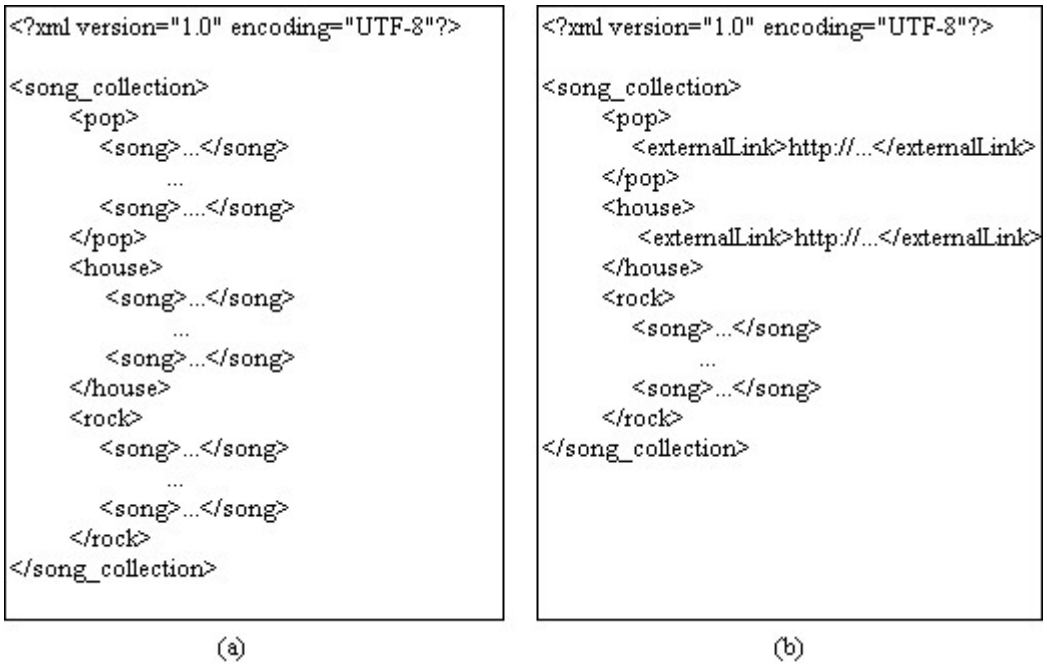


Figure 3.3 (a) Example of an XML Document (b) Skeleton Replica Corresponding to the Fragment Defined by the Path Expression "/song_collection/rock/*"

External links are special elements identified by the tag name *externalLink*. External links indicate where the data of the element, to which they belong, is located. They can actually be viewed as an intentional description of this missing data and give the means to obtain it if needed. From the XML tree perspective, external links play the role of some kind of external nodes. In Figure 3.3, we see an example of an XML document and a replica corresponding to the path query "*/song_collection/rock/**". As we see, only the data contained at the *rock* element is replicated. Data contained at the sibling elements *pop* and *house* is not replicated. In its position instead, an external link is placed, pointing to the original document.

Query Processing Algorithm

```
Input: query path q, current peer p
Output: set of XML nodes that satisfies q

 1  Begin
 2  fr_root := fragment-root node of q
 3  res : = {}
 4  for each document di in p do
 5      if match(di, q) then
 6          if fr_root has children then
 7              for each child ni of fr_root do
 8                  if ni has external link then //not all data that satisfy q present
 9                      link : = external link url
10                      forward(q, link)
11                      break
12                  else // query satisfied
13                      res : = all children of fr_root
14                      break
15              endfor
16          else // fr_root leaf node
17              res : = fr_root
18 endfor
19 return res
20 End
```

Figure 3.4 Query Processing Algorithm for Skeleton Replication

When a peer receives a path query it checks if it can answer it, in order to complete the lookup. More precisely, for each document it possesses, the path query is matched against the corresponding XML tree. If there is not any match the query is forwarded

further to some neighbouring peer. If there is a match, then the peer executes the following steps.

- If the fragment-root node defined by the path query, is a leaf node then the query is satisfied by this node, the data residing at this node is returned to the requester peer and the processing is stopped.
- Else if the fragment-root node defines a subtree, then its child nodes are checked.
  - If none of them contains an external link, it means that all the requested data is present at the current document. The data is collected, returned to the requester peer and the processing is stopped.
  - In the case where one or more child nodes contain an external link a different approach is followed. The presence of an external link indicates that the current document is a fragment of another document and that the data defined by the corresponding external node was not replicated. Thus, the current fragment cannot completely satisfy the query. For this reason, the peer uses the link to forward the query to the peer that stores the document which contains all the requested data.

The query processing algorithm is shown at Figure 3.4. Consider the XML tree shown in Figure 3.2. If the peer that stores the corresponding document decides to create a replica for the fragment defined by the path *"/A/B/D"*, then the replica will also contain the element E, due to the constraint we pose. The data residing at node E is not replicated. In its position instead, an external link is placed to the location of the original document (ip address of the peer that stores the document). Consider the case that the peer that stores the replica has to evaluate the path query *"/A/B"*. The requested data consist of the nodes D and E. If the replica didn't contain the external link for the element E the peer would answer the query returning to the requester peer only the data residing at node D. With our approach, the peer uses the link to forward the query *"/A/B"* to the peer that stores the original document ensuring the correctness and the completeness of the query evaluation process.

3.4.2. Subtree Replication

In subtree replication, a different approach is followed. The replicas that are created contain only the fragment defined by a path query and not the whole skeleton of the original document. Viewing it from the XML tree perspective, a replica consists of the subtree (of the original document) rooted at the fragment-root node.

```
<?xml version="1.0" encoding="UTF-8"?>

<rock path="song_collection/rock" parent node="id" hasSiblings="yes">
    <song>...</song>
            ...
    <song>...</song>
</rock>
```

Figure 3.5 Subtree Replica Corresponding to the Fragment Defined by the Path Expression "/song_collection/rock/*"

In order to be able to evaluate a path query over a replica, three attributes are added to the root element, the *path* attribute, the *parent node* attribute and the *hasSiblings* attribute. The path attribute contains the sequence of element names from the original document's root node to the replica's root node. As we mentioned before all elements have a unique id. Thus, the parent node attribute contains the parent node's id in order to achieve cohesion between the original document and the replica. Finally, the hasSiblings attribute takes the values *yes* or *no* and indicates whether the replica's root node has any sibling nodes or not, which have not been replicated, respectively. As an example consider the XML document shown in Figure 3.3. The replica corresponding to the path query "/song_collection/rock/*" has now the form shown in Figure 3.5.

The query processing procedure for a path query *q* is described below. When a peer receives the query it tries to match it against the documents it possesses. If none of the documents matches the path expression, the query is further forwarded to a neighbouring peer. In the case of a match:

- If the current document is not a replica (meaning that the root element does not contain the path attribute) then all the requested data is gathered and sent back to the requester peer and the processing procedure is terminated.
- Else if the current document is a replica then:
  - If the path query q is subsumed by the value of the path attribute then the replica contains all the requested data which are gathered and sent back to the requester peer.
  - Else if the value of the path attribute is subsumed by the path query q then the value of the hasSiblings attribute is checked. If it is *no*, then the replica satisfies the query since all the requested data is present. If it is *yes*, then part of the requested data is missing, the replica cannot satisfy the query, so the next document is parsed.

3.5. Replication Using REpRIX

For XML documents, the replication routing indexes, termed REpRIX, can be modified to find the best unit for replication. The basic modification that takes place, when dealing with XML documents, is on the information that is maintained by peers. Instead of maintaining statistics for whole documents, the entries of a REpRIX now correspond to paths, representing fragments that the peer has processed queries for. The field Ownership takes the value 1 if the path, representing the fragment, is contained in a local file.

In skeleton replication, the presence of external links, in the replicated fragments, requires an addition in the way requests are handled. As mentioned before, during a lookup, a query might follow a number of external links until it reaches the peer holding the requested data. Since the peer that uses an external link to forward a query might not be a neighbor of the peer that receives it, current structure of RepRIX is not sufficient to handle these kinds of situations. For this reason, whenever a peer receives a query from a direction outside its neighbouring list, it adds the necessary extra fields to keep track of these requests. Following this approach peers cache the location of peers holding data that a part of the network is interested in but is not able

to reach it unless an external link is used. Thus, during replication process the extra fields are also taken into account in order to create and send replicas via external links towards the direction they are considered hot.

As peers process queries, they use RepRIX to keep all the necessary information that would help them take the right replication decisions. Peers decide to replicate fragments of their documents based on the replication criterion we defined in chapter 2. When processing a query, before trying to match it against their documents, a peer *p* executes the following check:

- If there is an entry in RepRIX(p) for the related path, (peer has processed queries for that fragment in the past) then *p* just updates the appropriate fileds.
- Else if the path is not contained in RepRIX(p), a new entry is inserted. The corresponding fields take their initial values, while the rest of them are set to 0.

When updating the access frequency of fragments, path subsumption is not taken into account. The entries of the index are updated separately. Path dependencies have to be checked during the replication procedure. Entries in the RepRIX that correspond to paths that one subsumes the other are handled as one. In other words, if a peer has processed queries for paths p_i i=1,…,k, which are subsumed by path p, then the replication utility for the path p is calculated as follows:

$$ru(p) = \sum_{i=1,\ldots,k} ru(p_i), \text{ for each } p_i \text{ that is subsumed by p}$$

The replication message that is sent refers to the fragment that contains the others. The fragment that is created and replicated is represented by the path that corresponds to the larger subtree. For example consider the case that a peer has processed queries for paths "*/a/b/c*" and "*/a/b*". Since the path "*/a/b/c*" is subsumed by the path "*/a/b*" the fragment that is finally replicated is that defined by the path "*/a/b*".

```
<?xml version="1.0" encoding="UTF-8"?>

<music_catalog>
      <rock>
         <cd title = "Hotel"> </cd>
            …
         <cd title = "Exciter"> </cd>
      </rock>
      <pop>
         <cd title = "Up"> </cd>
            …
         <cd title = "Music"> </cd>
      </pop>
</music_catalog>
```

Figure 3.6 Document “music_catalog.xml”

3.6. Replacement Policy

Throughout this work, we assume that peers have limited storage capacity. For this reason, a replacement policy for replicas has to be used. Each time a peer decides to store a new replica the following check takes place, in order to avoid data redundancy. If the fragment that is going to be stored contains a fragment that is already stored, then the older one is discarded and the new one takes its place. We say that a fragment f_1 is contained in a fragment f_2, if the path expression representing f_1 is subsumed by the path expression representing f_2. For example, if a peer that already has a replica of a fragment represented by the path expression “*/a/b/c”*, decides to store a fragment represented by the path expression “*/a/b”*, then the replica represented by “*/a/b/c*” is replaced.

Replacement also takes place when the storage limit is reached. In this case the peer uses RepRIX to find the fragment which gathers the prerequisites to be replaced. As in the case of simple data files, replication utility is used to find the replica that is going to be replaced. The path with the smallest replication utility is matched against the replicas stored by the peer. The replica that matches the path is found and the fragment defined by the path is evicted. In fact, the data corresponding to that fragment are discarded and in its position instead, an external link is placed pointing to the document, the replica was originated from. This is done in order to maintain consistency among external links and ensure the correctness of the search process.

When a peer follows an external link, during a lookup operation, it may reach a replica that won't contain the requested data, as a result of the replacement strategy. However, another external link will be present to direct the lookup towards a document that might contain it. In the worst case scenario, the requested fragment would be evicted form all replicas. In this case, the lookup operation will end when we follow all the possible external links and finally reach at the original document.

However, since replicas have different sizes, their size has to be taken into account when the storage limit is reached. More specifically, when the new replica that is going to be stored is quite large, then evicting a replica with smaller size is not an adequate solution. In such a case, the peer replaces more than one fragment in order to make sufficient space for the new one. The fragments that are discarded are the ones with the lowest replication utilities. In general, it is preferable to replace two or three small fragments that are not considered hot than replacing a large hot one.

```
<?xml version="1.0" encoding="UTF-8"?>

<music_catalog>
      <rock>
         <cd title = "Hotel"> </cd>
            ...
         <cd title = "Exciter"> </cd>
      </rock>
      <pop>
         <externalLink>"link to music_catalog.xml" </externalLink>
      </pop>
</music_catalog>
```

Figure 3.7 Skeleton Replica Corresponding to Path "/music_catalog/rock/*"

3.7. Use of External Links

Consider the following example that shows another use of external links. Assume that a peer *p* stores among others the document "music_catalog.xml" shown in Figure 3.6. Assume as well that RepRI(p) has an entry for the path *"/music_catalog/rock/*"* and that the field ownership is set to 1. If the replication utility of this path allows for a replica to be created, the peer sends a replication message for the relative path to one

of its neighbors. The replica that is created contains only the information about rock cds, while information about pop cds is not replicated. This information is replaced by an external link that points to the source of the document “music_catalog.xml”. The replica is shown in Figure 3.7. After receiving the replication message, the peer first decides whether it should store the replica and then it continues forwarding the message. Now consider the case that a peer that has stored the replica shown in Figure 3.7 receives a query for path: *“/music_catalog/pop/*”*. After receiving the query, the peer checks its local documents to see if it has the requested data. While traversing the XML tree of the replica, to match the query, the peer will reach the external node, which refers to pop cds. The peer will use the external link, stored at that node of the tree to send the query directly to the peer storing the original document “music_catalog.xml”. Thus, external links can act as “jumps” at the search process, resulting to the reduction of the average number of hops required to answer a query and the reduction of the total number of messages exchanged among peers.

CHAPTER 4. EXPERIMENTAL EVALUATION

In order to evaluate our approach and prove its efficiency we performed a series of experiments under a simulation environment. In this chapter we present our experimental results.

4.1. Simulation Environment

All replication techniques were implemented in Java as components of the Peersim[5] simulator. Peersim is a simulator for unstructured p2p systems composed of many simple extendable and pluggable components, with a flexible configuration mechanism. All components of the simulator are completely interchangeable and specified by object-oriented programmatic interfaces, whose methods describe the expected behavior of a component. For our experiments we used the cycle-driven simulation engine of Peersim, meaning that simulation proceeds through time steps called cycles, in which all nodes get a chance to execute. Every simulation starts with an initialization phase. During the initialization phase, the topology of the network is formed and the documents are distributed randomly along the network. The documents used for the experiments were generated by the ToXgene[6] XML generator. ToXgene is a template-based tool for generating large, consistent synthetic collections of complex XML documents.

The queries we use are simple path queries extracted by the data set and the query load follows a Zipfian distribution. The search mechanism that is used is random walkers with a TTL parameter. The networks have random and power-law topologies,

with sizes ranging from 1000 to 10000 peers and average peer degree 8. Except for static networks, we also consider dynamic networks where peers can go offline for a certain time period. In all experiments, we consider that the system poses a constraint on the number of replicas that can be stored. Each peer has a limited storage capacity given as a system parameter.

Table 4.1 Simulation Parameters

Parameters	Default value	Range
Network size	5000	1000-10000
Random walkers	16	
File distribution	Random	
Query distribution	Zipf (alpha = 1.0)	alpha = [0.0, 2.0]
TTL	7	
Avg file size	4KB	3.5KB-4.5KB
Avg fragment size	1.3KB	0.6KB-3KB
Storage limit	16KB	8KB-64KB
Weight α	0.6	0.2-0.8

4.2. Experimental Results

The metrics that we are most interested in are the number of replicas created by each technique, the percentage of the successful queries, the average search size, that is, the average number of hops a query must travel to reach an answer and the amount of hard disk space occupied by replicas. All the reported values were averaged over multiple runs.

Network characteristics play an important role on the performance of the system. First of all, the network topology. In our experiments we used two different topologies, random graphs and power-law graphs. Power-law graphs seem to achieve better results, especially when they are combined with a random walk search strategy. However, in the case of failures and high churn rates, power-law graphs appear a certain disadvantage. A power-law network consists of a few peers with a high degree and a large number of peers with a low degree. In the case of path replication, the peer

with a high degree forwards much more data than a peer with a low degree, so that a large number of replications will occur at the peers with a high degree. Therefore, the storage load can be concentrated on a few high-degree peers, which thus play an important role in the system. When these peers fail or go offline for some period of time, the system requires a large amount of time to recover the previous performance standards.

Another factor that can affect in a high degree the performance of the system is the query distribution. When a uniform query distribution is used, the search cost becomes almost the same for all items. With zipf distribution the behavior of the system is different. As the parameter *a* of the zipf distribution increases, the search cost for hot items decreases significantly. Popular items are easier to be found and the percentage of successful queries is high. However, the search cost for cold items becomes bigger. Another characteristic of zipf distribution is that as *a* increases, the time required to adapt to a change in the query pattern increases, compared to the time required by a uniform query distribution. By shifting the query distribution we change the popularity of items. Items that were popular until that point are made unpopular and reversely, unpopular items become popular.

When fragment replication is performed, the storage capacity of peers and the access frequencies of fragments play an important role to the results of the replication strategies. When the storage capacity of peers is quite small then in order to achieve good performance, with respect to the average search cost and the percentage of successful queries, the size of popular fragments is required to be small. In other words, when most queries refer to relatively small fragments the system shows better performance. The reason for this is that increasing the access frequency of small fragments results in a greater number of replicas stored by each peer making them easier to be found.

4.2.1. Performance of RepRI for Simple Documents

In our first set of experiments we evaluated the performance of proactive replication using RepRI for the case of simple documents. We compared our approach with the

two most widely used strategies for passive replication, path replication and owner replication.

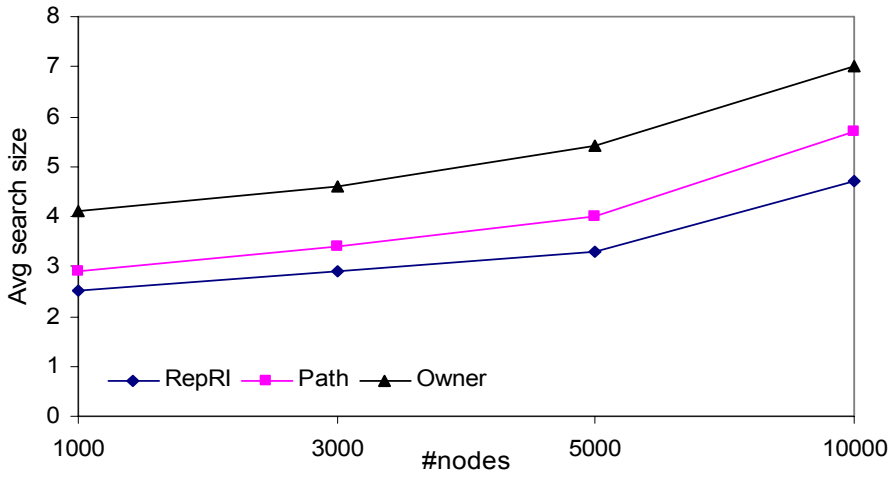


Figure 4.1 Average Search Size for Different Sizes of Netorks

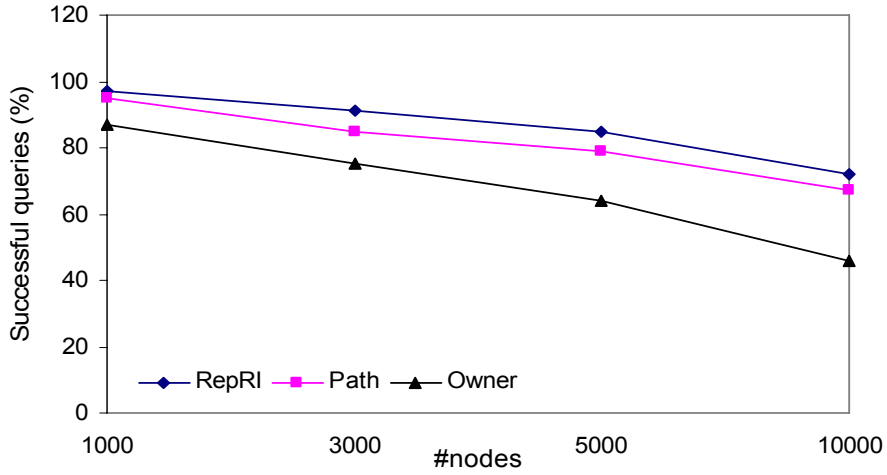


Figure 4.2 Percentage of Successful Queries for Different Sizes of Networks

Figures 4.1 and 4.2 show that proactive replication using RepRI outperforms both path and owner replication. RepRI replication achieves a lower average search size and increases the percentage of the successful queries. The gain in the performance is explained by the following two factors. The statistics maintained by each peer allow us to find a path with many requests for a document and push replicas towards that

direction. In addition, during the search process, hints are used to forward queries towards peers that are more likely to have an answer.

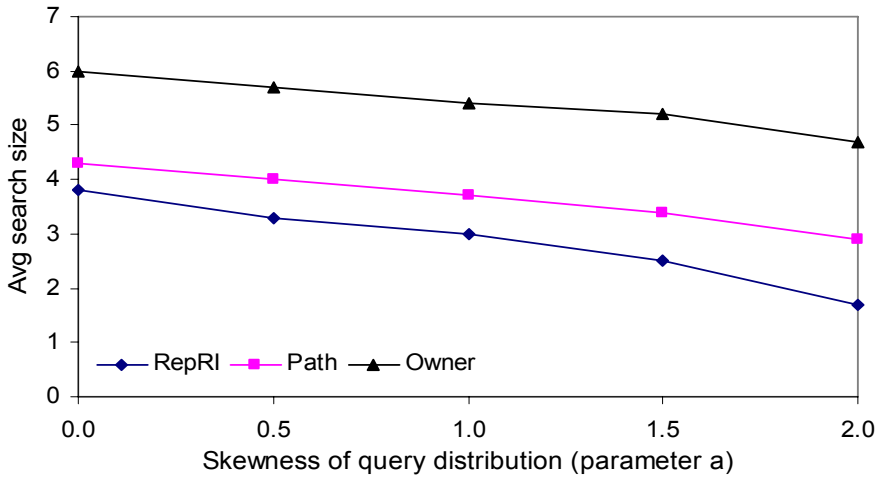


Figure 4.3 Average Search Size for Different Values of a of the Zipfian Query Distribution

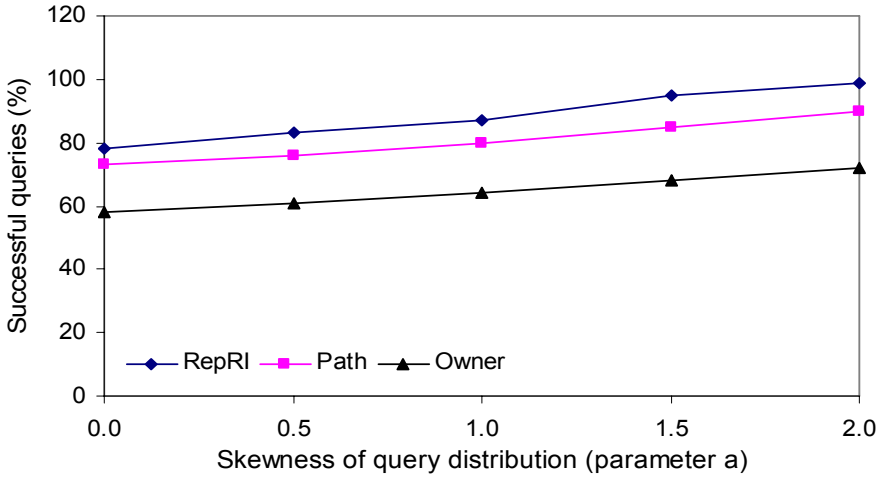


Figure 4.4 Percentage of Successful Queries for Different Values of a of the Zipfian Query Distribution

In Figures 4.3 and 4.4 we show how the skewness of the query distribution affects the performance of the system. In the experiment we set the value of α in the replication utility to 1. As the value of the parameter a (of the zipfian query distribution) increases, all strategies achieve higher standards of performance. For values greater than 1.0, the gain acquired by RepRI replication is bigger, since the number of

replicas for popular documents increases significantly. Favoring replication of popular documents help us improve the overall performance, with respect to the average search size and the percentage of successful queries.

4.2.2. Proactive vs Path Replication for XML Documents

In the next set of experiments we performed a comparison between skeleton replication using RepRIX and path replication. Both techniques were tested under the same query distribution (zipfian a=1.0). They occupied all the available space for replica storage, creating approximately the same number of replicas. In RepRIX replication, RepRIX was used for replicas' replacement and for assisting the routing process, while in path replication we used a first in first out replacement policy.

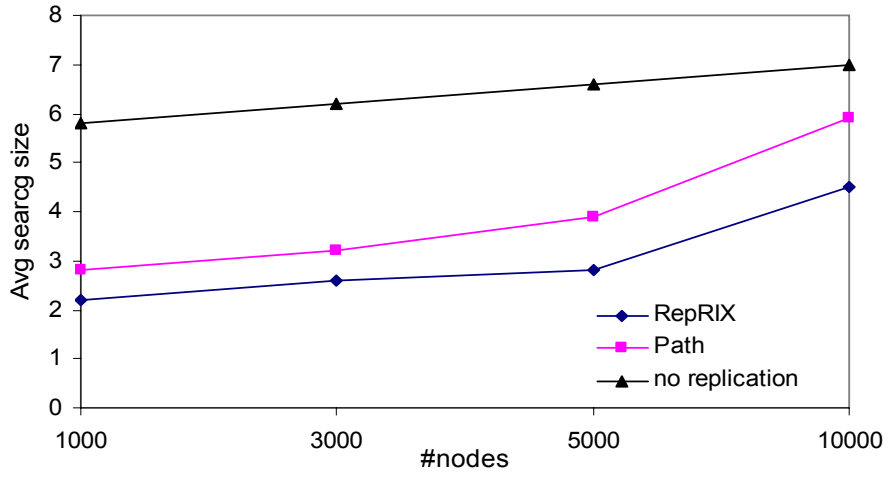


Figure 4.5 Average Search Size for RepRIX, Path and No Replication for Different Sizes of Network

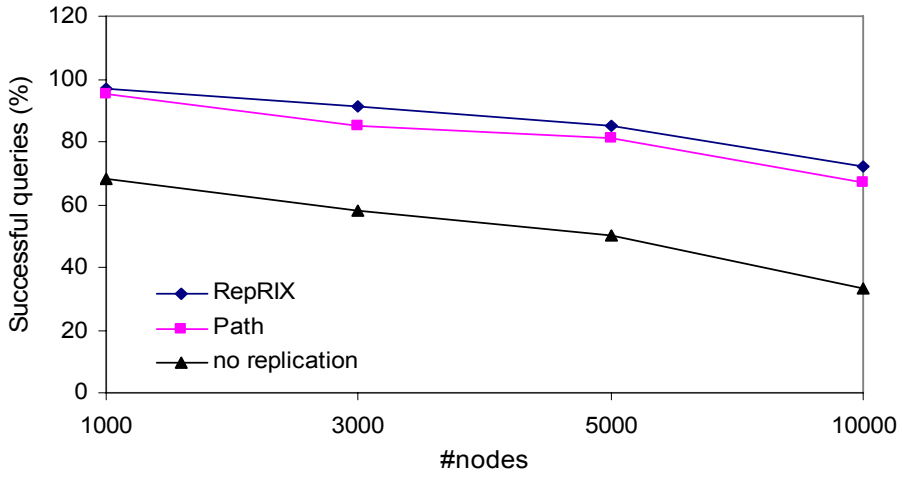


Figure 4.6 Percentage of Successful Queries for RepRIX, Path and No Replication for Different Sizes of Network

In Figures 4.5 and 4.6 we show how the two replication strategies perform for different sizes of networks. The value of α is set to 0.6. Our results show that for all sizes, RepRIX replication outperforms path replication with respect to the average search size and the percentage of successful queries. The reason for this is that with RepRIX we manage to distribute replicas along the network in a more efficient manner than path replication. Replicas are pushed towards the part of the network that actually needs them. Moreover, replacement of replicas based on the replication utility has a positive impact on the performance of the system, since peers manage to keep replicas that are considered *hot* based on our replication criterion. We also show the case where no replication is taken place at the system.

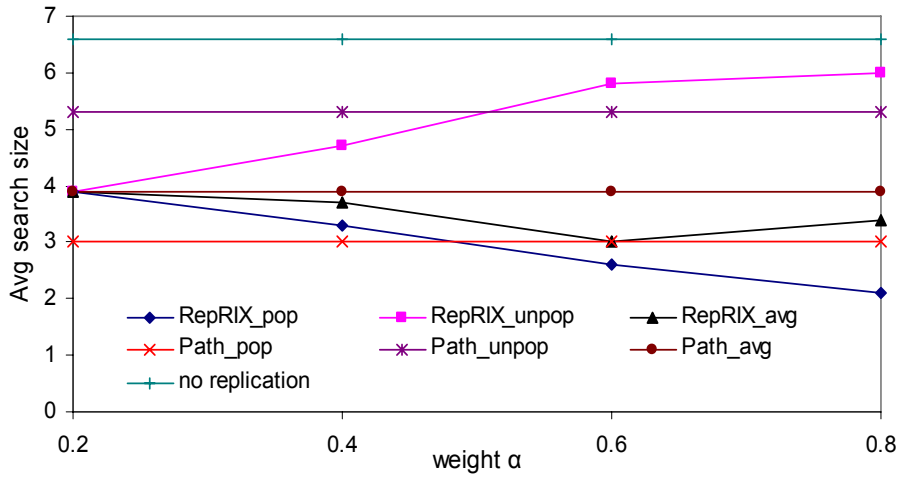


Figure 4.7 Average Search Size for Various Values of the Weight α

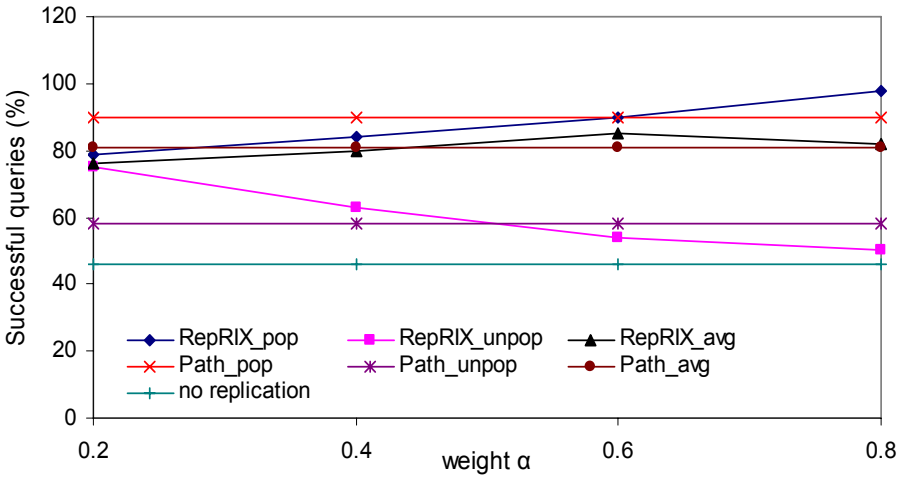


Figure 4.8 Percentage of Successful Queries for Various Values of the Weight α

Next we show the impact of the weight α in the performance of the system. We fixed the size of the network to 5000 nodes and measured the average search size and the percentage of successful queries for the 20% most popular and the 20% most unpopular fragments. As mentioned in chapter 2, the replication utility is defined as: ru = α * popularity_factor + (1-α) * distance_factor. Figures 4.7 and 4.8 show that as the value of α increases, RepRIX shows better performance than path replication for popular fragments. As α increases, the replication utility depends more on the number of requests. Popular fragments have many requests so they are replicated more easily,

resulting in high percentage of query hits and low average search size. However, as α increases, path replication achieves better results for unpopular fragments. For the average popularity, we see that RepRIX replication performs best when α is set 0.6.

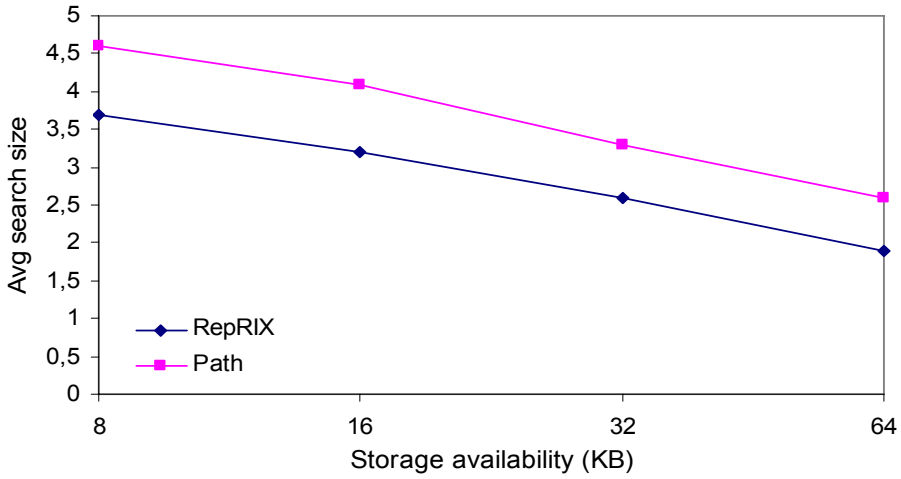


Figure 4.9 Average Search Size for Different Storage Limits

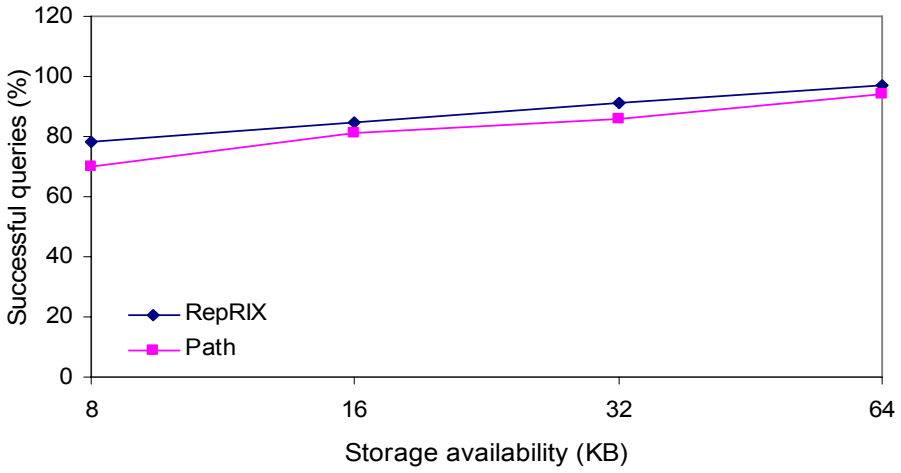


Figure 4.10 Percentage of Successful Queries for Different Storage Limits

Regarding storage availability, RepRIX and path replication show similar behavior as expected. For a fixed size of network and fixed value of α (0.6) for RepRIX, Figures 4.9 and 4.10 show that as the storage capacity of peers increases from 8KB to 64KB, both strategies achieve better performance with respect to the average search size and the query hits. When the storage availability of a peer increases, the number of

replicas stored at the peer also increases. The bigger number of replicas along the network explains the gain we have in performance.

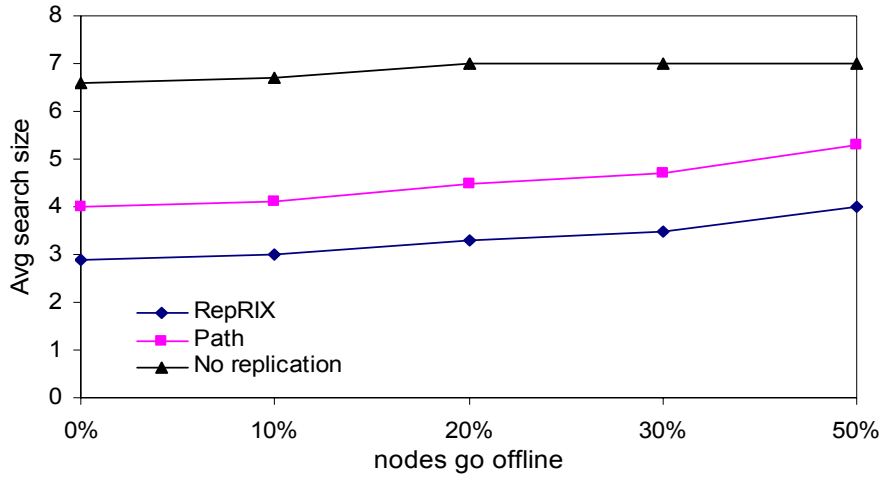


Figure 4.11 Average Search Size for Different Churn Rates

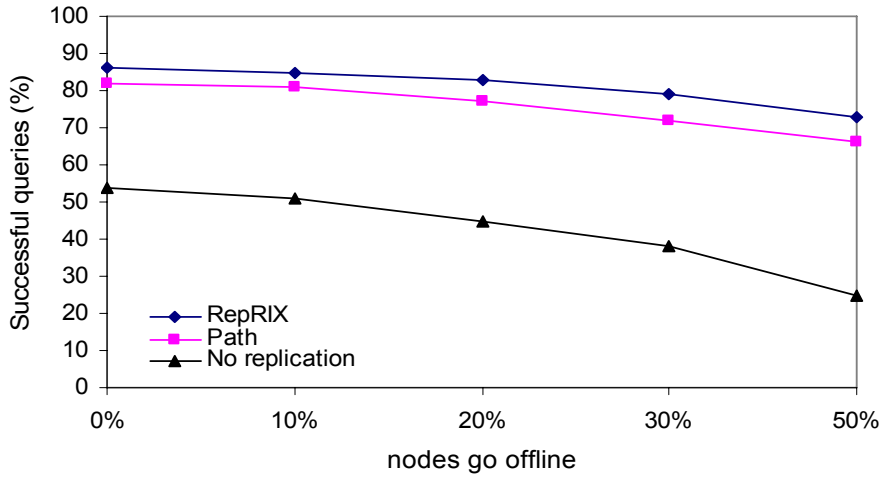


Figure 4.12 Percentage of Successful Queries for Different Churn Rates

In the next experiment, we evaluate the performance of the two techniques under a dynamic network of 5000 peers. The network is dynamic in the sense that we allow a percentage of peers to leave the network and stay offline for some period of time. In Figures 4.11 and 4.12 we see that as the churn rate increases, the performance standards decrease. However, our results show that for all churn rates, our approach

continues to work better compared to path. Figures also show how the system behaves when no replication is taken place.

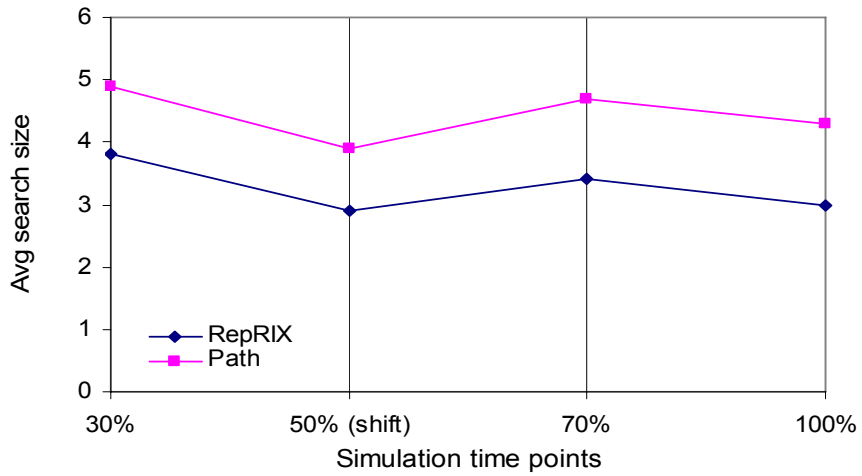


Figure 4.13 Behavior of Average Search Size to Changes in the Query Distribution

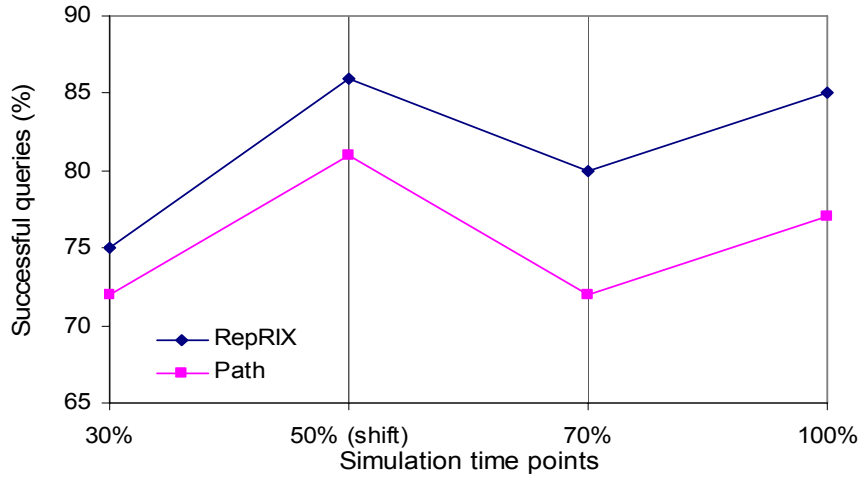


Figure 4.14 Behavior of the Percentage of Successful Queries to Changes in the Query Distribution

In Figures 4.13 and 4.14, we show how RepRIX and path adapt when the pattern of the query load is changed. When we reach at the middle of the simulation, we shift the query distribution, meaning that we change the popularity of fragments. Fragments that were popular till that time become unpopular while unpopular fragments become

popular. As we see, after the change the average search size increases and the percentage of successful queries drops. However, RepRIX replication manages to reach its standards quicker than path. This is due to the replacement policy we use. Changes in the popularity of replicas are mapped to their replication utility. Thus, peers manage to keep those replicas that are considered *hot*.

4.2.3. Fragment Replication vs Whole Document Replication

In the next set of experiments, **w**e show a comparison between fragment replication and whole document replication, with respect to the average search size and the percentage of successful queries. We used path and RepRIX in two different ways. In the first case, we follow our approach creating replicas consisting of fragments of the original documents, while in the second case we replicate whole documents. When using RepRIX for whole document replication, we keep the same statistics as mentioned except that the paths are not matched with fragments but with the original documents.

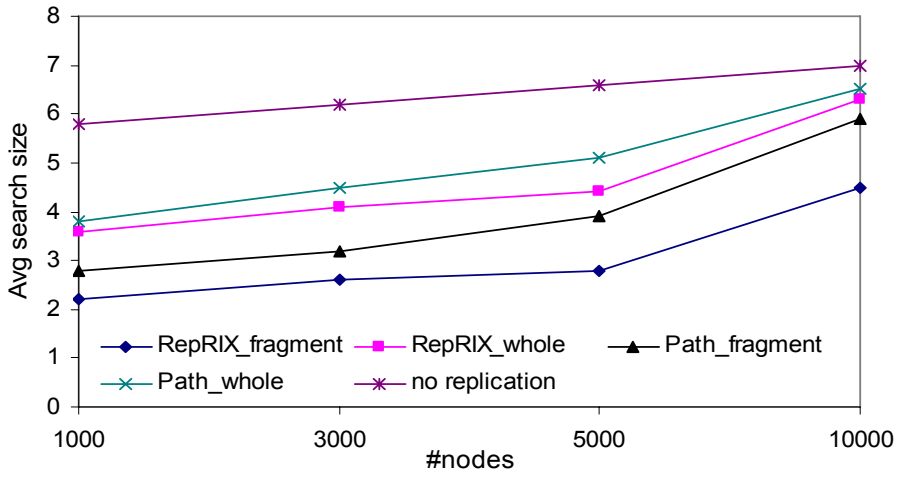


Figure 4.15 Fragment Replication vs Whole Document Replication with Respect to the Average Search Size

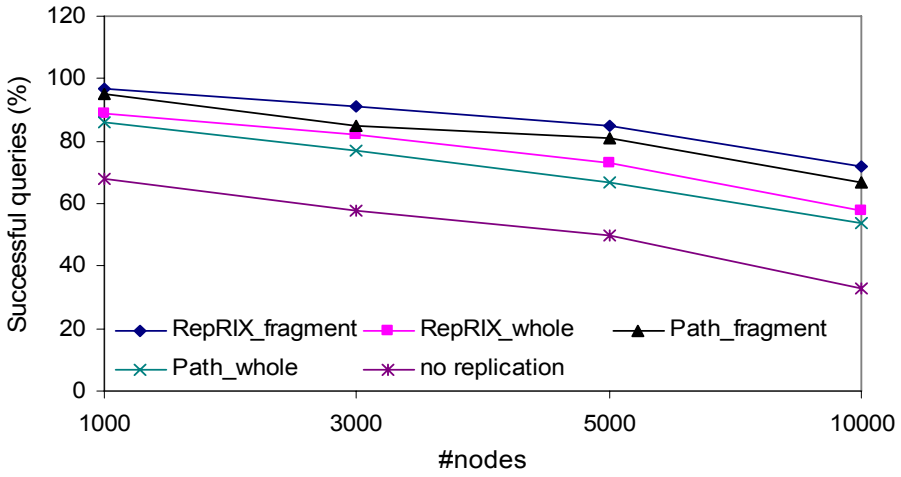


Figure 4.16 Fragment Replication vs Whole Document Replication with Respect to the Percentage of Successful Queries

In the first experiment, we tested the scalability of both fragment and whole document replication. Figures 4.15 and 4.16 show that for all sizes of networks replicating at the fragment level is much more preferable than replicating at the whole document level. The average search size, for RepRIX replication, is reduced by almost 2 hops, while the difference in the percentage of successful queries is close to 20% for a network with 5000 peers. The reason for this behavior is that when peers store fragments of documents, they store much more data that they are interested in than when they store whole documents. Thus, the available storage is used more wisely.

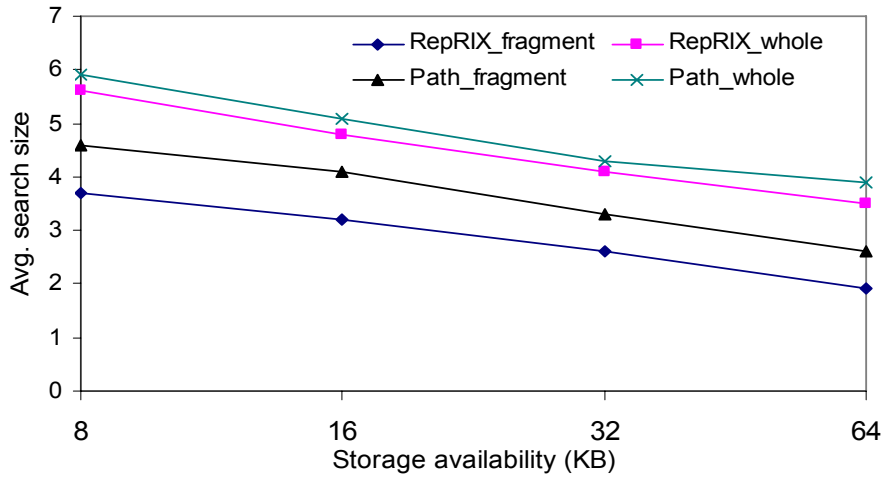


Figure 4.17 Fragment vs Whole Document Replication. Dependence of Average Search Size on Storage Availability

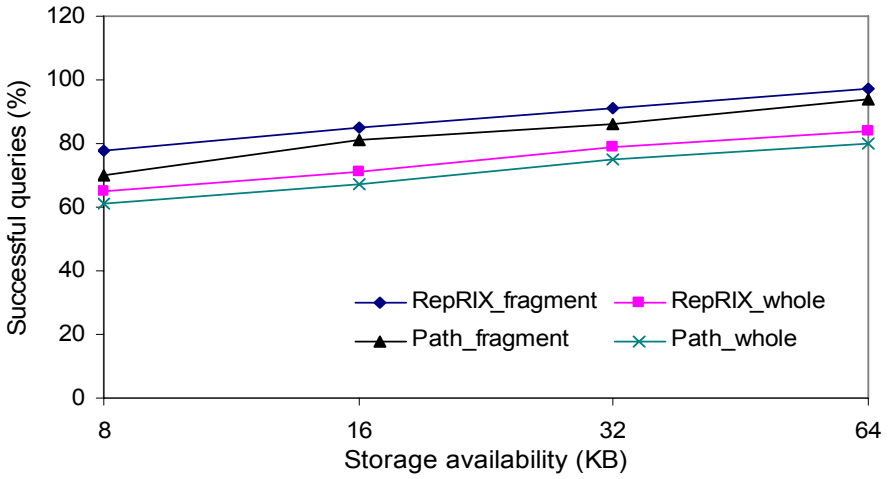


Figure 4.18 Fragment vs Whole Document Replication. Dependence of Query Hits on Storage Availability

Regarding storage availability, Figures 4.17 and 4.18 show that as the storage capacity increases from 8KB to 64KB, both fragment and whole document replication achieve better results but still the first outperforms the second. As the storage availability increases, the number of replicas that a peer can store also increases. This increase is much bigger in the case of fragment replication. Thus, the fact that the total number of replicas along the network is proportional to the storage availability explains the behavior shown in Figures 4.17 and 4.18.

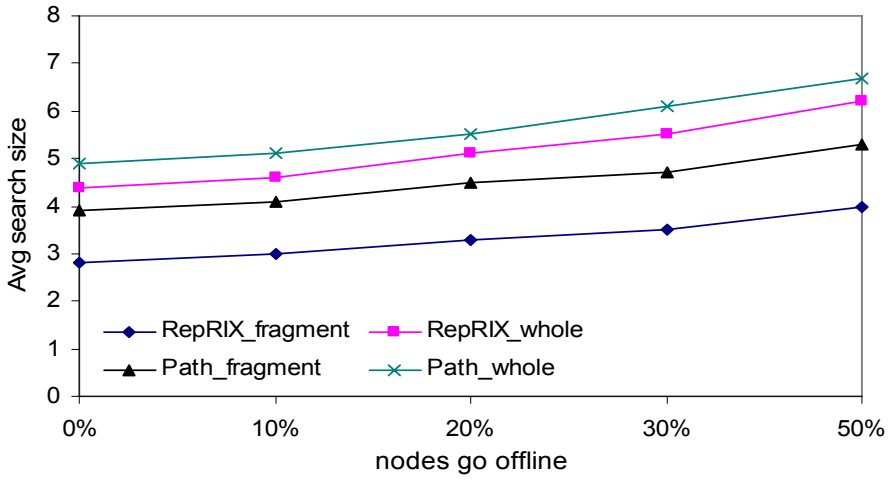


Figure 4.19 Fragment vs Whole Document. Average Search Size for Different Churn Rates

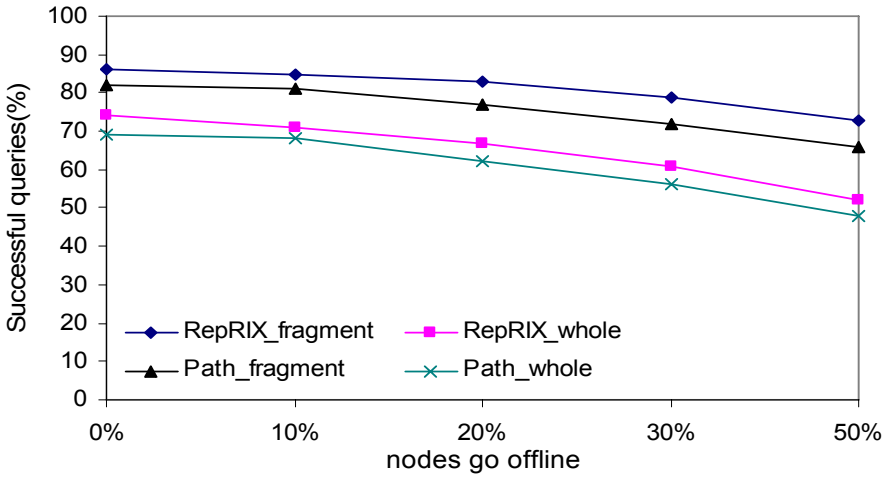


Figure 4.20 Fragment vs Whole Document Replication. Percentage of Successful Queries for Different Churn Rates

Figures 4.19 and 4.20 show how the two approaches behave in a network where peers can join and leave dynamically. As expected, the performance of both approaches drops as the number of peers that can go offline increases. Peers require more hops to locate an item, while the number of queries that are answered is reduced. However, since the churn rate is the same for both cases, the number of replicas maintained at the network by fragment replication remains bigger than that of whole document replication.

4.2.4. Characteristics of RepRIX Replication

In this set of experiments, we perform a comparison between skeleton and subtree replication and we investigate other parameters that affect the performance of RepRIX Replication.

In Figure 4.21 we see a comparison between skeleton and subtree replication with respect to the percentage of successful queries. As we see, skeleton outperforms subtree, since it manages to answer about 6% more queries. Skeleton replication achieves better results than subtree replication due to the presence of external links. External links have a positive impact in the search process, since they assist a peer to locate a fragment that is out of its search range. However, with subtree replication we

can reduce the size of the messages created by our replication technique. In particular, as Figure 4.22 shows, when the average size of fragments is kept small, the messages created by subtree replication can be 25% smaller than those created by skeleton replication

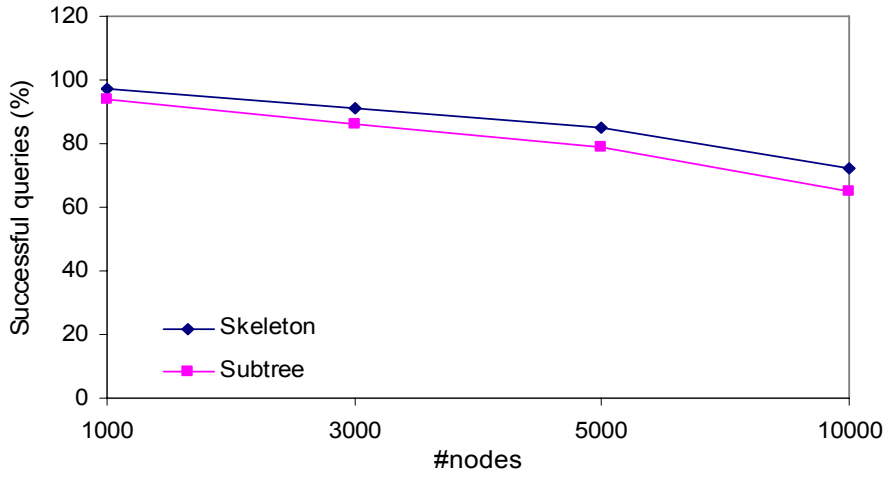


Figure 4.21 Skeleton vs Subtree Replication. Percentage of Successful Queries

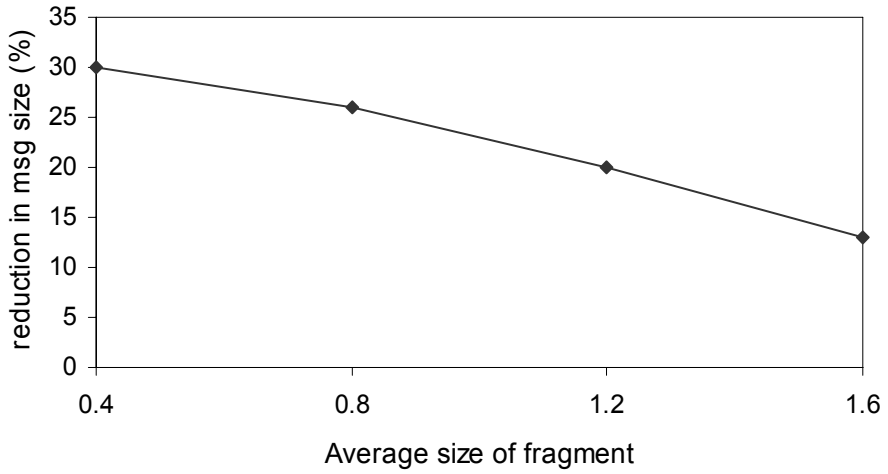


Figure 4.22 Skeleton vs Subtree Replication. Difference in Message Size

Next, we show how the various values for the size of fragments affect the performance of the system. Our results in Figures 4.23 and 4.24 show that when most queries refer to relatively small fragments, both RepRIX and path replication show

better performance, since the percentage of successful queries is higher, while the average search size is also reduced. Taking the two extremes, average file size 0.6KB and 3KB, we see that for RepRIX replication, the difference in performance is 1.7 hops with respect to the average search size and 16% with respect to the percentage of successful queries. As the access frequency of small fragments increases, peers store more replicas. The increase in the number of replicas along the network explains the gain we have in performance.

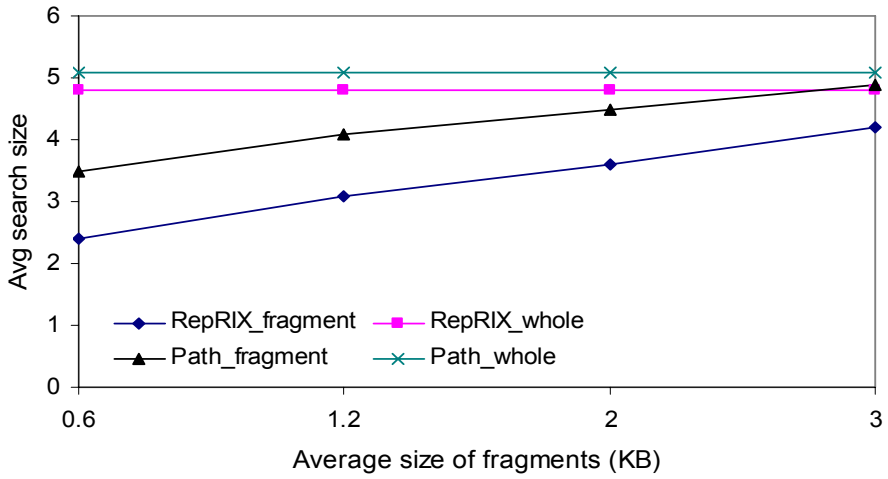


Figure 4.23 Average Search Size for Various Sizes of Fragments

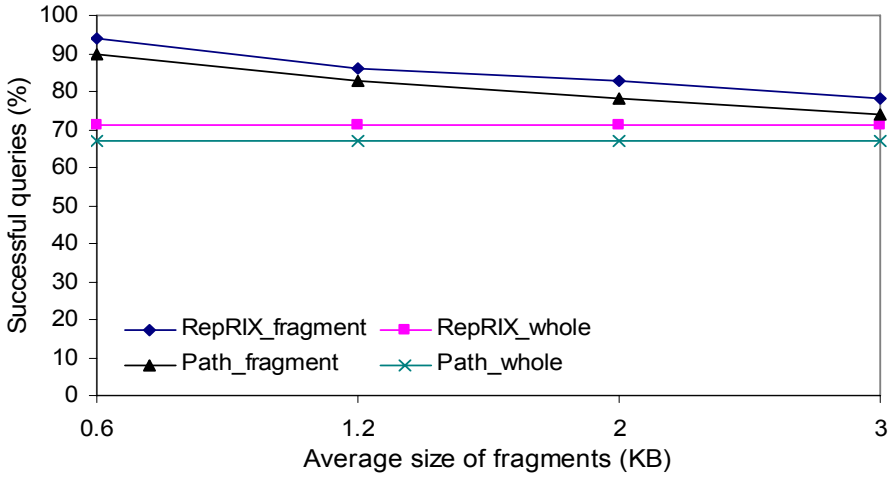


Figure 4.24 Percentage of Successful Queries for Various Sizes of Fragments

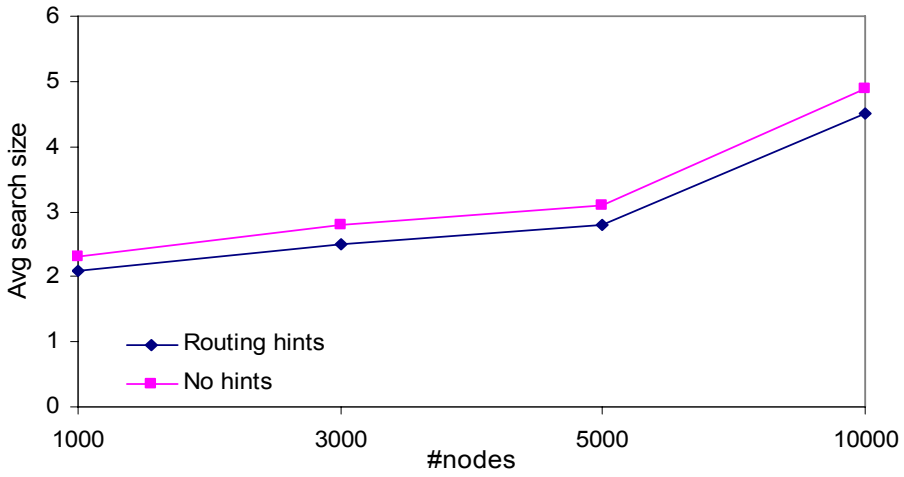


Figure 4.25 Impact of Routing Hints on Average Search Size

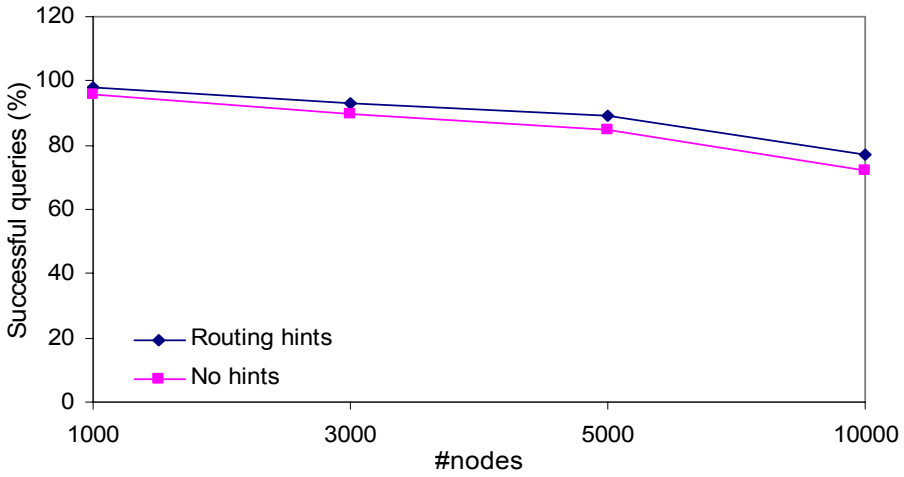


Figure 4.26 Impact of Routing Hints on the Percentage of Successful Queries

Finally we present the impact of routing hints on the performance of RepRIX replication. As we mentioned in chapter 2, RepRIX can also be used to further improve the routing process by maintaining the direction of the provider of a fragment. When a peer forwards a replication message to one of its neighbors, without storing the replica, it can set the corresponding counter to a hint value (instead of resetting it to 0) indicating that the file has been copied along this direction. Thus, the next time the peer receives a query for that file, the peer knows towards which of its neighbors to forward the request, thus reducing the search cost. Figures 4.25 and 4.26

show that the use of routing hints can drop the average search size by almost 0.5 hops and increase the percentage of successful queries by 5%, for networks consisting of 10000 peers.

4.2.5. Cost of RepRIX Replication

In our final set of experiments we measured the cost of RepRIX replication with respect to the number of replication messages and the size of RepRIX.

In Figure 4.27 we show a comparison between RepRIX and path replication with respect to the number of messages they create for placing replicas along the network. As it was expected, path replication creates slightly more messages than RepRIX, since it performs an aggressive replication after each successful query. In RepRIX replication only fragments that satisfy the replication criterion are replicated.

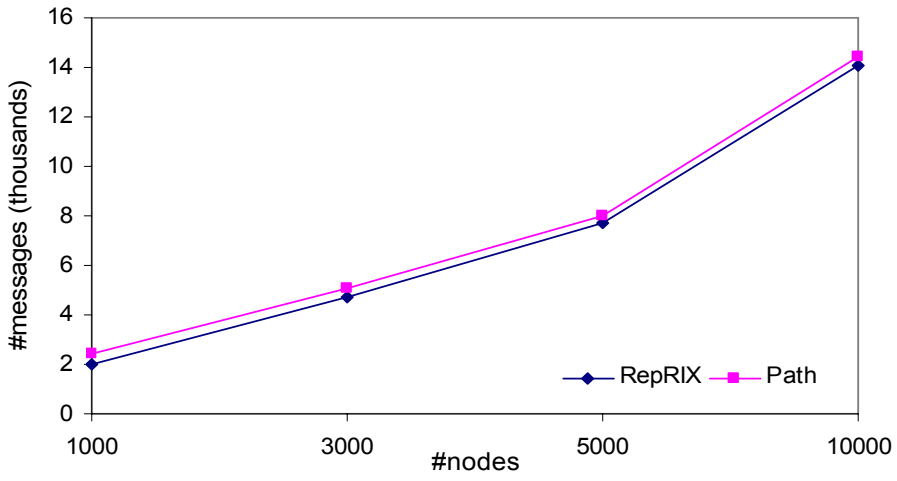


Figure 4.27 Cost of Replication with Respect to the Number of Replication Messages

In Figure 4.28 we show the cost, in storage space, that RepRIX replication pays for maintaining the statistics. The size of RepRIX depends on the peer's degree (number of neighbors). We categorized the peers based on their degree and measured the size of RepRIX for each case. The minimum degree in our network is 4, the average degree is 8 and the maximum degree is 20. The values that are reported refer to the size of RepRIX at the end of a period. We see that in the average case, RepRIX occupies 4KB of the available storage space, which corresponds to around 4 replicas.

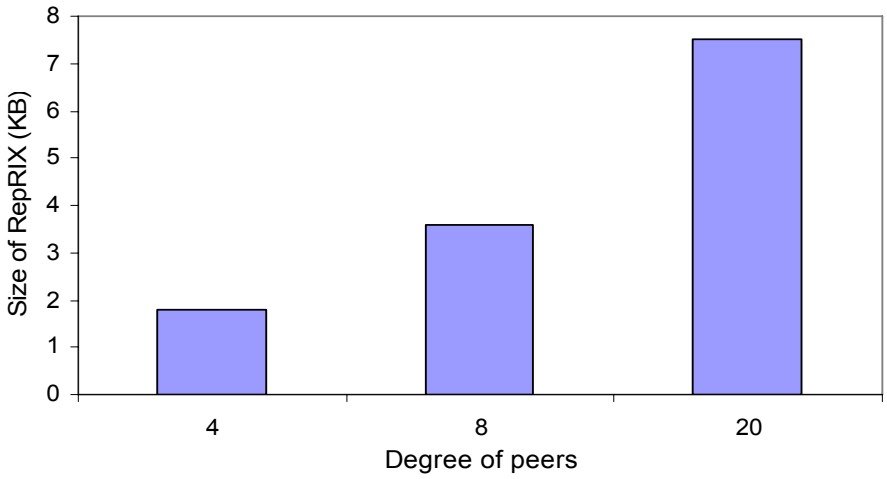


Figure 4.28 Size of RepRIX

CHAPTER 5. RELATED WORK

5.1. Replication in Unstructured p2p Systems

As mentioned before, in unstructured p2p systems there is neither a centralized directory nor any precise control over the network topology or data placement. The network is formed by nodes joining the network following some loose rules. The resultant topology has certain properties, but the placement of data is not based on any knowledge of the topology. The main issue in such systems is locating peers that hold data of interest. One way to improve the search performance is data replication. In this section we present the related work on data replication in unstructured p2p systems.

The authors of [1] consider the general problem of what is the best way to replicate data in unstructured p2p systems given that the total amount of storage in the network is fixed. Two natural ways to perform replication is uniform and proportional replication. With *uniform replication*, the same number of copies is created for all items, while with *proportional replication*, the number of copies created for each item is proportional to the item's popularity, i.e., its query rate. It is shown that both replication strategies have the same expected search size for successful queries. However, they differ in other aspects. Uniform replication distributes the load evenly to all copies, whereas in the case of proportional replication, copies receive load proportional to their query rates. Proportional replication also makes popular items easier to find, at the expense of making less popular ones harder to find. Thus, with proportional replication, a much higher limit (TTL value) is required for locating them. On the other hand, uniform replication minimizes this limit. It is also shown

that in terms of search size, uniform and proportional replication lie at two extremes where the ratio of allocation of two items is between 1 and the ratio of their query rates. All replication strategies that lie between these extremes yield better search sizes for successful queries with square-root replication achieving the optimal such size. *Square-root replication* allocates replicas to items proportional to the square root of their query rate. The gain attained with square root replication grows with the query skew.

In [2], an evaluation of two easily implementable replication strategies, namely owner and path replication, is provided, under a realistic setting. Under *owner replication*, when a search is successful, the object is stored at the requester peer only. *Path replication* is implemented by storing the object at all peers on the path from the requester peer to the provider peer. The evaluation of the two strategies is done in conjunction with a k random walkers search strategy. In this case, the numbers of peers between the requester peer and the provider peer is $1/k$ of the total number of peers visited. Since path replication tends to replicate objects to peers that are topologically along the same path, the authors also consider a third replication strategy called random replication. *Random replication* counts the number of peers on the path between the requester and the provider, say p, then, randomly picks p of the peers that the k walkers visited and stores the object at them. The evaluation is done on a random graph network topology. The replica allocation achieved by both path and random replication are quite close to the square-root. They also reduce the overall traffic by a factor of three to four mainly by reducing the search size. Random replication improves over path replication for the cost of a more involved implementation.

Path replication distributes query load for popular items across multiple peers, reduces latency and alleviates hot spots. However, path replication tends to replicate files to peers that are topologically along the same path. Moreover, the number of replicas created can become very large, which eventually may be more than necessary to achieve the required search performance. In p2p networks with random topologies, where peers are randomly chosen to initiate queries, we would like to avoid the topological impact of path replication. Our approach manages to overcome this

problem, since the use of Replication Routing Indexes allows us to distribute replicas in a more efficient manner, based on the statistics we keep.

In most p2p networks, the number of neighbors (degree) of each peer follows the power law; there exist a few high degree peers and a large number of low-degree peers. Therefore, a huge number of requests can go through these few high-degree peers, and the storage loads due to reading and writing requested data is concentrated on them. The authors of [7] consider the problem of how to mitigate the load concentration on the high-degree peers over a p2p network without deteriorating the search performance too much. They introduce the replication ratio, which is the ratio of the created replicas to all the intermediate peers on the path for each requested data. The replication ratio is determined in advance. Two replication methods are proposed, called Path Random Replication and Path Adaptive Replication, which make replicas on some chosen peers, through which the data passes. In *Path Random Replication*, each intermediate peer randomly determines whether or not the replica is created and placed there based on the probability of the pre-determined replication ratio. In *Path Adaptive Replication*, the procedure in the decision to make a replica on a peer depends on how much storage is still available on it as well as the predetermined replication ratio.

Path Random Replication and Path Adaptive Replication suffer in the cases where peers are highly different in their degrees. Constant replication probability may still cause much load imbalance, because high-degree peers are frequently located in the data transmission path. Path Random Replication and Path Adaptive Replication manage to reduce the large number of replicas created by path replication but in comparison with our approach, they don’t avoid the topological impact of path replication.

A dynamic data replication algorithm that is used in distributed systems with tree networks is the *Adaptive Data Replication* algorithm (ADR) [8]. ADR is a distributed algorithm, in the sense that each peer makes decisions to locally change the replication scheme, evaluating statistics collected locally. ADR changes the replication scheme of an object dynamically, as the read-write pattern of the object

changes in the network. The changes in the read-write pattern may not be known a priori. The changes at the replication scheme result in the decrease of its communication cost. The communication cost of a replication scheme is the average number of messages required for a read or a write of the object. The replication scheme expands as the read activity increases, and it contracts as the write activity increases. The ADR algorithm works in a read-one-write-all manner. A read of the object is performed from the closest replica in the network, while a write updates all the replicas and is propagated along the edges of a subtree that contains the writer and the peers of the replication scheme. In the ADR algorithm the initial replication scheme consists of a connected set of peers and at any time, the peers of the replication scheme, denoted *R*, are connected. A peer is considered to be an *R-neighbor* if it belongs to *R* but it has a neighbor that does not belong to *R*. An *R-fringe* peer is defined to be a leaf of the subgraph induced by *R*. The need for changes at the replication scheme is determined using three tests, namely, the expansion test, which is executed by each peer that is an R-neighbor, the contraction test, which is executed by each peer that is an R-fringe and the switch test. A peer can be both an R-neighbor and R-fringe. In this case, it first executes the expansion test and if it fails, then it executes the contraction test. A peer in *R* that does not have any neighbors that are also in *R* executes first the expansion test and if it fails, then it executes the switch test. Each peer *i* that is an R-neighbor performs the expansion test. For each neighbor *j* that is not in *R*, let *x* denote the number of reads that *i* received from *j* during the last time period and *y* the total number of writes that *i* received in the last time period from *i* itself, or from a neighbor other than *j*. If $x>y$, then *i* sends to *j* a copy of the object with an indication to save the copy in its local database. Thus *j* joins *R*. Except for *i* and *j*, no other peers are informed of the expansion of *R*. The expansion test is performed by comparing the counters (one for the reads and the other for the writes). The counters are initialized to zero at the end of each time period and incremented during the following time period. The expansion test succeeds if the if condition is satisfied for at least one neighbor. The expansion test fails if it does not succeed. The contraction test is performed by an R-fridge peer. A peer *i* is called an R-fridge peer if it is in *R* and has exactly one neighbor *j* that is in *R*. Let *x* denote the number of writes that *i* received from *j* during the last time period and *y* the number of reads that *i* received in the last time period (the read requests received by *i* are made by *i* itself or

received from a neighbor of *i* different from *j*). If *x>y*, then *i* requests permission from *j* to exit *R*, that is, to discard its copy. Peer *i* does not exit unconditionally, since *i* and *j* may be the only peers of the current replication scheme, and they may both announce their exit to each other, leaving an empty replication scheme. Therefore, if the contraction test succeeds, then *i* keeps its replica until it receives the next message from *j*. If this message is *j*'s request to leave *R*, then only one leaves *R*. Except for *i* and *j*, no other processor is informed of the contraction. When a peer *i* constitutes the only peer in the replication scheme, then *i* is an R-neighbor, thus it executes the expansion test. If the expansion test fails, then *i* executes the switch test. For each neighbor *j*, let *x* denote the number of requests received by *i* from *j* during the last time period and y the number of all other requests received by *i* during the last time period. If *x>y*, then *i* sends a copy of the object to *j* with an indication that *j* becomes the new singleton peer in the replication scheme, and *i* discards its own copy. When the if condition of the contraction or switch test is satisfied, then the test succeeds. Otherwise, it fails.

ADR is similar to our approach in the sense that replication decisions are made periodically based on statistics. However, in ADR, the only criterion for replicating a file is its popularity. In our approach we also take into account the cost, measured in number of hops, for locating it.

Although data replication significantly improves the performance of the system, it raises the problem of replicas' consistency, in the case of updates. The most popular algorithms for update propagation are the *epidemic* algorithms, presented in [9]. Typical examples of epidemic algorithms are *direct mail*, *anti-entropy* and *rumor mongering*. In direct mail, when an update occurs, it is immediately mailed from its originating peer to all other peers. The main advantage of this algorithm is that updates are propagated very quickly. In anti-entropy, periodically, each peer selects randomly another peer and resolves any differences between them, by exchanging content. The anti-entropy strategy is reliable, but quite slow. In rumor mongering, when a peer receives a new update, it periodically selects another peer and checks if this peer has seen the update, in order to send it to it. A peer stops sending the update to other peers, when many other peers have seen it.

The update strategy that is proposed in [10] is based on a hybrid push/pull rumor spreading algorithm and provides a fully decentralized, efficient and robust communication scheme which offers probabilistic guarantees rather than ensuring strict consistency. During the first phase of the algorithm, called the *push phase*, the peer where the update occurred, pushes the update to a set of peers that have a replica of the updated object. These peers, in turn propagate the update to another set of peers and this procedure continues in a manner similar to flooding. The second phase is the *pull phase*, where peers coming online or peers that received no update for some time, contact other peers and ask for newly updated objects.

5.2. XML Replication and Fragmentation

The problem of replicating XML data or indexes in p2p systems has not received much attention yet. An important issue that arises is the granularity of replication and distribution for XML. In this section, we present related work on replication in p2p systems where peers store XML documents. Similar approaches to ours for document fragmentation are also described.

In [11], an approach for replicating XML documents, in unstructured p2p systems, is presented. The authors consider a new class of documents called *dynamic XML documents*. Dynamic XML documents are XML documents that contain materialized XML data that are part of the document and intentional data that can be produced by service calls. Since dynamic documents may contain calls to services on other peers, some form of distribution is inherently part of the model. Fragments of the documents, including services, can be replicated or distributed along the network. External edges are added to the replicated documents to point to peers that store other parts of the documents to allow for a higher form of distribution. The approach is similar to ours. The focus there is making decisions on whether to materialize the result of a call or not based on a cost model. The cost model intends to reflect the *observable performance* of a given peer: the costs and performance metrics perceived by this particular peer. This observable performance is influenced by some objective parameters (e.g. size of data transfers, from/to a given peer, incurred by the

execution), and some subjective parameters (e.g. the relative impact of communication, space, and computation costs on the overall cost afforded by the peer.)

The authors in [12] address the problem of replicating XML indexes for load balancing. A distributed catalog framework based on a structured p2p system is proposed. The system uses Chord [13] as the overlay network. The distributed catalog stores sets of key-summaries information for all the peers. The *keys* for XML data are either element or attribute names. The summaries that correspond to each key are either structural or value summaries. The *structural summaries* of a key are all possible paths leading to that key. The type of the *value summary* depends on the domain of the key, i.e. histograms are used for arithmetic keys. For each new peer that enters the system, each key-summary pair is inserted in the system by the Chord protocol according to the hash values of the keys. Two methods are proposed for splitting the load among peers, namely the split-replicate and the split-toss methods that split catalog information among peers. A peer increases the level of a popular index key, where level is the number of XML-tree path steps contained in the key (initially set to 1), and either replicates the new keys at the corresponding peers (according to the hash function) in the Chord ring (split-replicate), or only sends them there and then discards them (split-toss).

The work reported in [14] addresses the XML allocation problem, that is given a collection of XML documents how to fragment them and allocate the resulting fragments among the nodes of a distributed system to improve performance. There, a global conceptual schema is used by a simplified structure called RepositoryGuide, which is a tree-structured index. The system supports XML-path and tree pattern queries. The fragmentation scheme decomposes the RepositoryGuide into a disjoint and complete set of tree-structured fragments that preserve data semantics. A sublanguage of XPath is used for data fragmentation that supports vertical fragmentation, which is solely based on the selection of node types through path properties. The allocation phase consists of three steps: determining which fragments to allocate at which system nodes, placing schema structures at local nodes and suitable instances of fragments at each node. For the first step, existing methods from

distributed databases are used. For the second step, the RepositoryGuide is fully distributed among the nodes. Finally, for the third step, the global context of each fragment is kept by storing the data path from the global root node to the root of the local fragment. To this end, three indexes are used: a path index that encodes the global context of local fragments, a term index that allows processing of queries that include conditions on terms and an address index that stores the physical addresses of the fragments. Space efficient path indexes are constructed with the use of a path identification scheme. Because of their small size, path indexes are replicated at all system nodes, while term and address indexes are distributed among them.

The problem we address in our work is different in that we do not consider partition and allocation but dynamic replication of fragments. Moreover, in our systmem peers have no global knowledge about the location of other fragments.

In [15] an approach for vectorizing and querying large XML repositories is presented. The idea is based on the decomposition of an XML document into a set of vectors that contain the data values and a compressed skeleton that describes the structure. In order to query this representation and produce results in the same vectorized format, they consider a practical fragment of XQuery and introduce the notion of query graphs and a novel graph reduction algorithm that allows to leverage relational optimization techniques as well as to reduce the unnecessary loading of data vectors and decompression of skeletons. This is similar to our approach of representing replicas. Again, in this work, we address a different problem, that of dynamically creating replicated fragments.

5.3. XML Processing in Structured and Unstructured p2p Systems

The use of XML as the format for data representation introduces additional problems in p2p systems. In unstructured p2p systems, research efforts focus on building space efficient routing indexes for XML documents. Most approaches build path indexes with the use of aggregation and suitable encoding schemes for the paths. In structured p2p systems, recent research focuses on exploiting the content of documents for determining the keys. In particular, a vector describing each document is extracted

and used as the key to map the documents to the virtual multi-dimensional space of the network. In this section we describe proposed techniques for processing XML documents in both structured and unstructured p2p systems. This research is complementary to ours since it does not address replication but deals only with query processing techniques.

Processing of containment queries in p2p systems is presented in [16]. Containment queries exploit the structure of XML data (i.e. book contains author contains name = “John Smith”). XML elements and text words are treated uniformly as index keys. Local indexes at each peer consist of inverted lists, which map keywords to XML documents stored at the peer. In addition to its local inverted lists, each peer also maintains routing indexes, called peer inverted indexes that map keywords to the identifiers of remote peers. A query is forwarded to remote peers by using the peer inverted index and set operations are used to minimize the number of relevant destinations. Indexes are built when a peer joins the system by exchanging information with other peers. These indexes are smaller than local indexes, since a peer only exchanges a small subset of its keywords, such as words that are often found in queries or that are representative of its local data. The result is a p2p system in which each peer has a summary of important data of all other peers. Horizons are used to limit the number of peers for which a peer has summarized information. A peer maps keywords outside of its horizon to peers on the boundary of the horizon that are closer to them.

In [17], each peer maintains a local index, summarizing its local content and one or more merged indexes summarizing the contents of its neighbors. The peers form hierarchies in which each peer stores summarized data for the peers belonging to its subtree. The roots are interconnected and store additional summaries for all other roots. Each peer that receives a query first checks its local index for any matches. Then, if it is an internal peer, it checks its merged index and if there is a match it forwards the query to its subtree. Furthermore, it sends the query to its parent or if it is a root peer to the other matching roots. The indexes used are based on Bloom filters that are compact data structures for the representation of a set of elements. To support the evaluation of regular XPath expressions, multi-level Bloom filters are introduced

that preserve hierarchical relationships between the inserted elements. These relationships are preserved by inserting the elements of the XML tree to a different level of the filter according to their depth in the tree (Breadth Bloom filters), or by using paths of different lengths as keys and inserting them to the corresponding level of the filter according to their length (Depth Bloom filters).

A similar fragmentation model with ours is described in [18] and [19]. However, the fragments that they create do not have any links to the original document and they focus more on query processing rather than replication. The authors of [18] deal with the problem of parallel query processing. They develop their idea for the evaluation of boolean XPath queries over a tree that is fragmented, both horizontally and vertically over a number of sites. The key idea is to send the whole query to each site which partially evaluates, in parallel, the query and sends the results as compact boolean functions to a coordinator which combines these to obtain the result.

XP2P [19] also extends Chord to support XML data. The system assumes that each peer stores a set of XML fragments (subtrees of XML data). In addition, each peer stores the local content of the user's fragments and their related path expressions that are the lists of each fragment's child fragments (path expressions stored as PCDATA within subtags in the fragment) and their super fragment (a path expression of the fragment which is the ancestor of the current fragment). These expressions are hashed into the Chord virtual space. The hashing technique used is different from that used in Chord. In particular, a fingerprinting technique is proposed. The produced fingerprints are shorter than the hash keys used in Chord and support a concatenation property that allows the computation of the tokens associated with path expressions to proceed incrementally. Partial and full match lookups are supported, where in the first case, a match to a fragment is returned without unfolding its child fragments, while in the latter case, all the sub tags of the fragment are unfolded and the corresponding child fragments are retrieved. The queries are fingerprinted as well and when the fingerprint of a query (either in full or partial lookup) matches the fingerprint of a data fragment, the results are located by the lookup functionality of Chord. If the system cannot find a match, for instance if some peers are temporarily unavailable, additional techniques

based on gradually pruning the query path are deployed to provide the user with at least a partial match.

CHAPTER 6. CONCLUSIONS

6.1. Summary

Peer-to-peer (p2p) systems have attracted considerable attention as a means of sharing content among large and dynamic communities of nodes. A central issue in p2p systems is locating the nodes that hold data of interest. There have been various proposals towards building overlays to support efficient content location. Such proposals vary from building rigid topologies and placing data on specific nodes in the overlay to unstructured networks with no correlation between the node content and its position in the overlay. In all types of overlays, content replication results in reducing the latency of lookups. Motivated by the fact that XML is increasingly being used in data intensive applications, in this work, we studied replication in unstructured p2p systems where participating nodes share content stored in XML. We considered XML replication for both passive and proactive protocols.

XML documents have a hierarchical structure and thus, different fragments of an XML document can have different access frequencies. We showed that replicating items at the *fragment level* is preferable to replicating whole documents. For proactive replication, we introduced a new data structure that we call replication routing index. For a peer *p*, a *Replication Routing Index (RepRI)* has one entry for each file that *p* has processed queries for. Each entry keeps statistics about the requests that *p* has received for the specific file through its adjacent edges. Our replication strategy uses these indexes to decide whether to maintain a copy locally or forward it along a path. A Replication Routing Index for XML, termed RepRIX, maintains statistics for

fragments. RepRIX allows us to fine-tune the unit of replication, so that fragments of the same document can have different numbers of replicas. Further, it allows us to push fragments closer to their requesters. RepRIX entries are also used as hints during lookup to direct nodes towards paths that most probably hold replicas of the requested items.

6.2. Future Work

One way to improve the efficiency of RepRIX replication in realistic p2p environments would be the ability to dynamically tune the value of the weight α. In order to exploit the ability to enhance the system performance for popular fragments each peer *p* could use the statistics maintained at RepRIX(p) to estimate the query distribution. Based on these estimations peers would be able to adjust the value of α to the actual requirements of the network.

As mentioned before, the duration of the period of the replication procedure depends on the query workload. A large value leads to making more informed decisions based on sufficiently large samples of requests and ignores popularity fluctuations that may be caused by random variations in the query workload. Furthermore, it reduces the associated network overhead. On the other hand, the system adapts to workload changes less promptly. An efficient way to dynamically adjust the duration of the period, based on the changes of the average replication utility, would improve the overall performance of RepRI replication.

Additional work could be done in the way path queries are processed. Instead of searching for the peer that holds all the requested data, partial evaluation could be executed by peers that hold some part of it.

REFERENCES

[1] E. Cohen, S. Shenker. Replication strategies in unstructured peer-to-peer networks. SIGCOMM, pp. 177-190, ACM, 2002.

[2] Q. Lv, P. Cao, E. Cohen, K. Li, S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. SIGMETRICS, pp. 258-259, ACM, 2002.

[3] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition).

[4] D. Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001.

[5] Peersim website: http://peersim.sourceforge.net

[6] ToXgene website: http://www.cs.toronto.edu/tox/toxgene/

[7] H. Yamamoto, D. Maruta and Y. Oie. Replication Methods for Load Balancing on Distributed Storages in p2p Networks. SAINT, pp. 264-271, IEEE Computer Society, 2005.

[8] O. Wolfson, S. Jajodia, Y. Huang. An Adaptive Data Replication Algorithm. ACM Transactions on Database Systems, pp. 255-314, 1997.

[9] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry. Epidemic Algorithms For Replicated Database Maintenance. PODC, pp. 1-12, 1987.

[10] A. Datta, M. Hauswirth, K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. ICDCS, p. 76, IEEE Computer Society, 2003.

[11] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. SIGMOD Conference, pp. 527-538, ACM, 2003.

[12] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating Data Sources in Large Distributed Systems. VLDB, pp. 874-885, 2003.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. CHORD: A Scalable Peer-to-peer Lookup Service for Internet Applications. SIGCOMM, pp. 149-160, 2001.

[14] J. M. Bremer and M. Gertz. On Distributing XML Repositories. WebDB, pp. 73-78, 2003.

[15] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, S. D. Viglas. Vectorizing and Querying Large XML Repositories. ICDE, pp. 261-272, IEEE Computer Society, 2005.

[16] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Processing Queries in a Large Peer-to-Peer System. CAiSE, Lecture Notes in Computer Science, Vol. 2681, pp. 273-288, Springer, 2003.

[17] G. Koloniari and E. Pitoura. Content-Based Routing of Path Queries in Peer-to-Peer Systems. EDBT, Lecture Notes in Computer Science, Vol. 2992, pp. 29-47, Springer, 2004.

[18] P. Buneman, G. Cong, W. Fan, A. Kementsietsidis. Using Partial Evaluation in Distributed Query Evaluation. VLDB, pp. 211-222, ACM, 2006.

[19] A. Bonifati, U. Matrangolo, A. Cuzzocrea, M. Jain. XPath lookup queries in P2P networks. WIDM, pp. 48-55, ACM, 2004.

AUTHOR'S PUBLICATIONS

P. Skyvalidas, E. Pitoura, V. Dimakopoulos. Replication Routing Indexes for XML Documents in P2P Systems. Preprint submitted to The Seventh IEEE International Conference on Peer-to-Peer Computing, 2007.

BRIEF CV

Panagiotis Skyvalidas was born in Larisa, in 1981. He received his BSc degree in Computer Science, in 2004, from the University of Crete. Since October 2004, he is an MSc student in the department of Computer Science, in the University of Ioannina and since December 2005 is a member of the Distributed Data Management Laboratory. His research interests include Peer-to-Peer Systems, Databases and Web development.