1

DATA REPLICATION AND UPDATE PROPAGATION IN XML P2P DATA MANAGEMENT SYSTEMS

A dissertation submitted to the Department of Computer Science,
Faculty of Science at the University of Cape Town
in fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in
Computer Science

— Marlon Paulse —

Supervisor
Dr S. Berman

UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

August, 2008

Abstract

XML P2P data management systems are P2P systems that use XML as the underlying data format shared between peers in the network. These systems aim to bring the benefits of XML and P2P systems to the distributed data management field. However, P2P systems are known for their lack of central control and high degree of autonomy. Peers may leave the network at any time at will, increasing the risk of data loss. Despite this, most research in XML P2P systems focus on novel and efficient XML indexing and retrieval techniques. Mechanisms for ensuring data availability in XML P2P systems has received comparatively little attention. This project attempts to address this issue.

We design an XML P2P data management framework to improve data availability. This framework includes mechanisms for wide-spread data replication, replica location and update propagation. It allows XML documents to be broken down into fragments. By doing so, we aim to reduce the cost of replicating data by distributing smaller XML fragments throughout the network rather than entire documents.

To tackle the data replication problem, we propose a suite of selection and placement algorithms that may be interchanged to form a particular replication strategy. To support the placement of replicas anywhere in the network, we use a Fragment Location Catalogue, a global index that maintains the locations of replicas. We also propose a lazy update propagation algorithm to propagate updates to replicas.

Experiments show that the data replication algorithms improve data availability in our experimental network environment. We also find that breaking XML documents into smaller pieces and replicating those instead of whole XML documents considerably reduces the replication cost, but at the price of some loss in data availability. For the update propagation tests, we find that the probability that queries return up-to-date results increases, but improvements to the algorithm are necessary to handle environments with high update rates.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Sonia Berman, for her patience, understanding and guidance throughout the project, and for allowing me the freedom to carry out my ideas. I also thank my fellow students in the Advanced Information Management research group for their friendship and making the lab an enjoyable working environment. Finally, thank you to my parents for their continuous love and support during my years of study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, research has been conducted to bring some of the benefits of peer-to-peer (P2P) systems to the distributed data management field [5, 28, 35, 36, 37, 41, 51, 52, 59]. A P2P system is a distributed computing system consisting of large populations of interconnected nodes (or peers) that cooperate to directly share resources such as files, storage space, network bandwidth or processing power. Unlike traditional distributed data management systems [12], P2P systems are generally known for their lack of central control, low administration and deployment overhead, high degree of autonomy and heterogeneity, and their ability to self-adapt and function in unstable network environments [3].

Peers in P2P systems use various data formats and schemas to describe their data. In order to share data with others, a common format that is understood by all peers is needed. XML's [93] ability to represent any hierarchical or semistructured data, makes it an interesting option for such data sharing in P2P environments. XML allows autonomous peers to interact despite the differences in their data formats and schemas. Furthermore, the use of XML enables users to pose queries using expressive query languages such as XPath [98] and XQuery [99], rather than employing simple keyword-based searches.

Motivated by the benefits of XML, many researchers have begun investigating the use of XML data in P2P data management systems [9, 42, 64, 65, 68]. Our survey of existing XML P2P data management systems revealed that most of this research focuses on the efficient indexing and retrieval of XML data in P2P networks. The problem of ensuring the availability of data when peers depart the network has received comparatively little attention. Koloniari and Pitoura make a similar observation in [43].

Ensuring data availability is an important problem in P2P systems. Peers may leave the network any time at will, resulting in data loss. Measurement traces of the Overnet P2P system by Bhagwan et al [6] showed that over 20% of the total peer population entered and departed the network every day. A similar trace by Saroui et al [63] on Napster [11] and Gnutella [75] showed that approximately 50% of the total peer population never remained online for more than one hour. While these traces are for P2P filesharing systems where users are more interested in sharing files for selfish purposes rather than being part of a community of integrated data stores, it still gives a rough idea of how high the rate of peer departures from a P2P network may become.

In this project, we investigate the data availability problem in XML P2P data management systems. In particular, we study algorithms for replicating XML data to reduce the data loss incurred when peers depart the network. We propose a P2P framework in which XML documents are fragmented into smaller pieces to reduce the transfer cost when replicating data. Finally, we look at an update propagation algorithm for transmitting updates to replicas when the original data items are updated.

1.1 Overview of Approach

In order to ensure data availability, we believe that a P2P system should fulfill three requirements:

1. *Wide-spread Data Replication*: The system must support the replication of data at *arbitrary* locations in the network to ensure data availability even after random peer departures or failures. The placement of replicas should not be limited to specific areas in the network, as this may weaken the spread of replicas.

2. *Replica Location*: Once replicas have been created and distributed throughout the network, there needs to be a mechanism to efficiently locate replicas when performing queries or data updates.

3. *Update Propagation*: When a data item is updated, its replicas should be updated as well. This update propagation may either be done eagerly or lazily. An eager approach would send updates to replicas immediately when data items are updated, while a lazy approach would delay update propagation for as long as possible until the system deems it necessary to send updates.

3. To develop and evaluate an update propagation algorithm for sending updates to replicas.

4. To develop a prototype implementation of our XML P2P data management framework as a platform for evaluating our work.

1.3 Scope and Limitations

We limit the scope of this work by making the following simplifying assumptions:

- Peers in the P2P network are homogeneous. They all have the same processing power, storage capacity and bandwidth capabilities. In addition, we assume that peers have an infinite amount of storage space available to accommodate all replicas sent to them, as we do not look at policies for deleting replicas when peers reach their maximum storage capacity.

- To process XML document updates, mechanisms are required for detecting changes between two versions of an XML document, representing update descriptions (deltas), and applying update descriptions to older versions of a document. We do not look at algorithms for performing such processing. We merely assume that they exist in the system.

- The system allows there to be a delay between the time a document is updated and the time the update is applied to a replica. Within this time period, the system does not prevent the older version of a replica from being accessed. Ensuring strong consistency would require locking mechanisms or quorum protocols [32, 34], an area that is well beyond the scope of this work. If the latest version of replica is not available at the time an access request is made, the system always tries to obtain the replica with the highest possible version number.

- Only a peer that originally contributed an XML document to the network is allowed to update it. If any peer was allowed to perform updates on any replica, then conflicts may arise when concurrent updates occur. Detecting and resolving such conflicts is beyond the scope of this work.

1.4 Dissertation Outline

This dissertation is structured as follows. Chapter 2 introduces the P2P computing field, and discusses how various P2P architectures affect data replication and replication location mechanisms. Chapter 3 gives an overview of current research into P2P data replication, replica location and update propagation techniques. Chapter 4 presents a survey of existing XML P2P data management systems. Our proposed XML P2P data management framework is described in chapter 5. It covers the fragmentation of XML documents, the location of XML fragments in the network and the processing of document updates. It also outlines a simple distributed XPath query processor that was used for evaluation purposes. Chapter 6 describes in detail our proposed data replication and update propagation algorithms. Chapter 7 presents a prototype P2P system that implements our XML P2P data management framework, as well as the replication and update propagation algorithms. Chapter 8 describes experiments that were conducted to evaluate the data replication and update propagation algorithms, and analyses the results obtained. Finally, chapter 9 concludes this dissertation and highlights areas of future work.

Chapter 2

P2P Systems

In this chapter, we look at various P2P architectures and discuss how their designs affect data replication and replica location mechanisms. This chapter serves two purposes:

1. To familiarise the reader with different P2P architectures; and

2. To justify our choice of using a separate data location catalogue for locating replicas in the network as mentioned in section 1.1.2.

2.1 P2P Computing

In the traditional client/server model of distributed computing, there are two types of components: a central server (or group of central servers) and a group of clients. The clients send requests to the server for access to certain services, while the server receives client requests, processes them and returns the result to the clients. This model is completely centralised, as the existence of the server is key to the successful operation of the network. The clients themselves never interact with each other, nor are they even aware of each other's existence in the network. Figure 2.1 shows an example of a typical client/server system.

The client/server model has three disadvantages. Firstly, the existence of a central server creates a single point of failure in the network. If the server fails due to a system crash or a malicious attack, the services offered by the server become unavailable to clients. Secondly, since the server is fully responsible for all client requests, performance degrades as more clients are introduced to the network and the load on the server increases. Finally, the organisation that owns

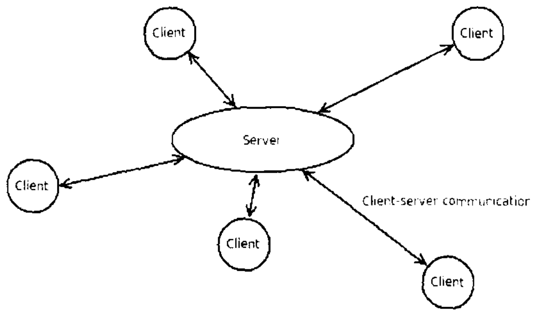


Figure 2.1: The client/server architecture. A central server (or group of servers) is fully responsible for servicing client requests. The clients themselves never interact with each other.

the server bears all the administration and maintenance cost involved in operating the server. Certainly, the availability of services and the scalability of the network can be improved by introducing more servers to the network and replicating services across all the servers. However, this further increases the administrative burden.

The P2P model, on the other hand, extends the client/server model by delegating part or all of the responsibility of the central servers to the clients in the network. Not only does this help eliminate single points of failure in the network and improve scalability, but the cost of administering and maintaining the network is shared by all participants in the network [3].

2.2 P2P Architectures

2.2.1 Overview

A P2P network consists of a set of nodes (the peers) that are logically interconnected in some manner to form an overlay network. An overlay network is a network that is implemented on top of an existing physical network infrastructure, typically spanning across private enterprise network boundaries such as NAT gateways. Since the overlay network does not correspond to the underlying network topology, the distance between two peers (in terms of the number of hops between them) does not necessarily reflect the communication cost between them.

Figure 2.2 shows an example of an overlay network implemented on top of the Internet Protocol (IP) [86]. The dashed lines indicate the physical connections between the peers in the underlying IP network, while the bold arrows represent the logical network connections between the peers in the overlay network. Each peer is assigned a location-independent ID, which allows

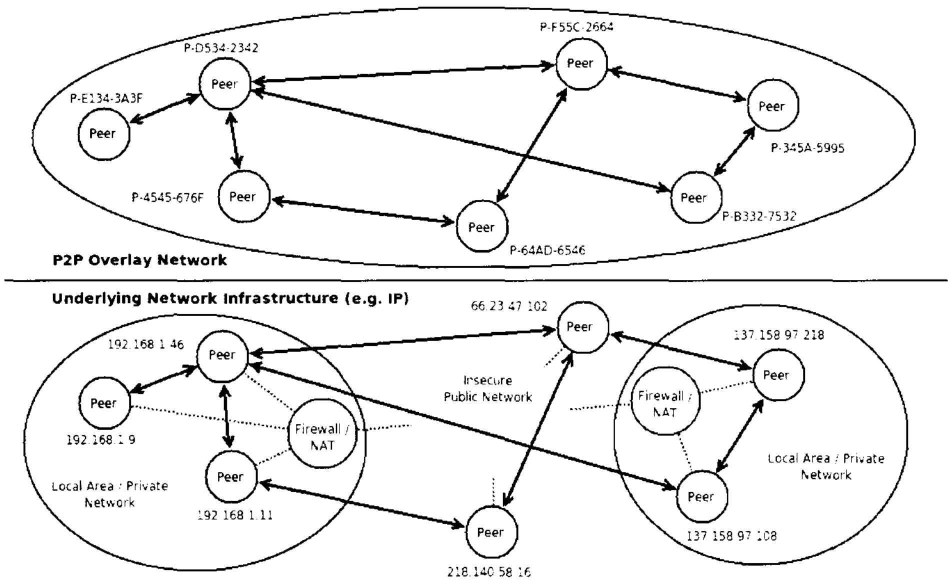


Figure 2.2: An example of a P2P overlay network built on top of the Internet Protocol (IP)

Distributed Computing Model	**Degree of Decentralisation**	**Overlay Network Structure**	**Examples**
Client/server	Completely centralised	Unstructured	HTTP server
P2P	Hybrid decentralised (super-peer)	Unstructured	Napster, Kazaa
		Loosely-structured	N/A
		Structured	N/A
	Purely decentralised	Unstructured	Gnutella
		Loosely-structured	Freenet
		Structured	Chord, CAN

Table 2.1: Classification of distributed systems

peers to be addressed in a consistent manner even when their physical locations change.

P2P overlay networks can broadly be classified in two ways: (1) by their structure, and (2) by their degree of decentralisation. Taken together, both the structure and degree of decentralisation of the overlay network affect the network topology, routing and searching algorithms used by the system, and ultimately, the fault-tolerance, self-adaptability, performance and scalability of the P2P network [3]. Moreover, it determines the data replication and replica location algorithms that can be used in the network.

We discuss the structure and decentralisation of overlay networks in the following two subsections. Our classification of P2P systems is presented in table 2.1.

2.2.2 Overlay Network Structures

The structure of the overlay network refers to the manner in which the network is constructed: whether it is created in an ad-hoc manner as peers enter and leave the network or determined by specific rules. It also refers to how data items are placed in the network. A P2P overlay network may be structured in three ways: it may be unstructured, structured or loosely-structured.

In *unstructured* overlay networks, the placement of data items in the system is completely unrelated to the topology of the overlay network [3]. There is no indication that certain peers store certain data items. Therefore, in order to locate data in such networks, unstructured P2P systems usually employ flood-based search mechanisms where search queries are propagated throughout the network by recursively broadcasting them to all neighbouring peers [75].

Structured overlay networks, on the other hand, place data items at precise locations in the network [3]. These networks are typically based on distributed consistent hashing schemes, where each data item is associated with a key, and each peer is assigned a range of key values corresponding to the data items it may store. Such structured overlay networks are essentially one large distributed routing table, which provides a mapping between the data item identifier and its location in the network. This distributed routing table is commonly called a distributed hash table (DHT) [22, 56, 61, 67].

Loosely-structured overlay networks also impose some form of structure on the network topology, but do not use consistent distributed hashing. Unlike structured overlay networks, the location of data in the network is not completely determined by the network topology. Instead, these networks typically cluster peers in the network into groups based on how "similar" peers are. Similarity among peers may be determined by a number of arbitrary properties. Peers may be clustered based on the type of data they store, how closely their data schemas match or how geographically close they are. In order to locate data in loosely-structured overlay networks, "routing hints" [3] are used to route queries to peers (or clusters of peers) which are *believed* to store certain data.

2.2.3 The Degree of Decentralisation in Overlay Networks

The degree of decentralisation in an overlay network refers to how much the P2P system depends on certain centralised components to operate. In particular, it refers to the extent to which the peers in the network accept server-like responsibilities. A P2P network may either be purely decentralised or hybrid decentralised.

In *purely decentralised* P2P systems, all peers perform exactly the same function. They act both as clients and servers. Through their client interface, peers accept requests from application users, and send those requests to remote peers in the network to initiate the exchange of content or participate in some online collaborative activity. Through their server interface, they receive requests from remote peers and then service the requests in some manner. Purely decentralised P2P systems are *true* P2P systems; they do not rely on any centralised components whatsoever in order to operate successfully. These type of P2P systems may have unstructured, structured or loosely-structured overlay networks.

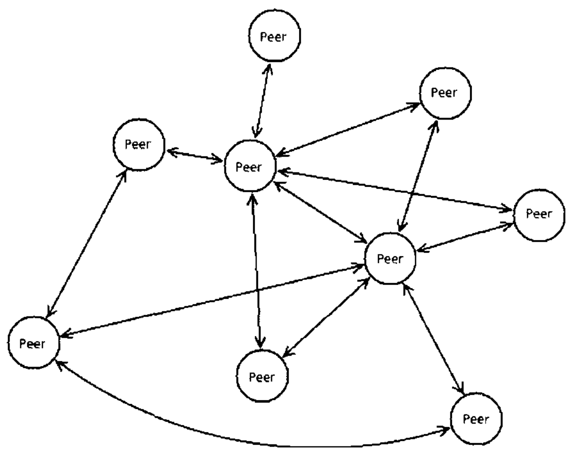


Figure 2.3: An example of a purely decentralised P2P system (with an unstructured overlay network). No distinction is made between clients and servers. All peers perform the same role in the network.

Hybrid decentralised P2P systems, on the other hand, still largely depend on centralised components to coordinate resources and for the successful operation of the network. These type of P2P systems have much in common with traditional client/server models, except that data are stored on the clients themselves rather than a central location, and clients interact among themselves to transfer data. The central server merely acts as a directory for finding data in the network. Figure 2.4 shows the architecture of a hybrid decentralised P2P system. An example of such a system is Napster [11].

Another class of hybrid decentralised P2P systems that are closer to purely decentralised P2P systems are super-peer networks. In super-peer networks, certain peers called super-peers are assigned additional roles in the network. These super-peers perform extra administrative duties such as acting as local central directories and proxying requests through the network on behalf of the regular peers. A peer is assigned the status of a super-peer either voluntarily, or automatically by the system based on its storage and bandwidth capabilities and processing power.

Since super-peer networks are more decentralised than the normal hybrid decentralised P2P systems, some authors such as [3] refer to these as *partially centralised* P2P systems to differentiate

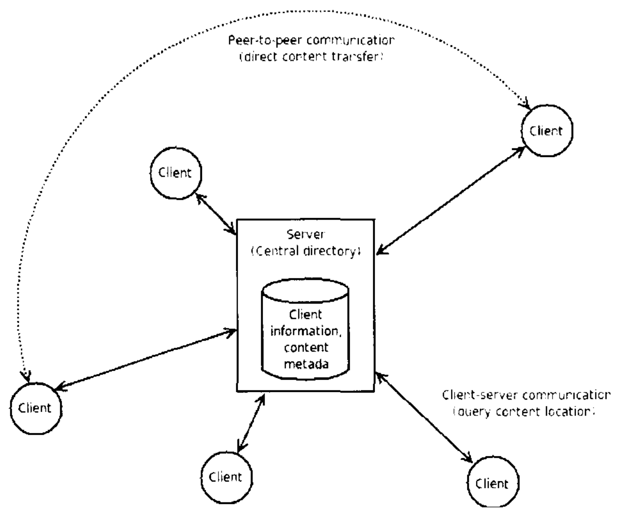


Figure 2.4: The hybrid decentralised P2P architecture. The central server maintains information about peers and data in the network and coordinates the interaction between the peers. The actual data transfer takes place directly between the peers.

them from normal hybrid decentralised P2P systems. However, the fact that the super-peers act as centralised components in the network, means that they can still be regarded as hybrid decentralised P2P systems. The only real difference between normal hybrid decentralised P2P systems and super-peer networks is that all normal hybrid decentralised P2P systems are based on unstructured overlay networks, whereas super-peer networks may use either unstructured, structured or loosely-structured overlay networks, or a combination. For example, an unstructured overlay network could be used between the super-peers and the normal peers, and between the normal peers themselves, whereas the connections between the super-peers could be arranged in some structured manner. Figure 2.5 shows an example of a super-peer network, where the super- and normal peers are connected in an unstructured manner, whereas the connections between the super-peers form a structured ring topology.

2.3 The Implications of P2P Architectures on Data Replication and Replica Location

Having discussed the various types of P2P overlay networks in terms of their structures and degrees of decentralisation, we now look at how each P2P architecture affects data replication. In particular, we determine whether they fulfill the following two requirements from section 1.1:

1. Support for *wide-spread data replication* to allow replicas to be placed on arbitrary peers

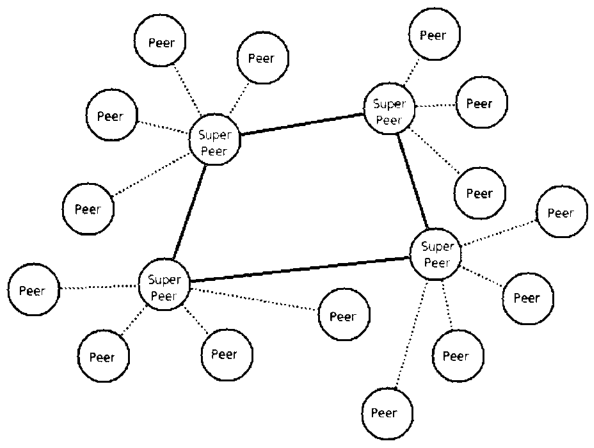


Figure 2.5: An example of a super-peer network

in the network, rather than only a few specific peers;

2. Support for *efficient data location* to allow the system to quickly determine the location of replicas in the network when propagating data updates

We look at each of the following types of P2P networks: unstructured, structured and loosely-structured P2P systems.

2.3.1 Unstructured P2P Systems

Unstructured P2P systems do not impose any rules on where data should be stored in the network. As a result, they are well-suited to wide-spread data replication, as the placement and number of replicas in the network are not restricted. However, due to this unrestricted data placement, the system has no global knowledge of where data are stored. Consequently, in order to locate data, the system needs to do a complete search of the network. This is generally performed using a broadcasting algorithm that floods the network with lookup messages until data items of interest are found or the search times out. An example of this broadcasting algorithm is the lookup mechanism used by the Gnutella protocol [75].

In Gnutella, lookups are performed by recursively sending lookup messages to each neighbouring peer. When these neighbouring peers receive the lookup messages, they forward it to each of *their* neighbours which, in turn, forward it to each of their neighbours. This process continues until the data is found. To prevent endless propagation of lookup messages in the network, each message is assigned a time-to-live (TTL) value that is decremented at each hop. When the TTL value reaches zero, the lookup message is removed from the network.

This broadcasting algorithm is illustrated in figure 2.6. In figure 2.6(a), peer A, sends a lookup message to all of its immediate neighbours: peers B, C and D. During the next hop (figure 2.6(b)), peers B, C and D forward the request to each of their immediate neighbours: B sends it to E, D sends it to C, while C sends it to D, E, F, H and I. Note that the messages that C and D send to each other will be dropped to avoid causing loops. Also, even though peer A is an immediate neighbour of B, C and D, the message is not sent back to A. In figure 2.6(c), the data is found at peer H, which responds with an appropriate reply message. This reply message is routed along same path through the network as the original lookup message, but in the opposite direction.

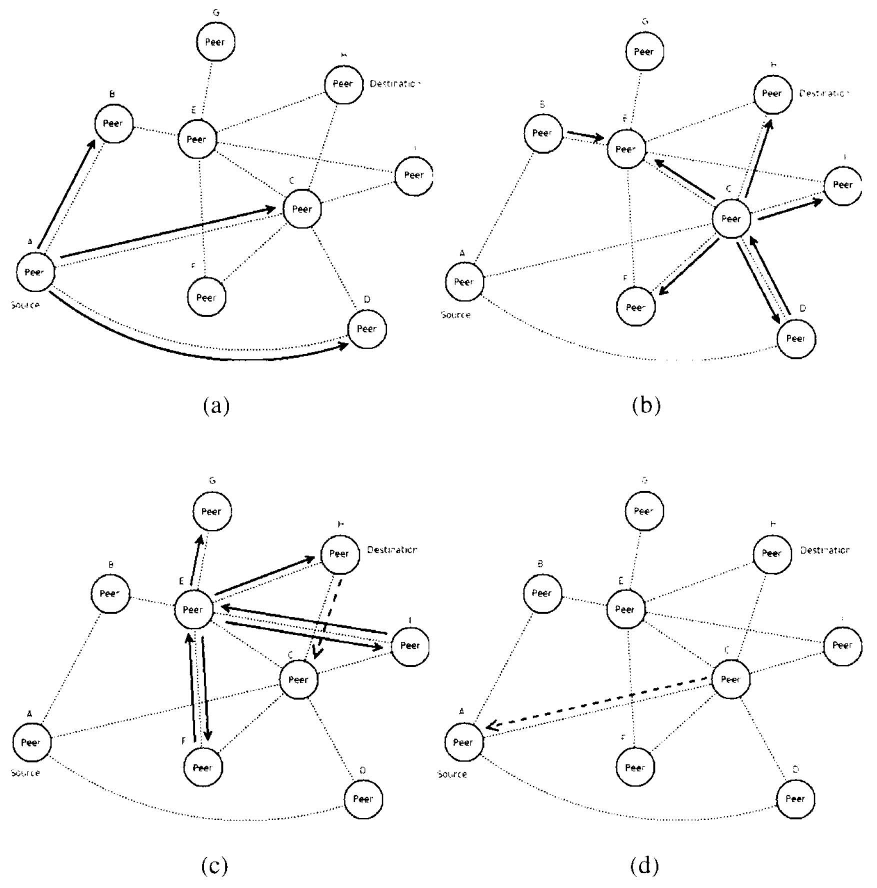


Figure 2.6: The Gnutella data location algorithm. The dashed lines represent the logical connections between the peers in the network. The bold solid and dashed arrows indicate the flow of lookup and lookup response messages, respectively.

The main problem with this data location algorithm is that it does not scale well. It incurs a heavy load on the network due to the high bandwidth required to transmit all the lookup messages. The larger the network, the greater the number of messages required to locate data. Furthermore, the TTL value assigned to messages effectively imposes a horizon in the network beyond which lookup messages cannot reach. If a replica exists at a location in the network that requires $(TTL + 1)$ message hops, then that replica will never be found. Therefore, while unstructured P2P systems are good for wide-spread data replication, they are let down by the inefficiency of their data location algorithms.

2.3.2 Structured P2P Systems

Structured P2P systems attempt to address the scalability issues of unstructured P2P networks by placing data at precise locations in the network. Data lookup operations are much more efficient, taking approximately $O(logN)$ message hops through the network, where N is the number of peers in the network [3].

Chord [67] is an example of a structured P2P protocol that is used to implemented DHTs. It uniquely maps a set of data IDs (or keys) to a set of peers. Each peer in the network is assigned a unique peer ID and is arranged in a circular topology as shown in figure 2.7. A data ID i is mapped to the peer with the smallest peer ID greater than or equal to i. This peer is called the *successor* of i. The successor of data ID 0 in figure 2.7, for example, is peer 2, since 2 is the smallest ID in the network greater than or equal to 0.

Each Chord peer maintains a routing table called the finger table that consists of $m = log_2(N)$ entries. Each entry in the finger table contains the ID and address of a successor that follows the peer in the circular topology. The ith entry in peer a's finger table is the smallest peer b that is greater than or equal to $(a + 2^i)mod(2^m)$. For example, in figure 2.7, the second entry in peer 2's finger table refers to peer 5, since peer 5 has the smallest ID greater than or equal to $(2 + 2^1)mod(2^3) = 4$.

Determining the location of a data item with an ID x in the network is equivalent to finding the successor of x. This is performed by first checking if x is between the ID of the current peer and the ID of the immediate successor, s. If it is, the lookup terminates, and peer s is returned. Otherwise, the lookup request is forwarded to the peer in the finger table with the largest ID preceding x. This process repeats until the successor of x if found. For example, assume a user at peer 7 in figure 2.7 wishes to determine the location of a data item with ID 4. The immediate successor of peer 7 is peer 2. Since 4 is not between 7 and 2, peer 7 forwards the request to peer 3, the peer in its finger table with the largest ID preceding ID 4. At peer 3, 4 is between 3 and 5. Since 5 is the smallest ID greater than or equal to 4, the search terminates, and peer 5 is returned.

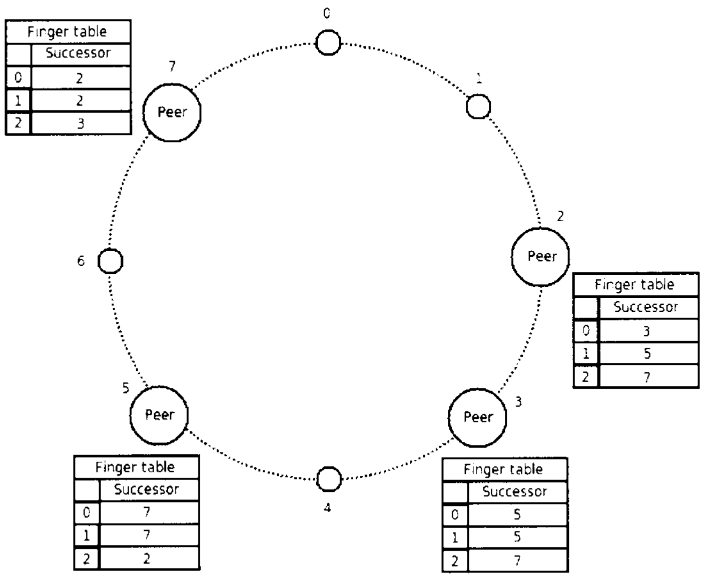


Figure 2.7: An example of a Chord P2P network

While systems like Chord provide efficient $O(logN)$ data location mechanisms, they restrict the placement of data in the network, and thus, do not support wide-spread data replication. In our example, a data item with ID 4 may only be placed on peer 5 in the network. Otherwise, the lookup algorithm will fail.

Dabek et al [21] suggest that data replication in Chord should be performed by placing a data item not only on its immediate successor, but on its k immediate successors in the ring, where k is less than or equal to the size of the finger table, m. Therefore, if k is 2 in our example, then a data item with ID 4 would be stored on peers 5 and 7 in figure 2.7. If the immediate successor then departs the network, the data item will immediately be available on one of the other k successors. The data item is also copied to the peer in the ring that becomes a new successor of the data item. This way, a constant k number of replicas of a particular data item is maintained in the network. For example, if k is 2 and peer 5 exits the network, then data item 4 will be copied to peer 2 so that it exists on both 7 and 2.

One problem with this replication approach, however, is that it requires data items to be copied to peers *whenever* a new peer becomes one of the k successors. If the rate of peer joins and departures is high, the k successors will constantly change, resulting in many data item transfers being made in an attempt to maintain the k replicas. If the sizes of the data items are very large, then the bandwidth cost incurred by all these transfers will be significant.

2.3.3 Loosely-Structured P2P Systems

Loosely-structured P2P systems can be regarded as a hybrid between unstructured and structured P2P systems. They impose some structure on the network like structured P2P systems. Yet, like unstructured P2P systems, the location of data in the network is not necessarily restricted by the network topology.

The data location mechanisms used in loosely-structured P2P systems are similar to those in unstructured P2P systems. However, the flood is constrained by only forwarding lookup messages to specific neighbouring peers, rather than all neighbours. The decision for which neighbours a lookup message should be forwarded to is based on a number of application-specific factors. Some loosely-structured P2P systems such as [42] use indexing structures that summarise the data stored at the peers reachable through the network links. These indexing structures provide hints on where the lookup message should be propagated next.

While this data location algorithm is more scalable than the unconstrained flood in unstructured P2P systems, it still has some limitations. Firstly, if the aforementioned indexing structures are used as routing hints, then whenever data items are stored or deleted at a peer, the indexing structures summarising that peer's data need to be updated as well. Propagating these index updates incurs bandwidth costs and may take some time to reach all peers in the network. Also, the indexing structures may only be probabilistic data structures. If they give false positives, unnecessary paths in the network will be followed [42]. Finally, some loosely-structured P2P systems such as FreeNet [20] assign TTL values to lookup messages. This has the same horizon effect as in unstructured P2P systems, where peers beyond the horizon are unreachable.

2.4 Discussion

Looking at the various P2P architectures, we observe the following: *there is a conflict between support for wide-spread data replication and support for efficient replica location.*

On the one hand, we have unstructured P2P systems that allow data to be placed anywhere in the network, and thus, fulfill our need for wide-spread data replication. However, the data location algorithms used in those systems makes locating replicas in the network very inefficient. On the other hand, we have structured P2P systems that provide more efficient $O(logN)$ data lookups. However, such systems restrict the placement of data on only a few specific peers in the network, and thus, restrict the spread of replicas.

To solve this dilemma, we propose the use of a separate global data location index (or catalogue) that maps the IDs of the replicas to the IDs of the peers in the network that store those replicas. This is the approach taken in this dissertation to locate replicas. The idea is based on the replica location mechanisms proposed in [10, 17], and is discussed in further detail in the following chapter. The global data location catalogue may either be implemented as a web service or, if complete decentralisation is desired, on top of a DHT. In this work, we are not concerned with how the catalogue is implemented. Such implementation details are orthogonal to our work and left for future research.

One consequence of using a global data location catalogue is that it allows the data replication and update propagation algorithms proposed in this dissertation to work at a higher level, without depending on the architecture of the underlying P2P network or its internal implementation details. If an unstructured P2P system is used, the catalogue can be consulted to quickly determine which peers in the network store a particular replica. If a structured P2P system is used, the structured network itself could act as the catalogue, storing the pointers to peers with replicas rather than storing the replicas itself.

Chapter 3

Existing P2P Replication, Location and Update Propagation Techniques

This chapter presents some of the research that has been conducted into data replication and update propagation techniques in P2P systems. Our discussion focuses on data replication, replica location and update propagation.

3.1 Data Replication

In [55], a data replication strategy is described where each peer in the network maintains an approximate model of the entire network. This model is used to determine the number of replicas to create, the time the replicas should be created and the location where the replicas should be placed in order to maintain a desired user-specified data availability level. This approach relies on a resource discovery service to find available storage in the network to place replicas, and a tool to provide information about the current state of the system. The model consists of four parameters: the probability that a peer is online, the ratio between the size of the data to replicate and the available bandwidth, the cost of storing a data item at a particular peer, and the accuracy of the replica location mechanism.

The number of replicas to create is computed as a function of the model parameters and the desired level of data availability. Once this number has been calculated, the replica location mechanism is used to determine the number of replicas currently in the network. If the number of existing replicas is less than some threshhold, the system recreates the difference and inserts it into the network.

In order to determine where the newly created replicas should be placed, the resource discovery mechanism is used to obtain a set of candidate storage peers. These candidates must have sufficient storage space available to store the replica, must have transfer times below a certain threshhold, and must not already store a copy of the data to be replicated. The best candidates are selected using a heuristic algorithm that seeks to maximise the difference between the replication benefit and the replication costs. The replication benefit is the reduction in transfer time from the candidate peer to potential requesters of the replica. The replication costs are the costs of storing the replica at the candidate peer and the time to transfer the replica from the current peer to the candidate peer.

To determine when replicas should be created, the authors propose performing periodic checks to determine the number of replicas in the network. The frequency of these checks is altered based on how often the checks reveal that more replicas should be created. If consecutive checks show that more replicas need to be created, the frequency of the checks is increased, and vice versa.

TotalRecall [7] also tries to dynamically adjust its replication strategy to ensure a user-specified level of data availability. It continuously monitors the availability of the peers in the network to measure two peer behavioural patterns: short-term and long-term (or permanent) peer departures from the network.

Using the short-term peer departure measurements, TotalRecall tries to maintain a certain level of data redundancy by creating additional copies of data in the network. The number of copies to create is calculated as a function of the target data availability and the mean availability of the peers in the network. To handle long-term or permanent peer departures, a data repair process is used to reinsert redundant data back into the network onto new peers. Two data repair processes are proposed: eager and lazy repair. Eager repair immediately repairs the loss of data whenever peers leave the network. This ensures that the data redundancy level remains constant. However, this results in high bandwidth costs, especially when peers constantly leave the network. Furthermore, eager repair does not make any distinction between short-term and long-term peer departures. Therefore, many repair procedures may unnecessarily be executed. With lazy data repair, the repair process is delayed for as long as possible. This result in reduced bandwidth costs. However, it requires that the availability and data of every peer in the network be tracked in order to determine when repair processes should be initiated.

The problem of placing replicas in the network to optimise data availability is considered in [25, 24]. The authors decompose the problem into two phases: initial placement and placement improvement. The initial placement phase is concerned with determining suitable locations in the network to store newly created replicas. The placement improvement phase uses a distributed hill-climbing algorithm to continually relocate replicas to improve data availability. Placement improvement is performed by swapping replicas between two peers in the network in order to bring the availability of the swapped replicas closer together. Three placement improvement algorithms are proposed: RAND-RAND, MIN-RAND and MIN-MAX. In the RAND-RAND algorithm, the replicas to swap are randomly selected by the peers. In MIN-RAND, one peer selects the minimum-availability replica from its data store, while the other peer selects a replica randomly. Finally, in MIN-MAX, one peer selects minimum-availability replica, while the other selects the maximum-availability replica.

Using simulations, it was shown that initially placing replicas on highly available peers in the network skews the distribution of free space in the network. Greater storage costs are imposed on high availability peers compared to lower availability peers. It also weakens the spread of replicas throughout the network, because replicas are placed only on a relatively small number of peers in the network. The authors thus conclude that replicas should initially be placed at random locations in the network. With regard to placement improvement, MIN-MAX was found to be the most efficient of the three hill-climbing algorithms, while RAND-RAND was the most effective.

Junqueira et al [40] also look at replica placement. They argue that replica placement algorithms should take the diversity in the network into account, and bias the placement of replicas on peers that are "different". A peer is considered "different" based on a number of factors: different geographical locations, different operating systems or software versions, and so on. The rationale behind this approach is that the availability of "similar" peers may be correlated in some manner. For example, if a large-scale attack on the network causes a peer to fail due to some vulnerability in the software the peer is running, then there is a good chance that another peer running the same software will fail as well. A model is developed in [40] to represent diversity, and simulation results are reported. However, no empirical results for an actual P2P system are provided.

3.2 Replica Location

The approach taken in this dissertation to locate replicas in the network is to maintain replica ID to peer ID mappings in a globally accessible catalogue. There are various existing systems that implement such a global catalogue. Giggle [17], which is part of the Globus Toolkit version 3 [10, 74], is one example.

The Giggle framework consists of 2 indexing structures: a Local Replica Catalogue (LRC) and a Replica Location Index (RLI). The LRC keeps information about replicas stored at a particular peer. It maintains a table of logical-to-physical name mappings, where the logical name is the data item ID and the physical name is the location of the data item in the network. Each RLI keeps a collection of entries that map the logical names to LRC addresses. This allows the information of one or more LRCs to be aggregated into a single index to support the lookup of mappings from multiple LRC sites. Determining the location of a data item in the network thus involves querying the RLI to obtain the addresses of LRC sites. Then, the LRCs are queried to lookup the physical names corresponding to the logical name. RLI mappings are assigned timeout values and must periodically be refreshed. This allows the system to automatically remove RLI entries associated with failed or inaccessible LRC sites.

For increased reliability and load balancing, multiple (and possibly redundant) RLI sites are deployed in a hierarchical manner. An example of this deployment is shown in figure 3.1.

P-RLS [10] is an extension of Giggle built on top of Chord. It attempts to eliminate the administration overhead involved in manually configuring a hierarchical network of RLI sites by dynamically organising sites into a P2P overlay network. Like Giggle, P-RLS consists of a network of LRCs and RLIs. However, RLIs are extended to operate as Chord peers. In order to resolve a particular logical name, P-RLS uses the logical name as a key into the Chord network to retrieve the addresses of LRC sites. The system then queries each of the returned LRCs to obtain the physical names corresponding to the logical name. P-RLS takes advantage of Chord's successor node information to operate reliably when peers depart the network. Mappings are replicated at the k successor nodes in the Chord ring as explained in section 2.3.2.

Boundary Chord [39] also uses Chord to manage its network of Replication Service Peers (RSPs). RSPs are organised in a multi-ring topology based on the logical domain in which each RSP resides. A logical domain is group of peers that belongs to a particular virtual organisation participating in the network. Each RSP consists of a Local Replica Discovery (LRD) compo-

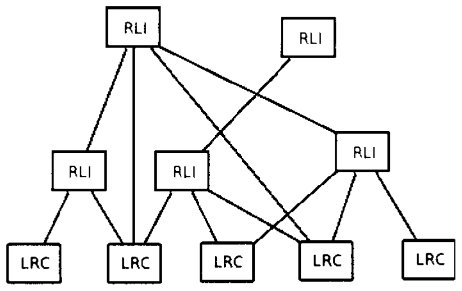


Figure 3.1: An example of the hierarchical deployment of RLI and LRC sites in Giggle

nent, which stores logical-to-physical name mappings, and a Local Replica Index (LRI) component, which maintains logical-name-to-LRD mappings. Physical name lookups in Boundary Chord is performed in the same manner as in P-RLS.

Ripeanu and Foster describe an alternative P2P-based approach in [57]. In contrast to P-RLS and Boundary Chord, their system does not route lookup queries through the network. Instead Replica Location Nodes (RLNs) are organised into an unstructured P2P overlay network, where each RLN distributes the collection of logical names for which it stores logical-to-physical name mappings to all other RLNs in the network. Each RLN thus eventually gains an entire view of the mappings stored in the network. To reduce bandwidth costs, the collections of logical names that RLNs propagate are summarised using Bloom filters [8]. Bloom filters are compact probabilistic data structures that are used to quickly determine whether a given element belongs to a set. They are much smaller in size than the set they summarise.

Logical-to-physical lookups are performed as follows. When a RLN receives a lookup query, it first checks if a mapping for the given logical name is stored locally. If so, the associated physical name is returned. Otherwise, it checks the Bloom filter summaries it received from remote RLNs to see which RLN in the network might store physical names corresponding to the requested logical name. If such a remote RLN is found, the request is forwarded to that RLN to obtain the physical name.

The advantage of this approach is that query latency is kept low. It requires at most one message hop in the network to perform lookup queries. However, the cost of performing mapping updates is high, as updates need to be propagated throughout the entire network to reach all RLNs.

3.3 Update Propagation

In [23], a push-pull rumour-spreading algorithm is proposed to propagate updates to replicas. During the push phase, updates are propagated throughout the network using a constrained Gnutella-like flooding scheme, in which update information is recursively broadcast to a selected number of neighbouring peers. During the pull phase, offline peers that rejoin the network search for data updates in an attempt to synchronise their replicas with those stored in the rest of the network. Multiple peers are contacted, and the update is pulled from the peer with the most up-to-date copy. Due to the flooding scheme, the network overhead involved in pushing updates through the network is significant, especially when data are frequently updated.

Wang et al [70] propose an alternative approach using a bidirectional chain structure as shown in figure 3.2. Each data item in the network has a logical replica chain composed of all the peers in the network storing replicas of the data item. Each replica holder maintains a list of the k nearest peers in the chain in both directions. These k peers are called probe peers. When a data item has been updated, the updating peer pushes the update to its online probe peers. The farthest online probe peer in the chain receiving the update then forwards the update to all its online probe peers in the same direction through the chain. This process continues in a recursive manner until the update has reached all the online peers in the chain. When an offline peer rejoins the network, it synchronises its replicas with those in the rest of the network by contacting an online probe peer and pulling updates from it.

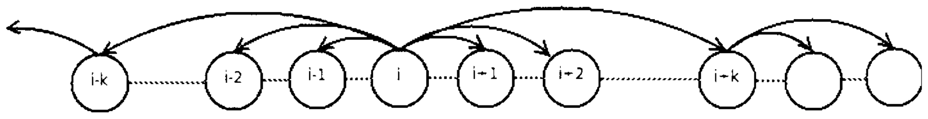


Figure 3.2: The update propagation chain proposed in [70]. The arched arrows indicate the flow of update messages from peer i through the chain.

A similar technique is presented in [71] that uses a tree structure instead of a chain. An n-ary tree is constructed for each data item in the network, where the root of the tree is the owner of the data item, and the other nodes are the peers storing replicas of the data item. Whenever a data update occurs, the update is propagated through the tree from the root to the leaf nodes.

One limitation of all update propagation techniques described in this section is that they do not take the access frequencies of replicas in the network into account. Whenever an update occurs, it is immediately propagated to all replica holders, even if the replicas stored there are accessed very infrequently. This may result in unnecessary bandwidth costs.

Chapter 4

Managing XML Content in P2P Systems

In this chapter, we briefly describe the XML data format, and then discuss examples of existing XML P2P systems.

4.1 XML - The eXtensible Markup Language

XML [93] is a markup language for describing and storing structured information. It is an open standard recommended by the World Wide Web Consortium (W3C) [91], and has become the *de facto* means of transferring data between heterogeneous systems on the Internet [88, 92].

Data in XML are described in plaintext documents using a vocabulary of tags (or elements). This vocabulary is not predefined. Instead, users are responsible for specifying their own vocabulary of tags to describe their data. This is often accomplished by defining a Document Type Definition (DTD) [96] or an XML Schema [97], a set of rules that dictate the names, structure and data types of the elements allowed an XML document. An example of an XML document is shown in figure 4.1.

4.2 Existing XML P2P Systems

In this section, we survey five existing XML P2P systems. We also briefly discuss the various data replication and update mechanisms in these systems.

```
<library>
  <books>
    <book catalogId="JA01" reserved="yes">
      <title>A Beginners Guide to Java</title>
      <author>J. Ava</author>
      <year>2001</year>
    </book>
    <book catalogId="GOF94">
      <title>Design Patterns for Dummies</title>
      <author>G. O. Five</author>
      <year>1994</year>
    </book>
  </books>
</library>
```

Figure 4.1: An example XML document

4.2.1 BRICKS

The BRICKS project [58] aims to integrate digital libraries across Europe in order to provide location-transparent access to cultural information. It is built on top of a DHT, and stores XML documents by splitting them into pieces and spreading the pieces amongst the peers in the network. The XML pieces are grouped into sets, each with a unique set ID. These sets usually contain related XML pieces, such as whole XML subtrees or siblings, where each XML piece is assigned an ID unique to that set.

In order to store data in the network, the set IDs are used as keys into the DHT, and the corresponding sets are placed on the peers to which the keys map. To retrieve data, a reference to the root element in the XML document needs to be known. This root reference is a cryptographic hash of a symbolic name associated with the XML document. The pieces which make up the XML document are then retrieved from all the peers on which they are stored, and combined. Retrieving the pieces requires that a peer have knowledge of both the ID of the set in which the desired piece is stored, as well as the ID of the desired XML piece within the set.

With regard to replication, BRICKS uses a modified read-one-write-all-available (ROWA-A) [66] replication protocol to maintain data availability. Missing replicas are periodically recreated and re-inserted into the network. The frequency of these re-insertion periods is determined by measuring the average data availability in the network. If the average data availability is above a certain threshhold, re-insertion periods are made less frequent. If the data availability is less than the threshhold, the frequency is increased.

Versioning is used to support data updates. Whenever a data item is updated, its version number is increased. It is then assumed that the data item with the highest version number is the latest version of the data item.

No experimental data replication and update results are provided.

4.2.2 XP2P

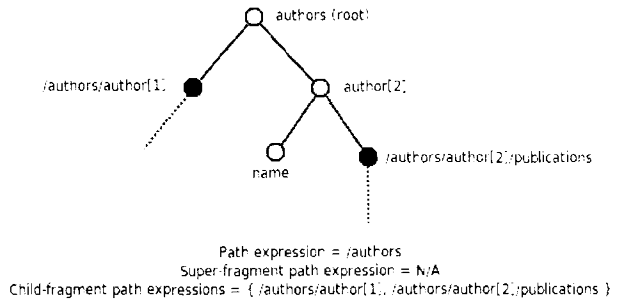
XP2P [9] is also built on top of a DHT. It extends Chord to support the lookup of XML fragments using linear path expressions of the form: $s_1[i]l_1s_2[j]l_2...s_n[k]$, where s_n is a path expression step, l_n is either a child (/) or descendant (//) axis, and $[k]$ is an optional positional filter. XML fragments are defined as XML subtrees that may contain special substitute elements in place of missing child fragments.

To store an XML fragment in the system, XP2P hashes the path expression of the fragment into the Chord virtual space, and then stores the fragment on the peer to which the path expression is mapped. The path expression of an XML fragment is defined as the distinct linear absolute path starting from the root of the XML document to the root fragment in the document. This is illustrated in figure 4.2. If two distinct XML fragments have the same path expression, the path expression is prefixed with the name of the originating document.

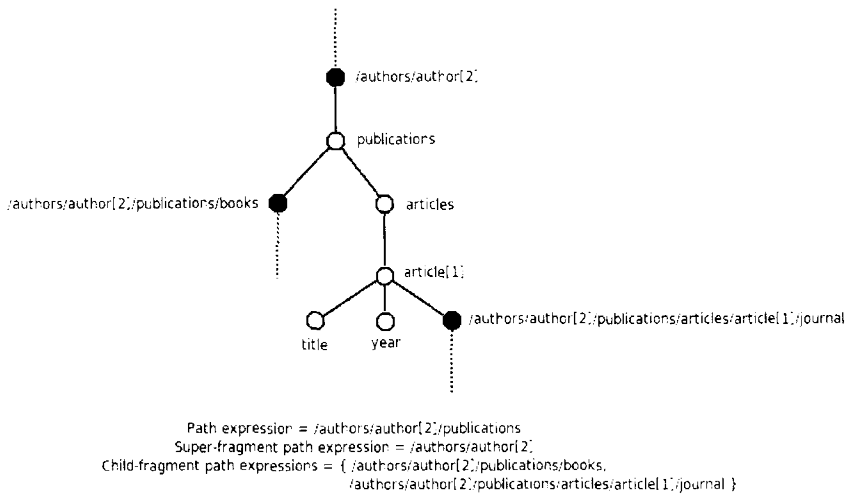
In addition to storing the content of the XML fragment, XP2P also stores a set of path expressions of related XML fragments. This set consists of the super-fragment path expression, as well as several child-fragment path expressions. The super-fragment path expression is the path expression of the XML fragment which is the ancestor of fragment being stored, whereas the child-fragment path expressions are the path expressions of all the XML fragments of which the fragment being stored is an ancestor. The related path expression set is used during data lookups to link related XML fragments.

XP2P takes advantage of the Chord overlay network structure to replicate XML fragments for increased reliability. In Chord, the peers are organised in a logical ring topology, where each peer stores list of its k successors in the ring. Instead of storing a XML fragment only at a peer's immediate successor, it is replicated at each of the k successors. However, as discussed in section 2.3.2, this replication scheme restricts replicas to only a few specific peers, weakening the spread of replicas through the network. Also, high bandwidth costs are incurred to maintain replicas at the k successors when the rate of peer arrivals and departures is high.

No information is provided on how replicas are kept up-to-date when one or more of the k successors depart the network and rejoin after the data updates have occurred.


authors (root)
/authors/author[1]
author[2]
name
/authors/author[2]/publications
Path expression = /authors
Super-fragment path expression = N/A
Child-fragment path expressions = { /authors/author[1], /authors/author[2]/publications }

(a)


/authors/author[2]
publications
/authors/author[2]/publications/books
articles
article[1]
title
year
/authors/author[2]/publications/articles/article[1]/journal
Path expression = /authors/author[2]/publications
Super-fragment path expression = /authors/author[2]
Child-fragment path expressions = { /authors/author[2]/publications/books,
/authors/author[2]/publications/articles/article[1]/journal }

(b)

Figure 4.2: Determining the path expressions of XML fragments in XP2P. Each circle in the diagram represents an element in the XML fragment. The filled circles are special substitute elements which stand in place of missing child or super fragments.

4.2.3 XPeer

XPeer [65] uses a hierarchical unstructured super-peer architecture to facilitate distributed XML query processing. Peers are organised into clusters based on the similarity of their data schemas. Each cluster is managed by a single super-peer that is responsible for performing administration tasks within the cluster. These tasks include query evaluation, maintaining schema information of the peers in the cluster and tracking the arrivals and departures of peers. Peers become super-peers on a voluntary basis, but still maintain their normal peer status.

Data in XPeer are represented as unordered forests of node-labelled trees, where each tree is augmented with the location of the peer that stores it. The trees are assigned a freshness parameter that indicates the last time an update was performed on the trees. Each peer exports a

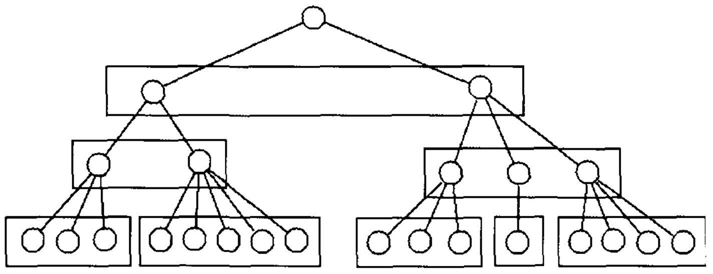
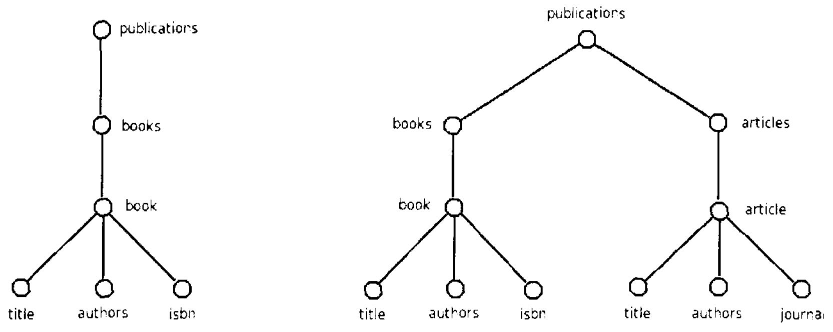


Figure 4.3: The hierarchical unstructured super-peer architecture of XPeer. The circles represent the peers in the network, while the rectangles indicate the super-peer clusters. The peer that is the parent of all the peers in a cluster is the super-peer.

description of its data in the form of a tree-shaped DataGuide [33] that is automatically inferred from the data using a tree search algorithm. This DataGuide contains each distinct path in the XML documents stored by the peer, and is used to identify relevent data sources during query compilation. The DataGuides for all the peers within a particular cluster are maintained by the cluster super-peer, and are intergrated to form one DataGuide for the entire cluster. Figure 4.4 shows an example of a DataGuide for a single peer and an integrated DataGuide for the cluster.



(a) An example of a DataGuide exported by a single peer in a cluster

(b) An example of a DataGuide stored at the super-peer. This DataGuide is the union of all DataGuides of the peers in the cluster

Figure 4.4: Examples of the tree-shaped DataGuides used by XPeer. These DataGuides are used during query compilation to find peers storing relevent data

XPeer uses the FLWR [99] core of XQuery as its query language, without the *ORDER* clause. Query compilation is performed in two phases. In the first phase, the peer issuing the query translates it into a location-free algebraic expression. This expression contains "holes" in the places that indicate peer locations. During the second phase, the query is sent to the each super-peer, where the holes are replaced with the actual locations of the peers containing relevant data. This is done by sending the query to the super-peer managing the cluster. This super-peer matches the query with the integrated DataGuide for the cluster to find all the peers within the cluster that store the desired data. Then, the super-peer sends the query to its parent super-peer which, in turn, matches the query with the DataGuide for *its* cluster. This process continues until the query has propagated up the entire network hierarchy to the root peer. Once this process is complete, the query is sent back to the issuing peer, where it is executed. Query execution involves applying common algebraic rewriting to the query, and splitting it into sub-queries that are sent to the relevent peers.

Replication is performed within the clusters to balance the load and to exploit peers with large computational resources. Even though data updates are allowed, there is no scheme to ensure data consistency. Instead, replicas are assigned a time parameter that determines their lifetimes. XPeer also incorporates operators in its query algebra that allow the user to specify how fresh the replicas should be for them to be included in the query result [64].

4.2.4 Content-Based Multi-Level Bloom Filters

In [42], a loosely-structured XML P2P system is discussed. Each peer stores a set of XML documents, where each document is represented as an unordered labelled tree.

A constrained flooding algorithm is used to locate data. The query is propagated through the network by iteratively sending it to neighbouring peers that are believed to store relevant data. In order to perform this constrained flood, each peer maintains a local index that summarises its local data, as well as a merged index that summarise the data of all the peers that can be reached through the links to neighbouring peers. To route a query, a peer first consults its local index to see whether it matches the query. Then, it routes the query along the links whose merged indices also match the query. This process continues until either a maximum number of hops have been reached or until the query reaches a peer that has no merged index that matches the query. In this case, the routing backtracks to the previous peer, who then propagates the query through another link that matches the query and that has not been followed yet.

The indexing schema used in [42] is based on Bloom filters [8]. Bloom filters provide an efficient and compact means of summarising the data stored by the peers in the network. Not only are they much smaller than the XML documents which they summarise, but performing comparison operations on Bloom filters is more efficient than performing comparison operations on the XML documents themselves. In order to support the evaluation of XPath expressions, multi-level Bloom filters are introduced. These multi-level Bloom filters preserve the hierarchical relationships between the nodes in the XML document.

The disadvantage of using Bloom filters, however, is that they sometimes give false positives. That is, they may incorrectly indicate a match. This results in paths in the network being followed that lead to peers storing no relevant data.

Another limitation of this indexing approach is the high cost of performing data updates. Whenever an update occurs, not only does the local index need to be updated, but also all the merged indices in the network that use the local index. This means that the peer performing the update needs to propagate its local index throughout the network in a manner similar to a query. In an attempt to improve the efficiency of propagating the updated local index, an update propagation method is proposed in [44] where peers only propagate bits in the Bloom filter that have changed rather than the entire Bloom filter.

4.2.5 Distributed XML Catalogues

A distributed catalogue framework for locating XML data sources is proposed in [30]. The system uses the Chord overlay network to map XML element names to sets of data summaries. These data summaries consist of two parts: a structural summary and a value summary.

The structural summary summarises the structure of the XML documents stored in the system. It is a set of all possible unique path expressions that lead to a particular element in the XML documents stored at a peer. For example, in figure 4.5(a), the structural summary for the *author* element at peer p_1 is the set, {*/library/book, /library/article*}. The value summary summarises the actual contents of the XML documents. It is some domain-specific description of the XML data, such as a value range (to support range queries) or a Bloom filter (to support equality queries).

Whenever a peer joins the network, it inserts *(key, data summary set)* pairs into the system by hashing the element names to keys in the Chord virtual space, and sending the data summaries corresponding to the element names to the peers responsible for those keys. Each peer in the network maintains a catalogue, such as a B+ tree [4], for storing the data summaries sent to it. Figure 4.5(b) shows an example of how the data summaries for the path expressions in figure 4.5(a) might be distributed in the network.

In order to locate data, the name of the element of interest is hashed into the Chord virtual key space to derive the DHT lookup key. The query is then sent to the peer to which the DHT key maps. The peer receiving this query matches the query to the data summaries in its catalogue to determine the peers that store relevant XML documents. This list of peers is then returned to

Peer	Path expressions
p_1	/library/book/author /library/article/author ...
p_2	/bookstore/book/price /bookstore/book/author ...
p_3	/people/address/city ...

(a) A sample collection of path expressions for XML documents stored in the network

DHT key	DHT values	Peer storing DHT values
book	$Summary_{p_1}$ = {/library} $Summary_{p_2}$ = {/bookstore}	p_2
author	$Summary_{p_1}$ = {/library/book, /library/article} $Summary_{p_2}$ = {/bookstore/book}	p_3
price	$Summary_{p_2}$ = {/bookstore/book}	p_1
city	$Summary_{p_3}$ = {/people/address}	p_2
...	...	...

(b) The contents of the DHT after inserting the data summaries for the path expressions in figure 4.5(a)

Figure 4.5: An example of how XML data in a P2P network are indexed using the distributed XML catalogue approach proposed by [30]

the querying peer. Finally, the querying peer issues the query to each peer in the list to perform the actual query processing on the XML documents.

When data are updated, the data summaries stored in the catalogues need to be updated as well. This process is performed in the same manner as data insertions. Since the DHT keys of the updated data summaries remain the same, an insertion simply overwrites the old data summary entries in the catalogue. In order to handle data consistency issues, data updates are restricted to the peer that created the data summaries (i.e. the owner of the data).

The system also employs a replication scheme. However, replication is performed on the data summaries rather than the actual XML documents. Furthermore, replication is used for load balancing rather than to increase data availability. Whenever a peer detects that queries for a particular DHT key mapped to it exceed some limit, it replicates the data summaries corresponding to that key on one or more peers in the network. Then, it creates mappings in its catalogue that point to the new locations of the data summaries. During data lookups, these mappings are handed to the querying peer, and are used to query the new locations one after the other in a round-robin fashion.

4.3 Discussion

In this chapter, we surveyed five existing P2P systems that use XML as their underlying data format. We observe that all of these systems either do not consider data replication for maintaining data availability, or do not handle data replication in a way that fulfill our requirements in section 1.1.

The DHT-based systems such as BRICKS and XP2P cannot store replicas at arbitrary locations in the network, as the DHT restricts the spread of replicas. XPeer, being an unstructured network, could place replicas anywhere in the network. However, XPeer only performs replication within super-peer clusters. Inter-cluster replication is not considered. The system proposed in [42] does not consider data replication issues, nor does [30].

In the following chapter, we present our own XML P2P framework. This framework was designed to meet our requirements in section 1.1.

Chapter 5

A P2P Framework for Replicated XML Document Fragments

Before we could embark on our study of replication and update propagation algorithms, there was a need to develop a XML P2P framework. This would allow for the evaluation of any number of alternative replication strategies and update propagation techniques.

To make this possible, the core mechanisms and support structures in the framework had to be identified and investigated. There were two objectives. Firstly (and most importantly), was the definition and design of components that would enable us to meet our requirements identified in section 1.1. That is, wide-spread data replication, replica location and update propagation. The second goal was to find the most general approach for each component so that the framework could serve as a base for future experimentation and be reused with different P2P architectures.

This chapter describes this framework. The following issues are discussed:

- *System constraints.* Issues considered here include the type of data each peer stores, the data ownership and access rights (whether peers may update or delete data), as well as how data items in the system are identified.
- *The fragmentation of XML documents.* This allows us to investigate the replication of smaller XML document pieces in order to reduce replica transfer cost. Of primary interest here is the manner in which XML documents are fragmented, and how fragments within XML documents are identified and extracted.
- *The location of replicas in the network.*

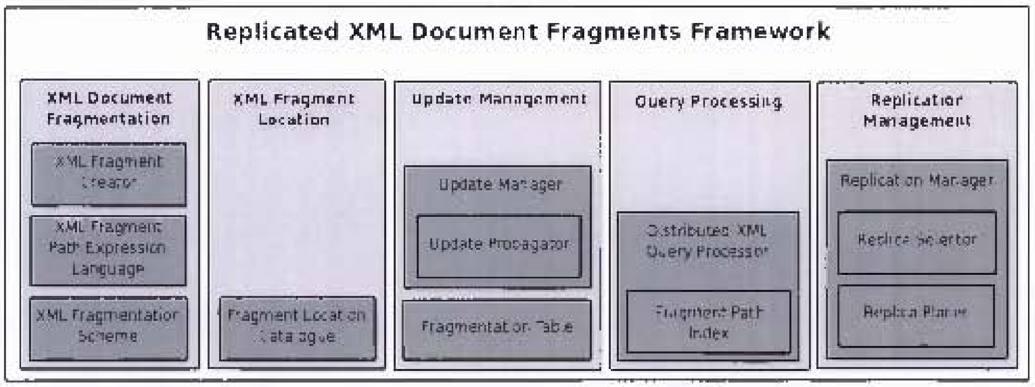


Figure 5.1: An overview of the replicated XML document fragments framework

- *Update management.* We specify the how updates are applied to XML documents, and what types of document update are supported.

- *Query processing.* While this is not a focus of our work, support for basic distributed query processing is described. The query processing mechanisms presented in this chapter were used to evaluate the replication and update propagation algorithms.

An overview of the components and mechanisms provided by the framework is shown in figure 5.1. Replication management is discussed in chapter 6.

5.1 System Constraints

Each peer in the network stores a collection of XML documents. The owner of a document is the peer that contributed it to the network. In addition to these documents, each peer may also maintain a set of replicas. These replicas are XML documents or pieces of XML data (i.e XML fragments) that were extracted from XML documents and distributed throughout the network. The fragmentation of XML documents is discussed in section 5.2. In this section, we simply refer to these replicas as *XML fragments*.

Peers have complete control over the XML fragments they store. They can delete them any time at will, or further replicate them if the need arises. However, peers may not update XML fragments. Only the peer which owns the original XML document from which the fragments were extracted may update them. Allowing any peer in the network to update XML fragments may result in conflicts if two or more peers perform concurrent updates on replicas of the same fragment. Resolving these conflicts is a challenging problem and is still an area of active research [27, 46, 50]. Another restriction on peers is that they may not further fragment XML fragments, as this would complicate updates.

Each XML fragment is assigned a unique ID. These IDs may be any identifier assigned by the application. In our work, IDs are generated as follows:

$$id = hash(peerId + rootElementName + R)$$

where $peerId$ is the ID of the peer which owns the fragment, $rootElementName$ is the name of the XML element that is the root of the unfragmented XML document, and R is an array of randomly generated bytes. The function, $hash()$, is a cryptographic hash function such as SHA-1 [84] that, with very high probability, produces a unique output for every distinct input. Concatenating the peer ID and R to the input of the hash function ensures that the ID of the fragment is globally unique throughout the system. The length of R is 20 bytes. This size should be enough to minimise the possibility that two different fragments with the same root element name on a particular peer receive the same ID.

In addition to the ID, each XML fragment is also assigned a version number to facilitate data updates. This version number is incremented whenever the fragment is updated.

5.2 Fragmentation of XML Documents

When performing data replication, copying entire XML documents may be quite expensive, especially when documents are large. It would be more efficient if only parts, or fragments, of documents are extracted and replicated, rather than entire XML documents. This will result in reduced networking cost when distributing replicas through the network. Furthermore, since only document fragments are replicated, the replica creation algorithm can choose to replicate only those parts of documents which are in demand, resulting in less unnecessary data being circulated in the network. The decision what fragments to extract from the documents is made by the replica creation algorithm, and is discussed in section 6.1.2. This section defines what the term *fragment* means in this dissertation, and describes our XML fragmentation model.

Initially, our plan was to support arbitrary fragmentation. This would allow the replication creation algorithm to extract specific pieces from XML documents for replication, excluding any unnecessary parts. Figure 5.3 illustrates such an arbitrary fragmentation scheme on the example XML document shown in figure 5.2. The circles drawn around the document nodes in figure 5.3 indicate the fragments extracted from the document.

Allowing fragmentation to be performed in this arbitrary manner means that fragments may contain "holes". These holes are places in the fragment where elements present in the original document were excluded. For example, in fragment $f1$ in figure 5.3, there would be holes in the positions occupied by elements b, h and i. Our solution to "fill" these holes was to insert placeholder elements within the fragments to act as references to the missing elements. This is reminiscent of the substitute elements mentioned in section 4.2.2 that are used in XP2P. These placeholder elements would contain `ref` attributes whose values are the path expressions or IDs of the subtree or fragment rooted at the missing element. Fragment *f1* would then have the structure shown in figure 5.4.

Whenever a query processor then encounters a placeholder element when traversing the document during query evaluation, it would use the reference to locate and retrieve the missing fragment, or alternatively, ship the query to the peer storing the missing fragments for further evaluation.

One problem with this arbitrary fragmentation approach, however, is that it incurs networking overhead whenever a placeholder element is encountered. If the document is "finely" fragmented, there would be many placeholder elements, resulting in a high network overhead. Other issues such as ensuring that the references to missing fragments are correct when the original XML document changes also presented a problem.

Therefore, instead of using an arbitrary fragmentation scheme, we opted for a simpler approach in which we constrain the fragmentation to *whole* XML subtrees. Our definition of an XML fragment as used in this dissertation is thus as follows:

XML fragment Given an XML document represented as a tree of nodes, t, with a root node, r, an XML fragment, f, is a subtree of t rooted at some node n in t, where n is an XML element node that is either a descendant of r or equal to r itself. An XML fragment is thus either a subtree in an XML document or the entire XML document. The smallest unit of fragmentation is a single XML element (a leaf node in t).

Figure 5.5 shows an example of how the sample document in figure 5.2 may be fragmented using this fragmentation scheme. Limiting the fragmentation to whole subtrees does mean that more data may be extracted from documents and replicated in the network, and thus involve greater networking cost when transmitting replicas between peers. However, we believe that this is the best approach, as it avoids the extra networking and computational overhead required by a more arbitrary fragmentation scheme, while still offering the benefit of replicating smaller XML fragments as opposed to replicating only entire XML documents.

```
<a>
    <b>
        <e><j><s/></j></e>
        <f><k/></f>
        <g><l/><m/><n/></g>
    </b>
    <c>
        <h><o/><p/></h>
        <i><q/><r/></i>
    </c>
    <d/>
</a>
```

Figure 5.2: An example XML document before fragmentation

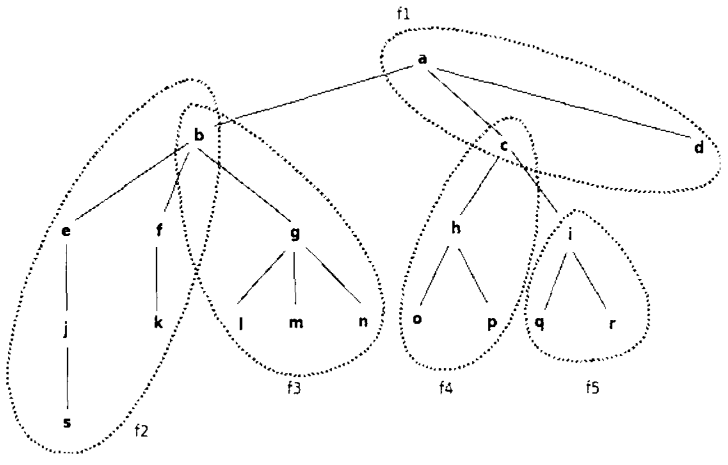


Figure 5.3: Fragmenting the XML document in figure 5.2 using some arbitrary fragmentation scheme

```
<a>
    <placeholder ref="f3"/>
    <c>
        <placeholder ref="ref_to_subtree_rooted_at_h"/>
        <placeholder ref="f5"/>
    </c>
    <d/>
</a>
```

Figure 5.4: XML fragmentation with placeholder elements. The fragment shown is fragment $f1$ in figure 5.3.

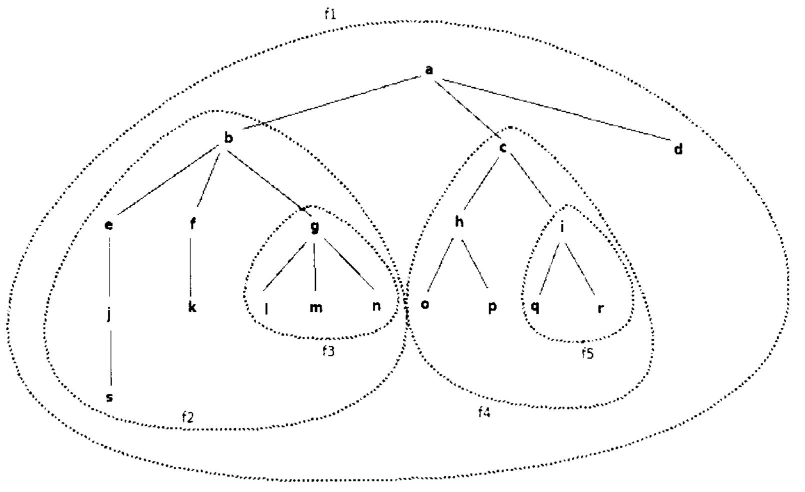


Figure 5.5: Fragmenting the XML document in figure 5.2 by extracting whole subtrees

5.3 XML Fragment Path Expression Language

XML documents are fragmented using a *Fragment Creator* component. It receives a path expression and an XML document as input, and extracts the subtree identified by the path expression. This is illustrated in figure 5.6.

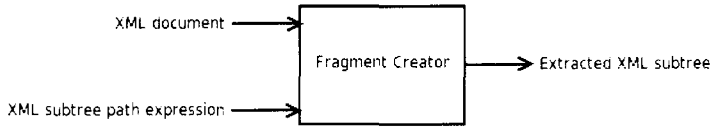


Figure 5.6: The Fragment Creator component

The path expression could be expressed in XPath. This would allow the system to use any standard XPath query processing tool to perform the subtree extraction. However, using XPath has two disadvantages. Firstly, XPath query processors depend on an in-memory model of the *entire* XML document to operate [98]. This model is usually represented as a DOM tree [95]. If there is insufficient memory available to accommodate the DOM tree for the entire XML document, subtree extraction will not be possible. Secondly, XPath query processors typically traverse the XML document twice: once when the DOM tree is being constructed, and again when traversing the DOM tree during query evaluation.

A more efficient approach than DOM is to extract the subtree on-the-fly as the XML document is being parsed, without depending on an in-memory model of the document. SAX parsers [87] are ideal for this. SAX parsers work by invoking callback procedures whenever certain events (e.g. encountering start and end tags) occur. These callback procedures are implemented by the application, allowing XML processing to occur on-the-fly as the document is being parsed. Also, since SAX parsers do not build a document tree in memory, the memory requirement of SAX parsers is very low.

However, the use of SAX also presents a problem: no information regarding the structure of the XML document is kept and made available to the application. As a result, using complicated XPath-like path expressions for addressing subtrees is not possible. Instead, a simpler path expression language is required that allows a SAX-based Fragment Creator to extract subtrees without knowing the document structure.

In this work, we use path expressions of the form

$$/p_1/p_2/\ldots/p_n$$

where p_i is an integer indicating the position of an element at level i in the XML document tree. Table 5.1 shows the path expressions for the example fragments $f1$ to $f5$ in figure 5.5.

Fragment	Path Expression
f1	/1
f2	/1/1
f3	/1/1/3
f4	/1/2
f5	/1/2/2

Table 5.1: Path expressions for the example fragments shown in figure 5.5

Using this path expression language, the Fragment Creator only needs to keep track of the level and position of an element in the XML document while the document is being parsed. It does not need any other information regarding the structure of the document. This path expression language is also more concise than XPath. The implementation of a Fragment Creator is described in section 7.4.

5.4 Locating XML Fragments in a P2P Network

Once fragments have been extracted and replicated through the network, there needs to be a mechanism to find fragments during query processing or when performing updates. These fragments may be placed anywhere in the network on any number of peers. To handle this, we introduce a component called a *Fragment Location Catalogue* (FLC), based on the replica location systems by [10, 17] described in section 3.2. The reason for using such a catalogue for locating fragments in the network were discussed in chapter 2.

Conceptually, the FLC can be viewed as a global lookup table mapping the ID of a fragment to the IDs of all the peers in the network that store that particular fragment. When a peer receives a fragment from another peer, it inserts a $(fragmentId.peerId)$ entry into the FLC, where $fragmentId$ is the ID of the fragment, and $peerId$ is the ID of the receiving peer. The FLC may be implemented as either a central registry accessible through a web service, or in a distributed manner on top of a P2P network such as a DHT.

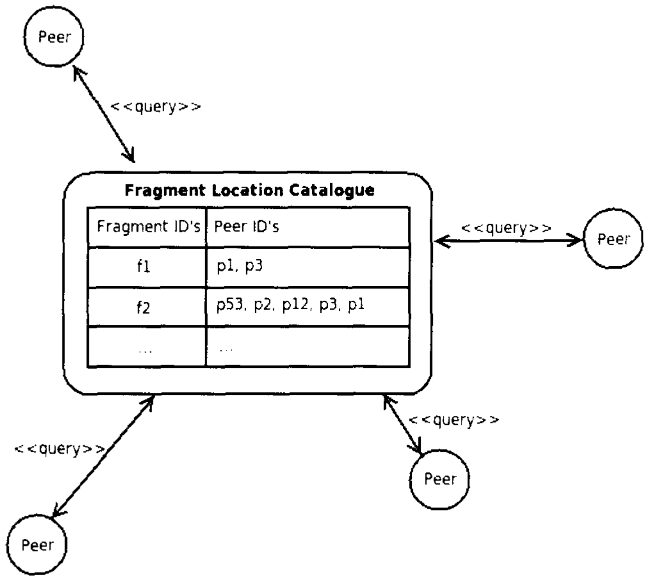


Figure 5.7: A conceptual view of the global Fragment Location Catalogue (FLC)

5.5 Update Management

When the original XML documents are updated, the replicas in the network need to be updated as well in order to keep them consistent with the original data. This section describes how updates are handled in the framework. The update propagation algorithm is discussed in section 6.2.

5.5.1 The Fragmentation Table

The Fragmentation Table maintains information about the manner in which local XML documents at a peer have been fragmented. It allows the system to determine what fragments in the network need to be updated when the original XML document from which the fragments were extracted has changed. The Fragmentation Table consists of two lists: a list of Document Information Nodes (DINs) indexed by the IDs of the XML document from which fragments were extracted, and a list of Fragment Information Nodes (FINs) indexed by the IDs of the extracted fragments. Figure 5.8 shows a diagrammatic view of the Fragmentation Table.

Each DIN keeps fragmentation information specific to a particular XML document. It stores:

- A list of IDs of all the fragments extracted from the XML document; and
- A table of $(pathExpression, FIN)$ mappings, where $pathExpression$ is a path expression in the form described in section 5.3, and FIN contains information about the corresponding fragment.

Information pertaining to extracted fragments are stored in FINs. This is:

- The ID of the extracted fragment;
- The ID of the XML document in the local data store from which the fragment was extracted;
- The path expression identifying the fragment in the XML document;
- A list of IDs of all the fragments that are *ancestors* of the extracted fragment; and
- A list of IDs of all the fragments that are *descendants* of the extracted fragment.

An ancestor and descendant fragment is defined as follows.

Ancestor and descendant fragments If T_1 and T_2 are two subtrees in some XML document D_i, then T_1 is an ancestor of T_2 if T_2 is a subtree of T_1. Conversely, if T_2 is a subtree of T_1, then T_2 is a descendant of T_1.

An example of the information stored by the Fragmentation Table for fragments $f1$, $f4$ and $f5$ shown in figure 5.5 is given in figure 5.9.

By arranging the Fragmentation Table in this two-level structure, the system can obtain information about extracted XML fragments in the following manner. If the system wants to obtain

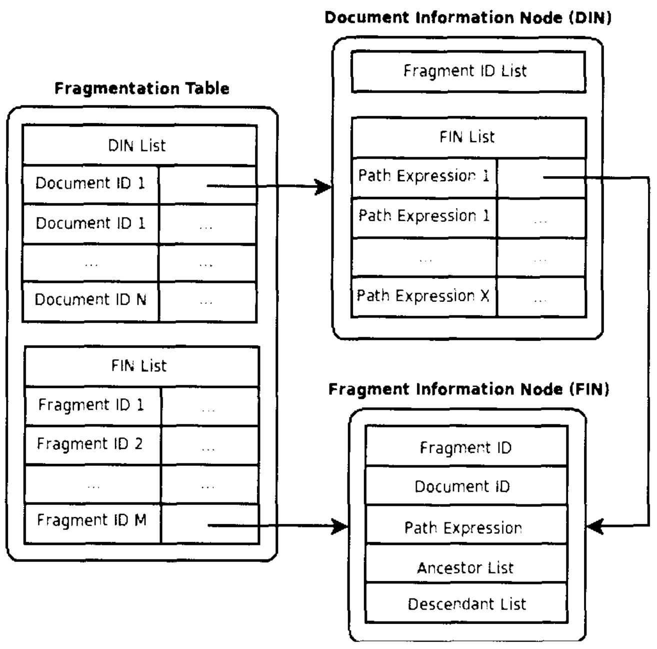


Figure 5.8: The Fragmentation Table

information about a particular extracted fragment when it only knows the ID of that fragment, it can lookup the fragment in the FIN list. If the system wishes to determine what fragments where extracted from a particular XML document, it can lookup the DIN for that document. Finally, if the system wants to retrieve information about a particular fragment when given a document ID and path expression, it can lookup the DIN for the document, and use its FIN list to obtain the FIN mapped to the given path expression.

The Fragmentation Table is maintained locally on the peer which created the fragments. It is populated whenever a fragment is extracted from an XML document. If no fragments were extracted from an XML document, then there will be no entries in the Fragmentation Table corresponding to that document.

5.5.2 Processing Document Updates

When a document has been updated, updates need to be propagated to the peers in the network that store replicas affected by the update. To accomplish this, we define an *Update Manager* component which, when given a list of fragments that have been created from a document, as well as the old and new versions of the document, returns a list of IDs of all the fragments that were updated, along with update descriptions (or deltas) that specify how each updated fragment has changed. This Update Manager is shown in figure 5.10.

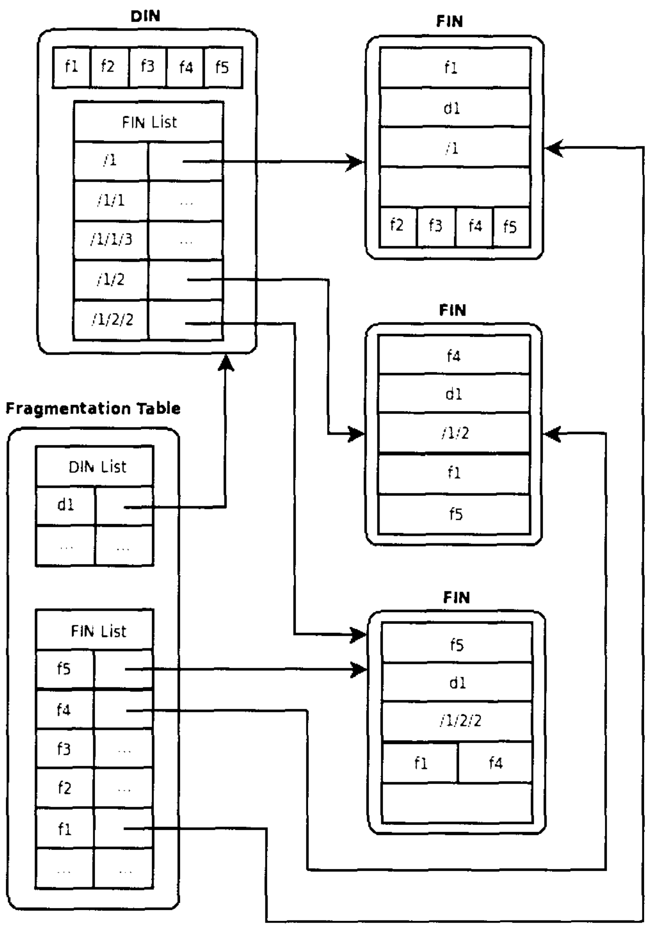


Figure 5.9: An example of the information stored in the Fragmentation Table for fragments $f1$, $f4$ and $f5$ in figure 5.5

Note that our framework does not require the update descriptions to be expressed in any specific format. Such details are entirely implementation-dependent. Since encoding update descriptions is well beyond the scope of our work, we simply assume that it would be possible for a peer receiving a series of update descriptions to derive the current version of a fragment by applying the update descriptions to the old version of the fragment. Examples of existing update description formats that may be used with XML data are described in [14, 18, 19, 46, 49, 60].

In what follows, we briefly outline the manner in which document updates are processed.

5.5.2.1 Types of XML Fragment Update

We consider the following four classes of update which may apply to XML fragments. Note that in this section, we make a distinction between an XML fragment and a (regular) XML subtree. A fragment is a subtree that has been extracted from a document and replicated in the

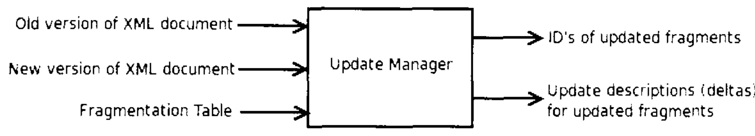

Old version of XML document
New version of XML document
Fragmentation Table
Update Manager
ID's of updated fragments
Update descriptions (deltas) for updated fragments

Figure 5.10: The Update Manager viewed as a black-box

network (and thus has a corresponding entry in the Fragmentation Table), whereas a subtree has not been made into a fragment.

Subtree insertion: A subtree insertion occurs when a new subtree is added to a fragment.

Subtree deletion: A subtree deletion occurs when a subtree is removed from a fragment.

Content change: A content change occurs when the text content of an element in the fragment is updated, or when an element's attributes change.

This class of update can be further decomposed into the four types of update: value updates, attribute insertions, attribute deletions and attribute value updates. However, in this dissertation, we group them under one classification, because they do not affect the element structure of the fragment whose content changed.

Fragment move: A fragment move occurs when a fragment is moved from one part of the document to another. This may further result in two additional updates: a subtree insertion into the fragment where the moved fragment was placed, and a subtree deletion from the fragment where it had been. The additional two updates will not occur if the places where the moved fragment was inserted and deleted were not fragments themselves.

Fragment moves only affect the path expression of the fragment, not its structure or content. This type of update is thus not propagated to peers storing replicas of the moved fragment. The path expression as well as the ancestor and descendant information in the Fragmentation Table are simply updated.

When updates are performed on fragments, other fragments in the document may be affected by the update as well. In particular, inserting a new subtree into a fragment f means that the subtree is also being inserted into all of f's ancestors fragments. This bubbling-up of updates may be explained with the aid of figure 5.11. Inserting a new subtree into fragment $f3$, means that the subtree is also being inserted into fragments $f2$ and $f1$. The same occurs during subtree deletions and content changes.

Updates may also bubble-down the document. This happens during a subtree deletion when the subtree being deleted is a fragment itself. All the descendant fragments of that deleted fragment

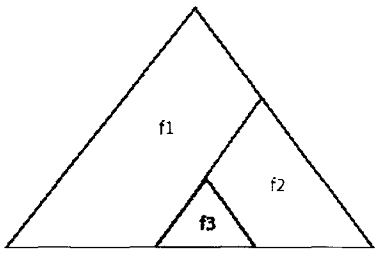


Figure 5.11: The bubbling of updates through an XML document. Inserting or deleting a subtree in fragment f3, means that a subtree has been inserted or deleted in fragments f2 and f1 as well.

are then considered deleted as well. For example, if the subtree being deleted is rooted at $f1$ in figure 5.11, then $f2$ and $f3$ are deleted as well.

Note that subtree insertions and deletions cause the path expressions of fragments to change. For example, inserting a new element between elements b and c in figure 5.12, causes c's path expression to change from $/1/2$ to $/1/3$. Whenever the Update Manager detects that a fragment's path expression has changed, it updates the path expression information in the Fragmentation Table.

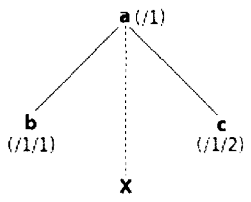


Figure 5.12: An example of how a subtree insertion affects the path expression information

5.5.2.2 Outline of the Update Process

Now that we have described the types of update that may be applied to XML fragments, we give a short high-level description of how the Update Manager might determine what fragments in a document have been updated.

First, the ID of the XML document being updated is used as input to the Fragmentation Table to lookup the IDs of all the fragments that have been extracted from that document. Then, for each fragment ID, the corresponding fragment is identified in the old and new versions of the document, and a comparison operation is performed to determine whether the fragment has been updated. If so, the fragment is marked as updated, and a description of the update is calculated and associated with the fragment. If the update was a subtree insertion, subtree deletion or a content change, then all the ancestors of the fragment are marked as updated as well, and are associated with the update description. If the fragment itself was deleted, then all the fragment's descendants are also marked as deleted. If a fragment's path expression has changed as a result of a subtree insertion, subtree deletion or fragment move, the fragment's path expression information is updated in the Fragmentation Table.

Note that comparing XML documents in order to detect updates is an XML differencing and tree matching problem that is beyond the scope of this work. Therefore, we do not look at how the system determines what fragments in an XML document have been updated and how they have been updated. We simply assume that an such algorithm is available to the system. For an in-depth discussion on methods for detecting changes in XML documents, the reader is encouraged to refer to [2, 13, 14, 15, 26, 38, 45, 46, 47, 48, 69].

5.5.2.3 Example

The following example illustrates how the update process works. Consider the XML document in figure 5.13(a) expressed as a tree of nodes. This document has five fragments extracted from it, labelled $f1$ to $f5$. The Fragmentation Table information for these five fragments is shown in table 5.13(b). Now, assume the document has been updated in the following manner:

1. *Subtree deletion*: element e is deleted from $f1$. Fragment $f1$ is marked as updated and a subtree deletion description is associated with it. Since $f1$ does not have any ancestor fragments, the subtree deletion does not bubble-up the document and no other fragments are marked as updated.

2. *Subtree insertion*: a new element r is inserted as a child of element a. Since a is not part of a fragment that appears in the Fragmentation Table, no fragments are marked as updated. The insertion does, however, affect $f3$'s path expression. Therefore, the path expression for $f3$ in the Fragmentation Table is changed from $/1/3$ to $/1/4$.

3. *Content change*: the text content of element k is changed to some value v. Since element k is part of fragment $f4$, fragment $f4$ is marked as updated and a content change update description is associated with it. The update bubbles-up to fragment $f4$'s ancestor fragment, $f1$, which is also marked as updated and associated with the content change update description.

4. *Fragment move*: element g, which is the root of fragment, $f5$, is unlinked from its previous parent, element c, and made a child of the newly inserted element r. The path

expression for fragment $f5$ is changed from $/1/2/1$ to $/1/3/1$ in the Fragmentation Table. Since $f5$ was previously a subtree of fragment $f2$, $f2$ is marked as updated and a subtree deletion description is associated with it.

The diagram in figure 5.13(c) shows the document after it has been updated. Figure 5.13(d) lists the updated fragments along with their update descriptions. Fragment updates would now be propagated to all replicas of fragments $f1$, $f2$ and $f4$ in the network, as dictated by the update propagation algorithm used by the system.

5.6 Query Processing

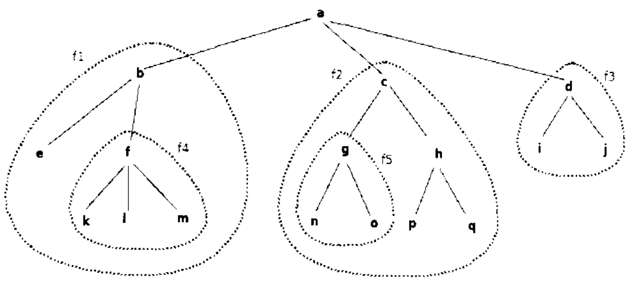
In this section, we describe a simple distributed XPath query processor for our framework. Since query evaluation is not the focus of our research, the purpose of the query processor was to simply provide a means of finding data in the network so that we could test the data replication and update propagation algorithms proposed in this dissertation. As a result, efficiency was sacrificed for simplicity.

5.6.1 Design Considerations

In designing the query processor, the following challenges were encountered:

1. *Dealing with fragmented data.* In our framework, XML data are fragmented and distributed throughout the network. The query processor needs to be able to find peers in the network storing fragments relevant to a particular query. While the Fragment Location Catalogue provides a means to determine the locations of fragments, it does not capture any path information. It merely acts a table of fragment ID to peer ID mappings. Consequently, there is no way to determine based on the information in the Fragment Location Catalogue what fragments in the network a given XPath query expression applies to.

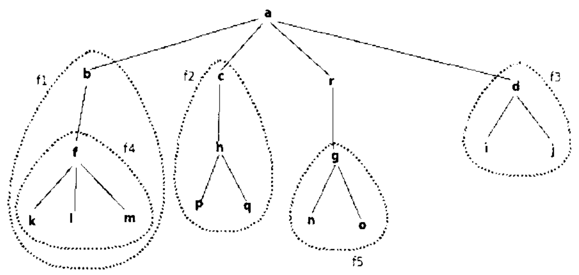
2. *Support for branch queries.* When evaluating branch queries, certain paths in the XML document will be excluded from the result. For example, the query */a/b[@x='y']/c* requests only the elements, c, whose parents, b, have x attributes with a value y. Some peers may return empty query result sets if the fragments they store do not have such branches. Sending queries to such peers would only result in unnecessary networking cost and increase query latency. It would be more efficient if, before sending queries to peers for



(a) The document structure before the update

Fragment ID	Path Expression	Ancestors	Descendants
f1	/1/1	–	f4
f2	/1/2	–	f5
f3	/1/3	–	–
f4	/1/1/2	f1	–
f5	/1/2/1	f2	–

(b) The Fragmentation Table information for the five fragments in figure 5.13(a)



(c) The document structure after the update

Fragment ID	Update Descriptions
f1	subtreeDelete(e)
	textUpdate(k, v)
f2	subtreeDelete(g)
f4	textUpdate(k, v)

(d) The updated fragments and their update descriptions

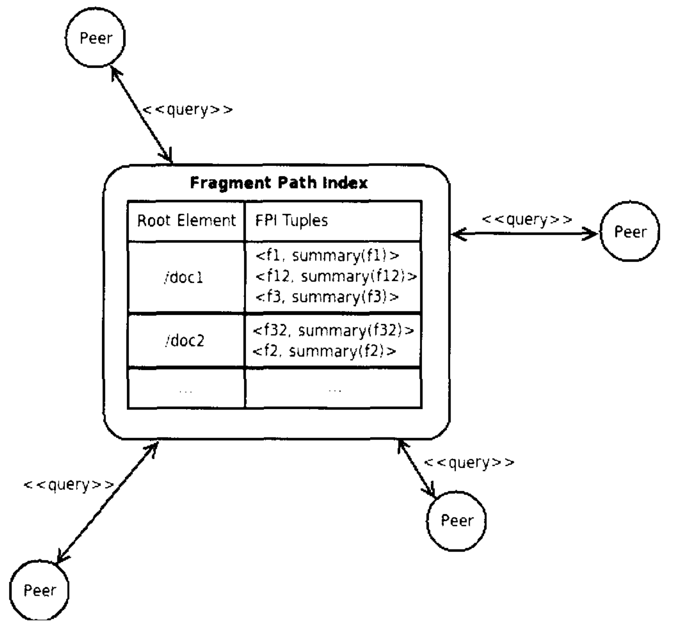
Figure 5.13: An example of a document update.

evaluation, the system could determine whether peers are likely to return a non-empty query result set.

5.6.2 The Fragment Path Index

In order to tackle the challenges described in section 5.6.1, we introduce another global lookup table into the framework, the *Fragment Path Index* (FPI). The FPI is a table that maps the root element name of an XML document to a list of $(fragmentId, fragmentSummary)$ tuples, where $fragmentId$ is the ID of a fragment extracted from a document with that root element, and $fragmentSummary$ is a field that summarises the structure and/or the content of the fragment. The $fragmentSummary$ allows the system to quickly determine whether a given path exists in a fragment without actually looking at the fragment itself. In our work, we do not prescribe how summary fields are calculated, only that such a mechanism be present. Possible XML summarisation techniques that may be used in conjunction with this work are described in [1, 16, 29, 53, 54, 68].

The FPI may be implemented as a completely separate entity in the system or as an extension of the Fragment Location Catalogue. Figure 5.14 shows a conceptual view of the FPI.



Root Element	FPI Tuples
/doc1	<f1, summary(f1)> <f12, summary(f12)> <f3, summary(f3)>
/doc2	<f32, summary(f32)> <f2, summary(f2)>
...	...

Figure 5.14: A conceptual view of the global Fragment Path Index (FPI)

5.6.3 Query Processing Algorithm

Using the FPI, the query processor may perform query evaluation as follows. When a peer receives a query of the form $/a/b/c/../n$, it retrieves the list of $(fragmentId, fragmentSummary)$ tuples corresponding to the root element name, a, from the FPI. This list is then pruned by eliminating all tuples corresponding to fragments that do not need to be traversed during query evaluation. This is achieved by comparing the query expression to the *fragmentSummary* field in each tuple. If, based on the *fragmentSummary* field, it is determined that the query will not be satisfied by the corresponding fragment, that tuple is removed.

Next, the locations of the fragments are determined by performing lookup operations on the Fragment Location Catalogue using the *fragmentId* fields in the tuples. The query is then sent to all peers returned by the Fragment Location Catalogue at the same time. The peers that receive the query, evaluate it on the fragments in their local data stores, and return the result to the querying peer. After all the results have been collected by the querying peer, or after a certain timeout has expired, the results are returned to the user.

Pseudocode for this query processor is presented in figure 5.15. To perform the actual query evaluation on locally stored XML fragments, any standard XPath query tool may be used.

```
procedure processQuery(queryExpression):
    rootElement := get root element name from queryExpression
    tupleList := get tuples from FPI that map to the root element

    // Prune the tuple list by removing tuples whose summaries
    // don't match the query expression.
    prunedTupleList := new list
    foreach tuple in tupleList do:
        if tuple.fragmentSummary.matches(queryExpression) do:
            prunedTupleList.add(tuple)
        end if
    end foreach

    // Send the query to the peers.
    peers = get peers that store fragments from FLC
    foreach peer in peers do:
        send queryExpression to peer
    end foreach

    results := wait until results arrive or timeout expires
    results := remove duplicates from results
    return results
end procedure
```

Figure 5.15: The query processing algorithm in pseudocode

5.6.4 Query Processor Limitations

The query processor has the following limitations:

- *The large size of the FPI tuple list.* The size of the tuple list returned by a lookup operation on the FPI can be very large, especially when there are many XML documents in the network with the same root element name. The larger the size of the tuple list, the greater the network cost when transferring this list to the querying peer.

  One way in which the size of the tuple list returned to the querying peer can be reduced is to perform the tuple list pruning operation within the FPI rather than at the querying peer. This requires extra processing within the FPI. However, the cost of this extra processing is outweighed by the cost of transferring large tuple lists.

- *No support for queries starting with a descendant axis.* The FPI uses XML document root element names as lookup keys. The querying peer determines these root element names by extracting it from the query expression. However, this forces queries to start with a child axis (e.g. $/a/b/.../z$). Queries starting with a descendant axis such as $//b/.../z$ cannot be evaluated, because the root element name is not specified in the query expression.

  A solution to this problem would be to retrieve all the tuples from the FPI. However, this approach is infeasible when the FPI has many entries.

- *The quality of the fragment summaries.* The query processor is also limited by how well the $fragmentSummary$ fields in the FPI tuples summarise XML fragments. If the $fragmentSummary$ incorrectly indicates that a fragment satisfies the query expression (i.e. a false positive), then unnecessary networking cost will be incurred when query messages are sent to peers that will simply return empty result sets.

5.7 Summary

This chapter introducted our P2P framework for replicated XML document fragments. This framework defines components and mechanisms for fragmenting XML documents for replication, locating XML fragments in a P2P network, handling XML fragment updates, as well as performing distributed XML query processing.

Each peer in the network stores a collection of whole XML documents, as well as a set of XML fragments it received from remote peers. An XML fragment is defined as either a subtree in an XML document or the whole XML document itself. When fragmenting XML documents, path expressions of the form $/p_1/p_2/.../p_n$ are used to identify subtrees, where p_i is an integer indicating the position of an element at level i in the document.

The Fragment Location Catalogue (FLC) is used to locate fragments distributed throughout the network. It acts as a global lookup table that maps the ID of an XML fragment to all the peers in the network storing that particular fragment.

The Fragmentation Table maintains information about the manner in which local XML documents at a peer have been fragmented. The Update Manager uses the Fragmentation Table to identify all the XML fragments that have been affected by an update, and returns a list of descriptions specifying how each updated fragment has changed.

The Fragment Path Index (FPI) is a global lookup table that maps the root element name of an XML document to a list of $(fragmentId, fragmentSummary)$ tuples. The Query Processor uses the FPI to determine where queries should be sent in the network. Query evaluation on XML fragments is performed using a standard XPath query processing tool.

Chapter 6

Replication Management and Update Propagation Algorithms

This chapter presents the replication and update propagation algorithms for the XML P2P framework proposed in chapter 5.

6.1 Replication Management

6.1.1 Overview

In this work, data replication is divided into two phases: replica selection and replica placement. During the replica selection phase, the system determines *what* XML fragments in the local data store should be replicated, *how many* copies of the selected fragments should be created in the network, and *when* replication should occur. During replica placement, the system identifies *where* in the network replicas should be placed. By splitting replication management into two phases, the selection and placement of replicas can function independently, making it possible to replace just a selection algorithm or just a placement algorithm.

This dissertation investigates three heuristic replica selection algorithms:

- Random Replica Selection (RRS)
- Most-Frequently Accessed Replica Selection (MFA)

- Most-Recently Accessed Replica Selection (MRA)

We also look at four heuristic replica placement algorithms:

- Random Replica Placement (RRP)
- Counter-based Replica Placement (CRP)
- Counter-State Replica Placement (CSRP)
- Average Uptime Replica Placement (AURP)

These replica selection and placement algorithms are described in sections 6.1.2 and 6.1.3, respectively.

In order to handle data replication, we define a *Replication Manager* component, consisting of three subcomponents: a *Replica Selector*, which implements the replica selection algorithm, a *Replica Placer*, which implements the replica placement algorithm, and a *Replicator*, which performs the transfer of replicas between peers in the network. Figure 6.1 shows how these components fit together.

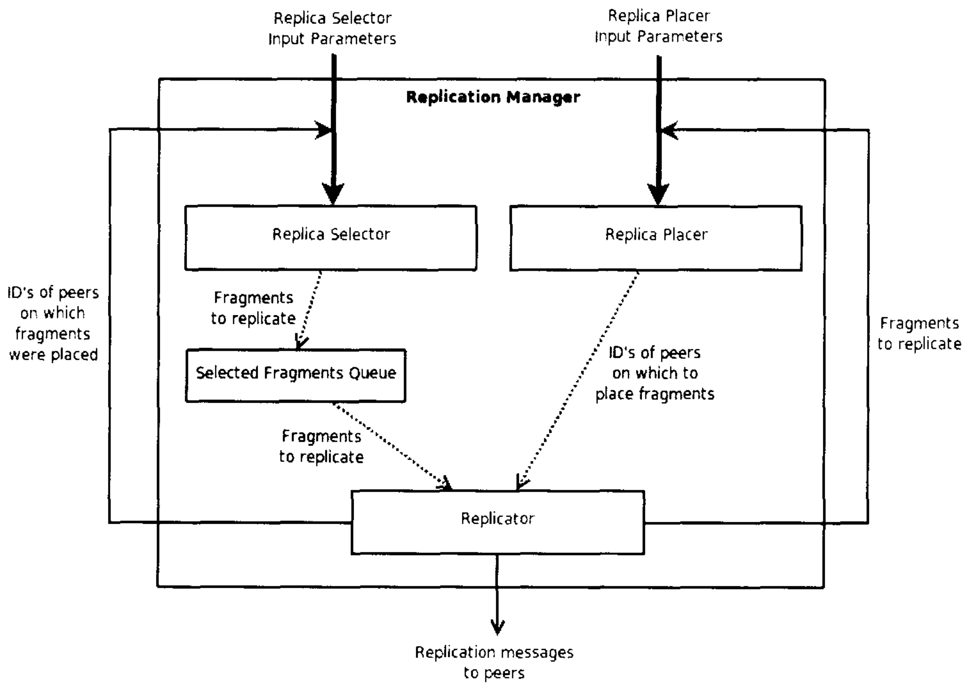

Replica Selector Input Parameters
Replica Placer Input Parameters
Replication Manager
Replica Selector
Replica Placer
ID's of peers on which fragments were placed
Fragments to replicate
Selected Fragments Queue
ID's of peers on which to place fragments
Fragments to replicate
Fragments to replicate
Replicator
Replication messages to peers

Figure 6.1: The design of the Replication Manager component

The Replication Manager subcomponents interact as follows. The Replica Selector identifies a set of fragments to replicate based on its input parameters, and places it on a queue shared between it and the Replicator. This triggers the Replicator to retrieve fragments from the queue and invoke the Replica Placer to generate a list of peers in the network to which the replicas should be sent. The Replicator then issues a *replication request message* to each peer identified by the Replica Placer, which, in turn, respond with either a *replica accept* or *replica deny message*. If a replica accept message is received, the Replicator establishes a connection with the peer in order to transfer the fragment. If a replica deny message is received, the replica transfer to that peer is cancelled. Peers may decide to deny a replica based on a number of factors, such as insufficient space, high load, etc.

The selection and placement of replicas may not always be independent. In some cases, the identification of peers on which replicas should be stored depends on what fragments were selected. For example, a replica strategy may require that certain fragments be sent to certain peers in the network. To accommodate this, the Replicator passes the fragments it obtained from the shared queue to the Replica Placer when the Replica Placer is invoked. This gives the Replica Placer the option of basing its placement decisions on the fragments selected.

The actions of the Replica Selector may also depend on the previous outputs of the Replica Placer. For instance, the Replica Selector may keep track of where fragments were sent so that it can re-queue fragments for replication when those peers depart the network. Therefore, after every successful replica transfer, the Replicator informs the Replica Selector where the replica was sent.

6.1.2 Replica Selection Algorithms

All the replica selection algorithms proposed in this dissertation work roughly as follows. During every replica selection period, t, a collection of XML fragments is selected for replication. For each selected fragment, the number of copies of the fragment to create is determined. The fragment is then queued for replication.

There are various ways in which the number of copies may be calculated. In this work, the maximum number of copies allowed in the network, m, is specified at system start-up. The number of copies, r, to create during a selection interval is

$$r = m - k$$

where k is the number of current copies determined using the Fragment Location Catalogue.

The replica selection process is outlined in pseudocode in figure 6.2. Each replica selection algorithm provides a different implementation of the `process()` procedure. The interval, t, between replica selection periods is a fixed parameter specified at start-up.

```
procedure selectReplicas():
    m := get maximum allowed replicas
    while stop != true do:
        selectedFragments := process()
        foreach fragment in selectedFragments do:
            k := FLC.lookup(fragment).length
            if k < m do:
                r := m - k
                queueForReplication(fragment, r)
            end if
        end foreach
        sleep(t)
    end while
end procedure
```

Figure 6.2: The basic structure of the replica selection algorithm

6.1.2.1 Random Replica Selection (RRS)

The Random Replica Selection (RRS) algorithm has two variants: one that only replicates entire XML documents (i.e. no fragmentation), and one that replicates both entire XML documents and XML fragments. We describe the former first.

Random Replica Selection Without Fragmentation (RRS-F)

During each replica selection interval, n documents are randomly selected from the data store, where n is a randomly generated integer between 1 and the total number of documents.

Random Replica Selection With Fragmentation (RRS+F)

This algorithm is an extension of RRS-F. Whenever a locally-owned XML document is selected for replication, a randomly selected subtree in the document is extracted and queued for replication.

The subtree to extract is selected by traversing the XML document to generate a list of all the possible path expressions from the root of the document to every other element in the document, and then randomly selecting a path expression from this list. This path expression along with the XML document are then passed to the Fragment Creator so that the subtree may be extracted. For example, consider the sample XML document in figure 6.3. The list of all possible path expressions is $\{/1, /1/1, /1/2, /1/2/1, /1/2/2\}$. Assume $/1/2$ is chosen from the list at random. Then, the subtree rooted at element c in figure 6.3 will be extracted from the document and replicated.

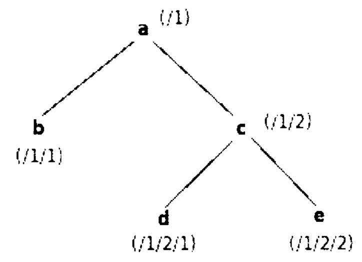


Figure 6.3: Generating the path expressions for each element in an XML document. The path expressions are shown in brackets next to each element.

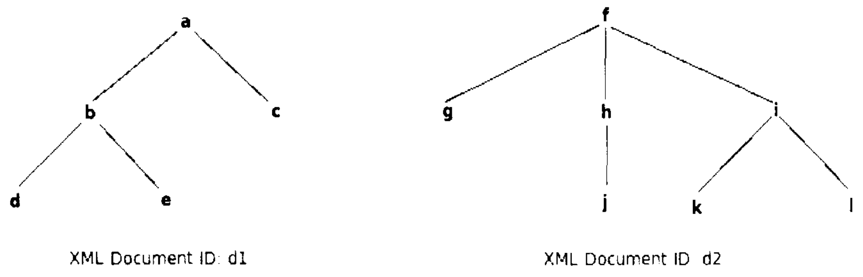
6.1.2.2 Most-Frequently Accessed Replica Selection (MFA)

The Most-Frequently Accessed Replica Selection (MFA) algorithm aims to increase the availability of those fragments that are in high demand in the network. It maintains a table of access information objects (AIOs), where each AIO stores the ID of the XML document that was accessed, the path expression of the subtree in the document that was accessed, and a counter indicating the number of times the subtree in the document was accessed. The access information table is sorted in descending order of access count.

During every selection interval, n XML fragments are selected for replication by obtaining the top n AIOs from the access information table, and retrieving the fragments specified from the data store. In this project, the value n is a fixed parameter configured by the user.

Like the RRS algorithm, the MFA algorithm has two variants: one that only selects whole XML documents for replication, and another that selects both whole XML documents and XML fragments extracted from documents. In the MFA algorithm without fragmentation (MFA-F), the path expression stored in the AIO will always represent the root of the XML document, regardless of what subtree in the document was accessed. There will thus always be one AIO in the access information table for each document accessed in the system. In the MFA algorithm with fragmentation (MFA+F), the path expression in the AIO is the reference to the actual subtree accessed in an XML document. Therefore, there may be many AIOs for a particular document in the access table, as the system needs to keep track of the access counts for each accessed subtree separately.

An example of the information stored in the access information table is given in figure 6.4. Assume that a peer stores the two XML documents in figure 6.4(a), and that queries evaluated at the peer access the elements in the following order: b, h, l, a, j, h. Each row in the access tables in figures 6.4(b) and 6.4(c) represents one AIO. Now, if the number of fragments to select, n, is 3, then documents $d1$ and $d2$ will be selected for replication when no fragmentation is performed, whereas subtrees $/1/2$ and $/1/3/2$ from document $d2$ and $/1/1$ from document $d1$ will be selected if fragmentation is enabled.



(a) Sample XML documents

XML Document ID	Path Expression	Access Count
d2	/1	4
d1	/1	2

(b) The access information table when no fragmentation is performed

XML Document ID	Path Expression	Access Count
d2	/1/2	2
d1	/1/1	1
d2	/1/3/2	1
d1	/1	1
d2	/1/2/1	1

(c) The access information table when fragmentation is enabled

Figure 6.4: An example of the information stored in the MFA algorithm's access information tables. The elements in figure 6.4(a) are accessed in the following order: b, h, l, a, j, h.

In order to determine what specific subtrees in the XML documents were accessed by queries, the Replica Selector interacts with the XPath Query Processor as follows. Whenever a query is evaluated on a locally stored XML document, the Query Processor passes the XPath query result, along with the ID of the document on which the query was evaluated, to the Replica Selector. XPath query results may either be expressed as a set of DOM tree nodes, a string, a number or a boolean value [98]. In this work, only queries that return DOM tree nodes are considered. Each of these DOM nodes is a root of a subtree accessed by the query. The Replica Selector first moves up the DOM tree from the DOM node in the result set to the root of the document. Then, it travels down the tree in reverse, calculating the position of each node along the path relative to its siblings.

For example, if node k in figure 6.4(a) is in the result set, then its path expression is calculated by first moving from k to the root f along the path $k - i - f$. Then, going down the DOM tree from f to k, the positions of f, i and k relative to their siblings in the DOM tree are calculated. These positions are 1, 3 and 1, respectively. The path expression for node k is thus $/1/3/1$. This path expression is then entered into the access information table.

6.1.2.3 Most-Recently Accessed Replica Selection (MRA)

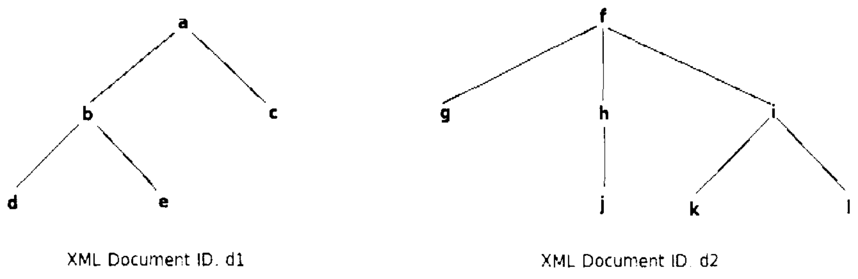
The MFA algorithm tries to ensure the availability of XML fragments that are in high demand. However, it may perform poorly when data request patterns change rapidly. The Most-Recently Accessed Replica Selection (MRA) algorithm, on the other hand, tries to adapt quickly to request pattern changes by selecting the most recently requested fragments for replication. It works exactly the same as the MFA algorithm, except that, instead of maintaining access counters, it moves AIOs to the top of the access table whenever the corresponding fragments are accessed in the system.

Using the same example from figure 6.4, the access information tables for the MRA algorithm will look like those shown in figure 6.5. If the number of fragments to select, n, is 3, then documents $d1$ and $d2$ will be selected for replication when no fragmentation is performed, while subtrees $/1/2$ and $/1/2/1$ from document $d2$ and $/1$ from document $d1$ will be selected when fragmentation is enabled.

6.1.3 Replica Placement Algorithms

The replica placement algorithms all take as input a list of peer IDs from which they select peers on which to place replicas. This list only contains the IDs of peers that are believed to be online. The replica placement algorithms thus depend on two other components in the system to operate: a *Peer Discovery* component that discovers new unseen peers in the network, and a *Peer Pinger* that periodically pings each discovered peer and notifies the Replica Placer whether peers are online or offline. Simple implementations of the Peer Discovery and Pinger components are described in section 7.9.

Each replica placement algorithm works roughly as follows. Whenever a fragment has been



(a) Sample XML documents

XML Document ID	Path Expression
d2	/1
d1	/1

(b) The access information table when no fragmentation is performed

XML Document ID	Path Expression
d2	/1/2
d2	/1/2/1
d1	/1
d2	/1/3/2
d1	/1/1

(c) The access information table when fragmentation is enabled

Figure 6.5: An example of the information stored in the MRA algorithm's access information tables. The elements in figure 6.5(a) are accessed in the following order: *b, h, l, a, j, h*.

selected for replication, the Replicator passes the fragment to the Replica Placer, along with a number, r, indicating the number of copies of the fragment to create in the network. The Replica Placer then selects r peers from the list of online peers according to some algorithm. If r is greater than the number of online peers, then all the peers in the online list are returned. If there are no peers that are believed to be online, then no peers are returned by the Replica Placer, and the replication process is cancelled. The algorithms for selecting peers from the online list is presented in the following sections.

6.1.3.1 Random Replica Placement (RRP)

The Random Replica Placement (RRP) algorithm is the simplest replica placement algorithm. The peers on which replicas should be placed are randomly selected from the list of online peers.

Peer	Peer Status										Counter Value
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	
a	1	1	1	1	1	1	0	1	1	1	8
b	0	0	0	1	1	0	0	0	0	0	0
c	1	1	0	0	0	0	1	1	1	1	4
d	1	0	1	1	1	1	1	0	1	1	6
e	1	1	0	1	1	1	0	1	0	0	2
f	0	1	0	1	0	1	0	0	0	1	1

Table 6.1: An example of how peer counters are calculated in the CRP algorithm. A 1 in the Peer Status column, indicates an online peer at ping interval t_i (where $1 \leq i \leq 10$); a 0 indicates an offline peer.

6.1.3.2 Counter-based Replica Placement (CRP)

The Counter-based Replica Placement (CRP) algorithm maintains a counter for each known peer in the network. Whenever a peer is detected as online by the Peer Pinger component, the counter for that peer is incremented. Whenever a peer is detected as offline, the counter is decremented. Peers with high counter values are selected by the CRP algorithm as destinations for replicas. By biasing its placement decisions toward such peers, the CRP algorithm tries to increase data availability by placing replicas on peers that are frequently online.

Table 6.1 demonstrates how the counter values are calculated for six example peers, a to f, after 10 time intervals, t_1 to t_{10}. During each time interval, t_i, the Peer Pinger notifies the CRP algorithm whether a peer is online or offline. An online peer is indicated by a 1 in the table, while an offline peer is indicated by a 0. The counter values for peers a to f after time interval t_{10} are 8, 0, 4, 6, 2 and 1, respectively.

When requested to identify r peers on which to place replicas, the CRP algorithm does not simply select the r peers with the highest counter values. This would result in high load on peers with high counter values, as the top r peers would always be selected as replica stores. Instead, to help balance load and increase the spread of replicas throughout the network, the CRP algorithm determines the median counter value, and then randomly selects r peers whose counters are greater than or equal to the median. Returning to the example in table 6.1, if 2 peers are to be selected, then any 2 peers in the set $\{c, d, a\}$ may be chosen, rather than just the top 2, a and d.

6.1.3.3 Counter-State Replica Placement (CSRP)

One drawback of the CRP algorithm is that it does not take the online-offline behaviour of the peers in the network into consideration. A peer that frequently enters and leaves the network

Peer	Peer Status										Counter Value	No. State Changes	Score
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}			
a	1	1	1	1	1	1	0	1	1	1	8	3	2.67
b	0	0	0	1	1	0	0	0	0	0	0	2	0
c	1	1	0	0	0	0	1	1	1	1	4	3	1.33
d	1	0	1	1	1	1	1	0	1	1	6	5	1.2
e	1	1	0	1	1	1	0	1	0	0	2	6	0.33
f	0	1	0	1	0	1	0	0	0	1	1	7	0.14

Table 6.2: An example of how peer scores are calculated in the CSRP algorithm

may at some point have the same counter value as another more stable peer that has just recently been discovered. Ideally, replica placement should be biased towards stable peers so that data availability can be maintained. The Counter-State Replica Placement (CSRP) algorithm attempts to overcome this drawback. It is an extension of the CRP algorithm which, in addition to maintaining counter values for peers, records the number of times the peers have changed state from offline to online or from online to offline. It then calculates a score for each peer by dividing the counter value by the number of state changes. The higher the score, the greater the chance that a peer will be chosen as a replica store. As in the CRP algorithm, the peers are chosen by randomly selecting those peers that have scores greater than or equal to the median score.

Table 6.2 shows a modified version of the example given in table 6.1. A state change occurs whenever the peer status goes from `1` to `0` or from `0` to `1`. A `1` in the table for time t_1 counts as a state change from `0` to `1`, as all peers are considered to be initially offline. Using the CSRP algorithm, peers a to f will have the scores 2.67, 0, 1.33, 1.2, 0.33 and 0.14, respectively. Note that peers a and c are now seen to be more stable compared to peer d, for example, which changes state more.

6.1.3.4 Average Uptime Replica Placement (AURP)

The Average Uptime Replica Placement (AURP) algorithm operates in the same manner as the CRP and CSRP algorithms. However, instead of maintaining counters or scores, it calculates the average uptime for each peer and uses that to make its placement decisions.

The average uptime of a peer is calculated by dividing the total time a peer is online by its number of online sessions. An online session is the period from the time a peer joins the network until the time a peer departs the network. The formula for calculating the average uptime is:

$$aveUptimeTime = \frac{\sum_{k=1}^{N}(t_j - t_i)}{N}$$

where t_i and t_j are the start and end times of the k^{th} online session, and N is the number of online sessions.

For instance, consider the diagram in figure 6.6 showing the timeline for a particular peer. The average uptime of that peer is calculated as follows:

$$aveUptimeTime = \frac{(t_2 - t_1) + (t_4 - t_3)}{2}$$

If the peer's 2 online sessions lasted for 5 and 3 time units, respectively, its average uptime will be 4 time units.

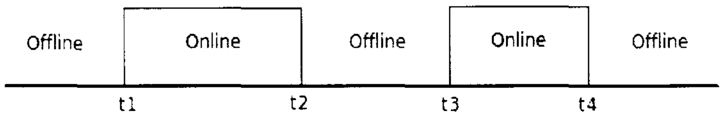


Figure 6.6: Calculating the average uptime of a peer

Table 6.3 shows the average uptimes for peers a to f from the previous examples in tables 6.1 and 6.2. If each block in the Peer Status column is equivalent to 1 time unit, then the uptimes for peers a to f will be 4.5, 2, 3, 2.67, 2 and 1 time unit(s), respectively. The median is 2.33 time units. Thus, peers a, c and d would be eligible for selection.

Peer	**Peer Status**										**Average Uptime**
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	
a	1	1	1	1	1	1	0	1	1	1	4.5
b	0	0	0	1	1	0	0	0	0	0	2
c	1	1	0	0	0	0	1	1	1	1	3
d	1	0	1	1	1	1	1	0	1	1	2.67
e	1	1	0	1	1	1	0	1	0	0	2
f	0	1	0	1	0	1	0	0	0	1	1

Table 6.3: An example of how the average uptimes of peers are calculated in the AURP algorithm. Each ping interval t_i (where $1 \leq i \leq 10$) in this example represents 1 time unit.

6.2 Update Propagation

After an XML document has been updated by the document owner as described in section 5.5, a "pull" mechanism is used to propagate updates to replicas. That is, all the peers holding replicas of fragments extracted from that document are notified by the owner peer when an update occurs. The replica holders retrieve the updates from the owner when needed. This is in contrast to a "push" model where updates are immediately sent to replicas whenever an update occurs. The rationale behind a pull mechanism is that it is unnecessary to send updates to peers if the replicas stored at those peers are accessed infrequently. By delaying the transfer of updates, networking overhead can be reduced.

Update propagation consists of two phases: an update notification phase and an update retrieval phase. These phases are discussed in the following sections.

6.2.1 Update Notification

When an XML fragment has been marked as updated by the Update Manager, the peers storing replicas of that fragment are looked up in the Fragment Location Catalogue. For each peer returned by the Fragment Location Catalogue, an update notification message is created and added to a queue. These update notification messages specify the ID and current version of the updated fragment. If the fragment was deleted during a document update, then instead of the version number, a flag indicating the deletion is specified.

The update notification messages are retrieved from the queue by an update notification process that sends the messages to the replica holders. When a replica holder receives an update notification, it either marks the replica in its data store as stale or deletes it, depending on whether the fragment was changed or deleted at the owner. It then responds to the owner peer with an update notification acknowledgement message. If this update notification acknowledgement is not received by the owner after a certain timeout period has expired, the replica holder is considered offline. The owner peer then re-adds the update notification message to the queue so that it may be re-sent at a later stage.

6.2.2 Update Retrieval

The update retrieval phase is initiated whenever a replica holder attempts to access a stale replica in its data store. It sends an update request message to the owner peer, specifying the ID and version number of the stale replica. Upon receiving the update request message, the owner establishes a connection with the replica holder, and sends it update descriptions that may be applied in sequence to the stale replica in order to produce the latest version of the fragment. These update descriptions are generated whenever an XML document is updated as described in section 5.5.2. Sending update descriptions rather than the actual updated fragment results in less network traffic. After the replica holder applies the update descriptions to its stale replica, the update retrieval process terminates.

If the owner peer does not establish a connection with the replica holder after a certain time has expired, the replica holder resends the update request message. If the owner still does not respond after a certain number of retries, it is considered offline. In this case, the replica holder attempts to retrieve a more up-to-date version of the fragment from another replica holder in the network. A version request message is sent to each peer returned by the Fragment Location Catalogue in order to determine what version each peer holds. An update request message is sent to the peer with the highest version number greater than that of the replica holder. The update retrieval process then proceeds in the same manner as before. If this update retrieval is unsuccessful, then the peer with the next highest version number greater than that of the replica holder is tried, and so on. If no peer with a higher version number is found, the Update Retrieval phase terminates.

Note that the owner peer will be included in the list of peers returned by the Fragment Location Catalogue. Therefore, if the owner peer rejoins the network shortly after the replica holder determined it to be offline, it will be contacted first, since it will have the highest version of the fragment.

To allow peers to retrieve updates from other replica holders, peers do not discard update descriptions after use. Instead, update descriptions are kept in the data store in case they are needed by other replica holders in the network. Since different replica holders may have different initial versions of an XML fragment, and thus hold different update description sequences, simply retrieving update descriptions from a replica holder may not always be sufficient for obtaining a later version of an XML fragment. In this case, an entire copy of an XML fragment is retrieved from a replica holder rather than just the update descriptions.

6.3 Summary

This chapter presented the replication and update propagation algorithms proposed in this dissertation.

Replication is divided into two sub-problems: replica selection and replica placement. Replica selection is performed by selecting n XML fragments for replication during every selection interval, t. The number of copies, r, to create of each fragment is calculated by subtracting the current number of copies in the network from the maximum number of copies, m, allowed in the network. The number of copies currently in the network is determined using the Fragment Location Catalogue. Random Replica Selection (RRS) selects n XML fragments to replicate at random. Most-frequently Accessed Replica Selection (MFA) maintains an access information table sorted in decreasing access count order, and selects the n fragments corresponding to the top n entries in the access information table. Most-recently Accessed Replica Selection (MRA) is similar to MFA, but moves access information table entries to the top of the table whenever the corresponding fragments are accessed.

Replica placement algorithms select r peers on which to place replicas from a list of online peers. Random Replica Placement (RRP) selects the r peers randomly. The Counter-based Replica Placement (CRP), Counter-State Replica Placement (CSRP) and Average Uptime Replica Placement (AURP) algorithms rank peers according to their online-offline behaviour, and randomly select the r peers whose ranks are greater than or equal to the median rank.

Update propagation is performed in two phases. During the update notification phase, replica holders are notified of XML fragment updates. During the update retrieval phase, replica holders retrieve and apply the updates.

Chapter 7

Prototype Implementation

This chapter discusses the development of a prototype XML P2P system that implements the framework presented in chapter 5. This prototype was used to evaluate the replication and update propagation algorithms described in chapter 6.

The prototype uses the simplest possible implementations for all framework components. For a production environment these components would be individually customized or replaced. The advantage of the framework is that each such component can be implemented and optimized independently.

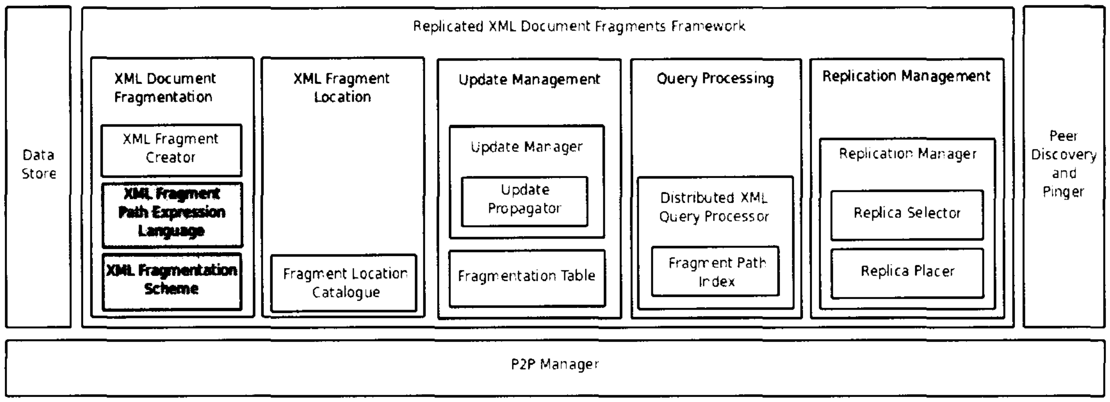


Figure 7.1: An overview of the components in the prototype

7.1 Overview

The prototype consists of nine components, as shown in figure 7.1. These are:

- A **P2P Manager** that handles the P2P network communications.
- An **XML Data Store** that manages the storage of XML documents and fragments.
- A **Fragment Creator** that extracts subtrees from XML documents.
- An implementation of the **Fragment Location Catalogue** proposed in section 5.4.
- An implementation of the **Fragmentation Table** in described in section 5.5.1.
- An **Update Manager** that handles document updates and performs update propagation.
- A **Query Processor** for evaluating simple XPath expressions. This component also includes an implementation of the Fragment Path Index.
- **Peer Discovery** and **Pinger** components used by the replica placement algorithms.
- A **Replication Manager** that is responsible for replicating data throughout the network.

Each component is described in the sections that follow. The prototype was implemented in Java to take advantage of the functionality provided by the J2SE [77] development framework and various widely available open-source Java-based toolkits.

7.2 The P2P Manager

The XML P2P prototype requires the following P2P network functionality:

1. Peers in the network need to be addressed using globally unique IDs. These IDs should be persistent and independent of the peer IP addresses to allow peers to be addressed in a consistent manner even when they depart and rejoin the network or when their IP addresses change.
2. Messages in the network should be routed between peers using the peer ID as the destination, rather than the IP address.
3. Higher level components in the system need to be informed whenever messages are sent and received, and when connections to remote peers are established and lost. This will enable system components to perform any application-specific processing when such events occur.

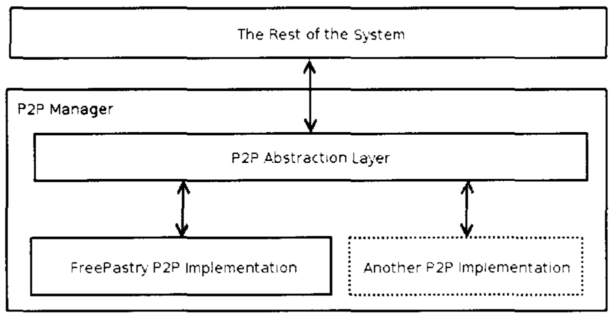


Figure 7.2: The design of the P2P Manager component

Creating such a P2P network implementation from scratch would take too much effort and was beyond the scope of our work. Therefore, we decided to use FreePastry [72] for our prototype. FreePastry is an open-source implementation of Pastry [61], a structured P2P routing protocol. It meets the aforementioned requirements as follows.

1. Each peer in FreePastry is assigned a globally unique ID that is generated using some application-specific algorithm.

2. A message in FreePastry is routed to a peer with an ID that is numerically closest to a given ID. This allows a message to be sent using the peer ID as a destination, rather than an IP address.

3. FreePastry informs higher-level system components via callback methods whenever messages at a peer are received and when neighbouring peers enter and leave the network.

In our prototype, peer IDs in the FreePastry network are generated using a pseudorandom number generator that produces random arrays of 20 bytes in length. This is done whenever a peer joins the network for the first time. After an ID has been generated for a peer, it is saved in the system, so that it may be reused whenever a peer leaves and rejoins the network at a later stage. In this way, peers retain their IDs across network sessions.

To allow for greater flexibility, an abstraction layer was built on top of FreePastry. This makes it possible to replace FreePastry with another P2P network implementation if a more efficient and reliable implementation is required at a later stage. This abstraction layer is illustrated in figure 7.2. It consists of interfaces that provide operations common to all P2P network implementations. These interfaces are described in detail in section A.1 of Appendix A with an accompanying UML class diagram.

7.3 The XML Data Store

The XML Data Store component manages the storage of XML data at a peer. For simplicity, the implementation in this prototype stores data directly on the file system. However, as in the case of the P2P Manager component, this file system-based implementation is hidden behind an abstraction layer, making it possible to replace this implementation with a more efficient version at a later stage.

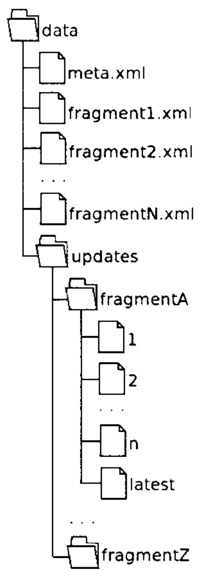


Figure 7.3: The directory structure of the XML data store

Figure 7.3 shows the structure of the directory in which the XML data are stored. All documents, whether local documents or fragment replicas, are kept in one top-level `data` directory as XML files. These XML files are named according to the ID of the XML fragments stored within them. For example, if the ID of a fragment (in hexadecimal format) is `1B00D24A4710A0351FDC9F2B5ED99E32C8CF65B9`, then that fragment will be stored in a file called `1B00D24A4710A0351FDC9F2B5ED99E32C8CF65B9.xml` in the `data` directory.

In addition to the XML files, a `meta.xml` file is maintained that stores the following meta information about the XML fragments in the `data` directory:

- The XML fragment ID encoded in base64 [85].
- The name of the XML fragment. This is a user-specified string that is used to conveniently refer to an XML fragment in the system, rather than using to the XML fragment ID.

For example, an XML document containing contact information might have the name, "address book".

- The name of the XML file in which the fragment is stored relative to the top-level `data` directory.
- The ID of the peer that owns the XML fragment. This is encoded in base64. For local XML documents, the value of this field is the ID of the local peer.
- The name of the XML document root element. This field allows the system to retrieve XML fragments from the data store based on the root element name when performing query processing. Note that this is not the name of the XML fragment root element, but the root element of the XML document from which the XML fragment was extracted.
- A flag indicating whether the XML fragment is an entire XML document or a subtree.
- The XML fragment version number. This value starts at 0, and is increased each time the XML fragment is updated.
- A flag indicating whether the XML fragment is the latest version. This is used by the Update Manager component when marking fragment replicas as stale during update propagation. For local XML documents, this flag will always be `true`.

An example `meta.xml` file is shown in figure 7.4.

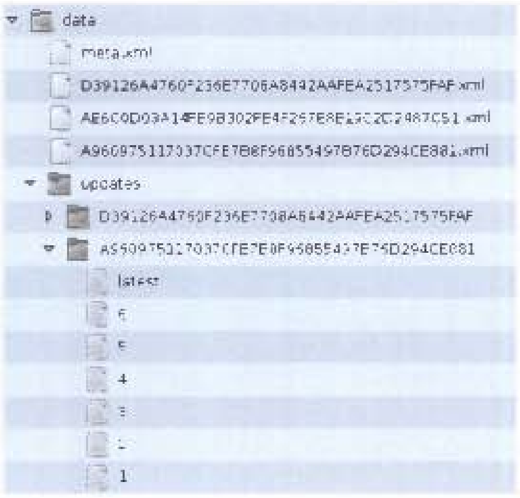
When performing document updates, the system needs to keep track of all the previous versions of an XML fragment so that replica holders may retrieve the most up-to-date version during update propagation. This is done by storing at the peer which owns the document, all the deltas (i.e. update descriptions) that may be applied to older versions in order to construct the current version. These deltas are stored in the `updates` directory below the top-level `data` directory. Within the `updates` directory, there is a directory for each fragment that was updated. These directories store collections of delta files, where each delta file is given an integer filename corresponding to the version of the fragment which it produces when applied to the previous version of the fragment. The delta filenames range from 1 to n, where n is the current version of a fragment. If a peer requesting an up-to-date version of an XML fragment holds version i of that fragment, then deltas $(i + 1)$ to n would be retrieved from the data store and sent to the peer.

In addition to the delta files, a file called `latest` is also stored in the `updates` directory for each fragment. This file contains a single integer indicating the current version number.

Figure 7.5 shows the contents of an example data store directory.

```
<fragments>
  <fragment id="GwDSSkcQoDUf3J5rXtmeMsjPZbk="
            name="shop catalog"
            filename="1B00D24A4710A0351EDC9E2B5ED99E32C8.xml"
            owner="jO2CYRXkbkn3CYLc8qyEakttA6E="
            root="catalog"
            isDocumentRoot "true" />
            version="0"
            isLatestVersion="true" />
  <fragment id="g8vhfXoBe55B8cWYcQHSKlnA5Z8="
            name="testament"
            filename="83CBB1709A017B9B41B1C5987101D22A29.xml"
            owner="Vq5MPzlXmCxDDRnOrZH.AcVbLyU="
            root="tstmt"
            isDocumentRoot="false" />
            version "2"
            isLatestVersion "false" />
</fragments>
```

Figure 7.4: An example meta.xml file in the XML data store directory



```
data
    meta.xml
    D39126A4760F236E7706A8442AAFEA2517575FAF.xml
    AE6C0D03A14FE9B302FE4F297E8E19C2C7487C51.xml
    A9609751170337CFE7B8F968554978 76D294CE881.xml
    updates
        D39126A4760F236E7708A6442AAFEA2517575FAF
        A9609751170337CFE7E8F968554978 75D294CE081
            latest
            6
            5
            4
            3
            2
            1
```

Figure 7.5: The contents of an example XML data store directory

7.4 The Fragment Creator

The Fragment Creator is responsible for extracting fragments from XML documents. When given a path expression in the form $/p_1/p_2/.../p_n$, where p_i denotes the position of an element in the document tree relative to its siblings, it traverses a given XML document, and writes the subtree identified by the path expression to a temporary file. This is done using a SAX XML parser, so that fragment extraction can be done efficiently and on-the-fly.

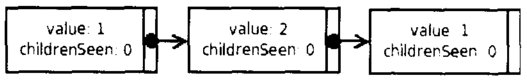


Figure 7.6: The path expression, /1/2/1, represented as a list of `PathStep` objects

The Fragment Creator accepts three input arguments: the XML document from which the fragment should be extracted, the path expression of the subtree in the XML document to extract, and the ID that should be assigned to the extracted fragment.

When the Fragment Creator is invoked, it parses the path expression, and loads it into memory as a `PathExpr` object. This `PathExpr` object contains a list of `PathStep` objects, where each `PathStep` represents p_i in the path expression. The `value` field in the `PathStep` object is the position of the element in the XML document tree relative to its siblings. The `childrenSeen` field indicates the number of child elements that have been processed. Figure 7.6 shows how the path expression, /1/2/1, would be represented as a list of `PathStep` objects. The `PathStep` at the end of the list is known as the *current step*.

During the fragment extraction process, the Fragment Creator maintains two path expressions: the target expression and the current expression. The target expression is the path expression of the subtree in the XML document that should be extracted, while current expression is the path expression of the element that is currently being processed in the XML document. The current expression's `PathStep` list is calculated as follows.

- Whenever a start tag is encountered in the document, a new `PathStep` object is created. If the `PathStep` list is empty, the `value` field of the newly created `PathStep` object is set to 1. Otherwise, it is set to the value of the current step's `childrenSeen` field plus 1. In both cases, the `childrenSeen` field of the new `PathStep` object is set to 0. After the new `PathStep` has been initialised, it is pushed onto the `PathStep` list. The resulting list of `PathStep` objects then represents the path expression of the element whose start tag was just reached.
- Whenever an end tag is encountered in the document, the last `PathStep` in the list is popped, and the `childrenSeen` field of the current step is incremented.

In order to determine whether the element currently being processed is part of the subtree that should be extracted from the document, the Fragment Creator checks whether the target expression is a prefix of the current expression or if the current expression is a prefix of the target expression. If either condition is true, the current element is outputted to a temporary file that stores the extracted fragment.

The fragment extraction process is best explained with an example. Consider the sample XML document in figure 7.7(a). If the path expression of the fragment to extract is $/1/2/1$, then the subtree rooted at element e should be extracted. Figure 7.7(c) shows how the Fragment Creator processes the XML document:

- When the a element's start tag is encountered, the current expression's `PathStep` list is empty. A new `PathStep` object is created with `value` of 1 and a `childrenSeen` value of 0. The `PathStep` list now represents the path expression, $/1$. Since $/1$ is a prefix of the target path expression, $/1/2/1$, element a is outputted.

- When element b is reached, a new `PathStep` object is created. Since the `PathStep` list is not empty, the `value` of the new `PathStep` is set to the value of the current step's `childrenSeen` field plus 1. The current step's `childrenSeen` value is 0, so the `value` of the new `PathStep` is set to 1. The new `PathStep` is pushed onto the list, resulting in the path expression, $/1/1$. Since $/1/1$ is not a prefix of the target expression, $/1/2/1$, and since the target expression is not a prefix of $/1/1$, element b is not outputted.

- When the XML parser exits from element b, the `PathStep` at the end of the current expression's `PathStep` list is popped. The `childrenSeen` field of the current step is then incremented to 1.

- At the start of element c, a new `PathStep` is created. Since the value of the current step's `childrenSeen` field is 1, the `value` of the new `PathStep` is set to 2. The resulting path expression is $/1/2$, which is a prefix of the target expression, $/1/2/1$. Therefore, element c is outputted.

- At the start of element e, the current expression is the path expression $/1/2/1$, which is a prefix of $/1/2/1$, so element e is outputted.

- At element g, the path expression is $/1/2/1/1$. This is not a prefix of the target expression $/1/2/1$. However, the target expression is a prefix of $/1/2/1/1$. Therefore, g is outputted.

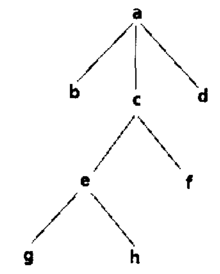
- Element h is also outputted, as the target expression, $/1/2/1$, is a prefix of element g's path expression, $/1/2/1/2$.

- When the XML parser reaches element f, the `PathStep` list represents the path expression, $/1/2/2$. Since this is not a prefix of the target expression, and since the target expression is not a prefix of $/1/2/2$, element f is ignored.

- Element d is also ignored, since its path expression is $/1/3$.

Once the XML document has been traversed, the fragment produced by the Fragment Creator has the structure shown in figure 7.7(b). Notice how elements b, f and d are omitted. Also notice how element e is not the root of the extracted fragment, but is enclosed within its ancestor elements, c and a. This is to allow the Query Processor to evaluate queries directly on the extracted fragment. If element e was not enclosed by c and a, then evaluating a query such as $/a/c/e/g$ on the extracted fragment would return no result.

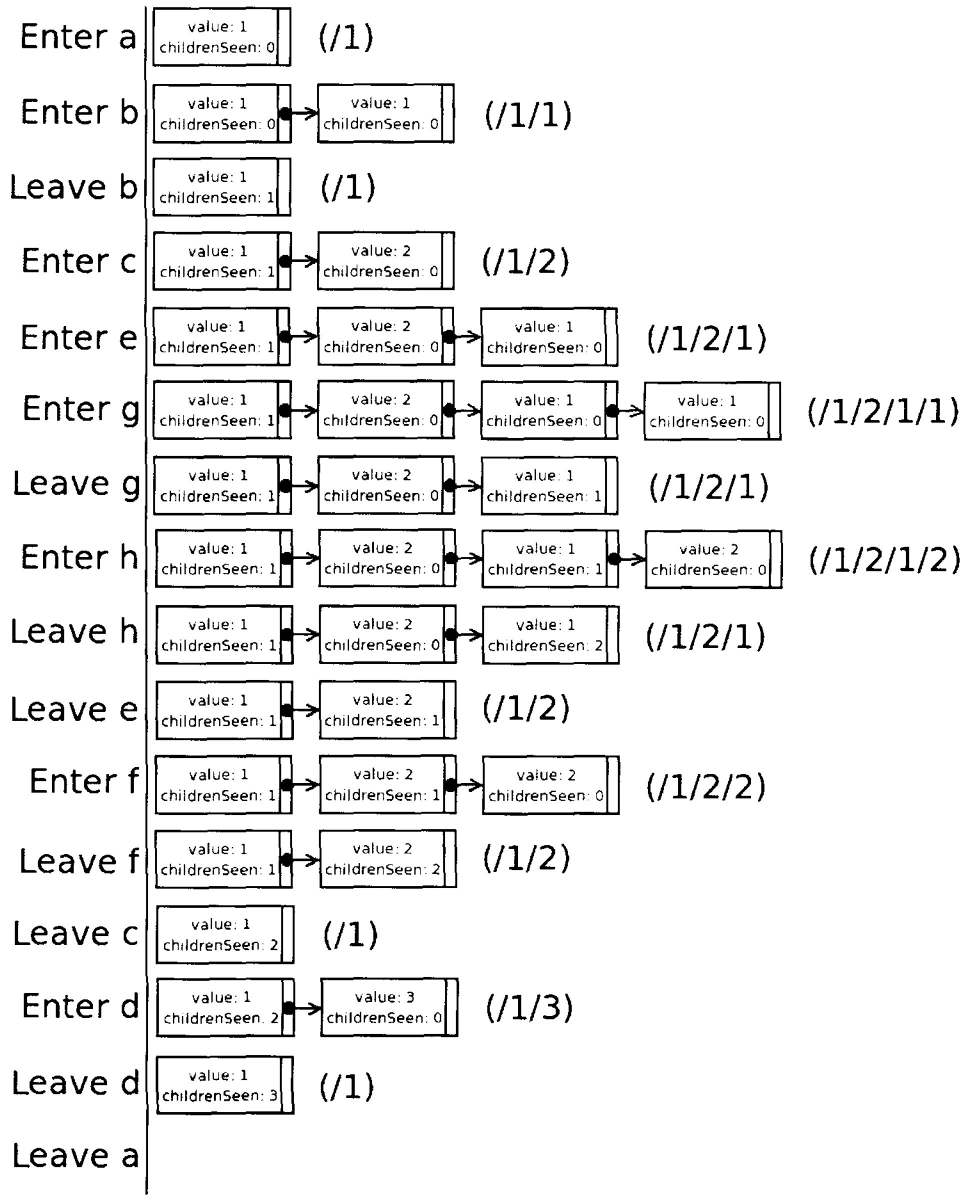
```
<a>
  <b/>
  <c>
    <e>
      <g/>
      <h/>
    </e>
    <f/>
  </c>
  <d/>
</a>
```



(a) A sample XML document

```
<a>
  <c>
    <e>
      <g/>
      <h/>
    </e>
  </c>
</a>
```

(b) The extracted fragment with path /1/2/1



(c) The contents of the current expression's `PathStep` list as the sample XML document is parsed

Figure 7.7: An example of how the Fragment Creator processes an XML document

7.5 The Fragment Location Catalogue

The Fragment Location Catalogue (FLC) component allows the system to determine the location of XML fragments in the network by mapping XML fragment IDs to peer ID lists. As mentioned in section 5.4, the FLC may be implemented as either a separate web service or on top of a P2P network using, for example, a DHT.

The FreePastry library that the prototype uses includes an implementation of Past [62], a DHT that uses the Pastry P2P routing protocol. Our first attempt at implementing the FLC used this Past implementation. However, it was soon discovered that the Past implementation does not handle the mapping of multiple peer IDs to one fragment ID correctly. Another approach would have been to implement our own DHT on top of FreePastry. However, this is beyond the scope of our work. Therefore, for this prototype, the FLC was implemented using a simple relational database that is globally accessible to all peers in the network. An abstraction layer hides the actual implementation from the rest of the system behind a generic interface. As a result, the system does not know whether the FLC is implemented as a web service, DHT or relational database, and thus, an alternative implementation can easily be accommodated.

Figure A.3 in Appendix A shows a UML diagram of the FLC implementation.

7.6 The Fragmentation Table

The Fragmentation Table is maintained as a simple XML file that closely follows the structure shown in figure 5.8. An example of this XML file for the sample document in Figure 7.8 is presented in figure 7.9.

A `<fragment>` element represents a Fragment Information Node. Its attributes, `id`, `document` and `path`, store the fragment ID, the ID of the XML document from which the fragment was extracted, and the path expression identifying the fragment in the XML document, respectively. Each `<fragment>` element may also optionally have `<descendants>` and `<ancestors>` child elements.

A `<document>` element represents a Document Information Node, and consists of a list of `<fragment>` elements specifying the fragments extracted from a document.

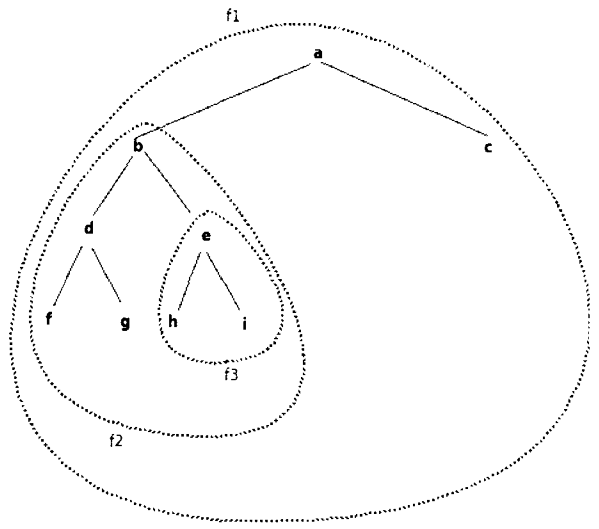


Figure 7.8: A sample XML document and its fragments

```
<fragmentationTable>
        <fragments>
                <fragment id="f1" document="d1" path="/1">
                        <descendants>
                                <descendant id="f2"/>
                                <descendant id="f3"/>
                        </descendants>
                </fragment>
                <fragment id="f2" document="d1" path="/1/1">
                        <ancestors>
                                <ancestor id="f1"/>
                        </ancestors>
                        <descendants>
                                <descendant id="f3"/>
                        </descendants>
                </fragment>
                <fragment id="f3" document="d1" path="/1/1/2">
                        <ancestors>
                                <ancestor id="f1"/>
                                <ancestor id="f2"/>
                        </ancestors>
                </fragment>
        </fragments>
        <documents>
                <document id="d1">
                        <fragment id="f1"/>
                        <fragment id="f2"/>
                        <fragment id="f3"/>
                </document>
        </documents>
</fragmentationTable>
```

Figure 7.9: The Fragmentation Table XML file for the sample document in figure 7.8

7.7 The Update Manager

The Update Manager processes XML document updates and performs update propagation as described in sections 5.5 and 6.2, respectively. It is implemented as an extension of the XML Data Store component using the Decorator design pattern [31]. In this way, update management is performed transparently to higher-level system components. Whenever a higher-level system component invokes an *update*, *retrieval* or *deletion* operation on the XML Data Store, the Update Manager intercepts the invocation, performs any necessary update management tasks, and then passes control to the XML Data Store to complete the operation. Requests for all other data store operations simply pass straight through the Update Manager to the underlying XML Data Store. Figure A.4 in Appendix A shows a UML class diagram of the Update Manager.

Since detecting document updates and handling update descriptions is not the focus of this work, the prototype simulates this functionality. Whenever an update is performed, the IDs of all the fragments extracted from a given document are retrieved from the Fragmentation Table. Then, for each fragment ID, if a pseudorandom number generator returns a value greater than 0.9, a delta for the corresponding fragment is created. This delta does not actually describe how a fragment was updated. It merely indicates a version number increase.

When transferring updates to remote peers, the Update Manager does not use the P2P layer (and thus the FreePastry library). Instead, it establishes direct TCP connections to remote peers outside the P2P network. This is done for the following reasons:

1. *FreePastry uses Java-object serialisation to transfer messages between peers* [73]. A message is represented as a Java object (the `Message` interface in figure A.1 in Appendix A) that is converted to a stream of bytes before it is routed through the network. When the message is received at the destination, it is reconstructed from the stream of bytes into a Java object. This means that in order to send an update to a remote peer, the entire update would first need to be loaded into memory and stored within a `Message` object before it can be sent. If the update is larger than the amount of memory available, the system would be unable to send updates.

   This problem could be resolved by breaking the update into smaller chunks and sending each chunk individually. However, this would complicate update transfers, as additional measures would be required to ensure that all chunks arrive at the destination successfully and in the right order.

2. *FreePastry uses a single connection to transfer Java-object messages* [73]. This connec-

tion is used to transfer both protocol maintenance and application-level messages. Sending large `Message` objects would thus stall other messages waiting to be sent, resulting in poor network performance.

The data sent via the direct connection are depicted in figure 7.10.

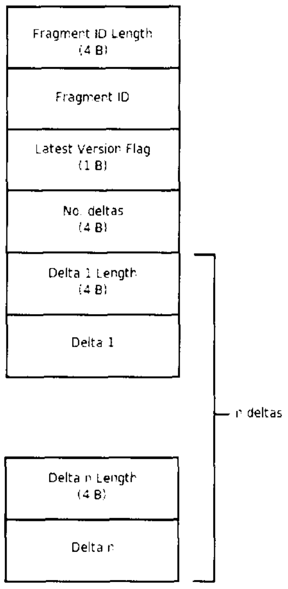


Figure 7.10: The data sent to remote peers when transferring updates via a direct connection

7.8 The Query Processor

The Query Processor consists of three parts:

- The actual query processor that evaluates queries on XML fragments stored in the XML data store;
- A network interface that allows the Query Processor to receive queries from remote peers for evaluation; and
- An implementation of the Fragment Path Index (FPI) described in section 5.6.2.

A UML class diagram of the Query Processor is shown in figure A.5 in Appendix A.

The Query Processor first initiates the processing of queries on locally stored XML fragments using the standard Java XPath query processor. Then, it sends queries to remote peers for evaluation. Since evaluating queries on the network may take an arbitrary amount of time, a callback interface is provided to enable non-blocking query processing.

A listener interface is also provided by the prototype. This interface takes three input arguments: the query string, the XML fragment on which the query was evaluated, and the query result. Unlike the callback interface, which is called only when a specific query was processed, the listener is called whenever *any* query was processed. This allows the Most-Frequently Accessed and Most-Recently Accessed Replica Selection algorithms (see sections 6.1.2.2 and 6.1.2.3) to hook into the Query Processor component, so that they may determine what specific subtrees in local XML fragments were accessed.

The FPI is used by the Query Processor to determine what fragments in the network a particular query should be evaluated on. It maps the names of XML document root elements hashed using SHA-1 to $(fragmentId, fragmentSummary)$ tuples, where $fragmentId$ is the ID of a XML fragment in the network and $fragmentSummary$ is a data structure that summarises the structure and/or content of the XML fragment. Since calculating XML fragment summaries is beyond the scope of this work, the prototype uses dummy $fragmentSummary$ objects that always return boolean true when checking whether a particular path expression exists within the corresponding fragments.

The UML class diagram for this prototype's FPI implementation is given in figure A.6 in Appendix A. The design is identical to that of the FLC. An abstraction layer hides the actual implementation from the rest of the system behind generic interfaces, so that higher-level components do not know whether the FPI is implemented as a web service, DHT or using a relational database. The prototype stores $(fragmentId, fragmentSummary)$ tuples in a relational database that is globally accessible to all peers in the network. Like the FLC, this is the simplest way of abstracting over this implementation detail.

7.9 The Peer Discovery and Pinger Components

The replica placement algorithms described in section 6.1.3 require a list of online peers from which to select destinations for replicas. In order to construct this list, a mechanism is needed that will search the network for new unseen peers. The Peer Discovery component provides this service.

Whenever the Peer Discovery component discovers unseen peers in the network, it informs the rest of the system about the discovered peers. Since the implementation of a peer discovery mechanism may be specific to a particular P2P network protocol, and since such an implementation is beyond the scope of this work, the prototype instead uses a mock implementation. Before an experiment is conducted with the prototype, the IDs of all the peers in the network are generated and added to a globally accessible relational database. Then, at system start-up, the a list of peer IDs is retrieved from the database and handed to the rest of the system.

Once the system knows what peers are in the network, it needs to periodically probe the peers to determine whether they are online or offline. This responsibility lies with the Peer Pinger component. The Peer Pinger maintains a list of all the peers discovered by the Peer Discovery component. During every probe period, it iterates through the list of peers, sending each peer in the list a `Ping` message. If the Peer Pinger receives a `Pong` response, that peer is considered online. If no `Pong` is received within a particular time period, the peer is considered offline.

7.10 The Replication Manager

The Replication Manager implements the replication algorithms described in section 6.1. A UML class diagram for it is presented in figure A.8 in Appendix A, along with a description of how the various classes and interfaces interact.

Like the Update Manager component, the Replication Manager does not transfer replicas to remote peers using the P2P layer, as XML fragments may be too large to load into memory as Java objects. Instead, it establishes direct TCP connections to remote peers outside the P2P network, and transfers replica messages in the format shown in figure 7.11. To help reduce networking cost when sending replicas, XML fragments are compressed using GZIP [83].

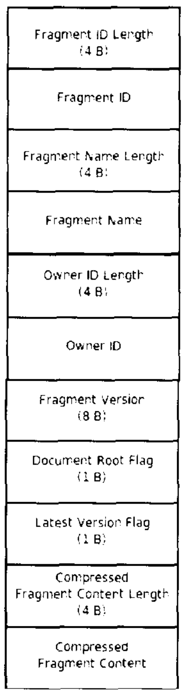


Figure 7.11: The data sent to remote peers when transferring replicas via a direct connection

7.11 Summary

This chapter describes a prototype P2P XML data management system that implements the framework presented in chapter 5 and the replication and update propagation algorithms discussed in chapter 6.

The FreePastry library was used to provide the P2P functionality. Appropriate interfaces for all components in the framework were developed to abstract over implementation details and to identify the minimum functionality required by each component. The simplest approach was then used for each implementation, resulting in a complete system that is easy to enhance and experiment with.

The prototype described in this chapter was used during the project experiments to test the replication and update propagation algorithms. These experiments are discussed in the following chapter.

Chapter 8

Experimental Evaluation

This chapter describes evaluation of the data replication and update propagation algorithms proposed in chapter 6. The data replication algorithms were evaluated by considering different combinations of replica selection and placement algorithms to see which performed best under two different network conditions. The update propagation algorithm was evaluated by measuring how often queries returned up-to-date results.

The experimental setup and method are first described. Then, the results obtained during the experiments are presented and analysed.

8.1 Experimental Setup

8.1.1 Network Environment

Experiments were performed on a cluster consisting of 13 identical computers interconnected via gigabit ethernet links. Each computer had a 3GHz Intel Pentium 4 processor with a cache size of 1MB, 512MB of main memory, and approximately 1GB of swap space.

ModelNet [80] was installed on the cluster. ModelNet is a wide-area network emulator that enables the evaluation of unmodified network applications in realistic large-scale Internet-like networking scenarios. ModelNet creates and runs multiple instances of a networking application on each cluster machine, where each instance represents a virtual node (i.e. a peer) in the ModelNet network.

ModelNet requires 2 types of physical computers to operate: emulator machines and host machines. Host machines run the virtual nodes. Emulator machines subject network packets to delays, losses, queueing, congestion and bandwidth constraints according to some network topology specification. This emulation occurs transparently to the application and in real-time, giving the application the illusion that it is participating in a real wide-area distributed environment. The application itself is completely unaware of the existence of ModelNet.

For this project, 12 cluster machines were set aside as ModelNet host machines, while 1 acted as a ModelNet emulator. All the host machines ran identical Linux installations, while FreeBSD 4.11 was used for the emulator machine. To obtain the best possible emulation results, the emulator's operating system kernel was configured with a clock rate of 10,000Hz, as recommended by the ModelNet documentation [79]. In addition to these 13 machines, another computer was used to host a relational database storing FLC and FPI mappings, as mentioned in sections 7.5 and 7.8, respectively. A diagram of this setup is shown in figure 8.1.

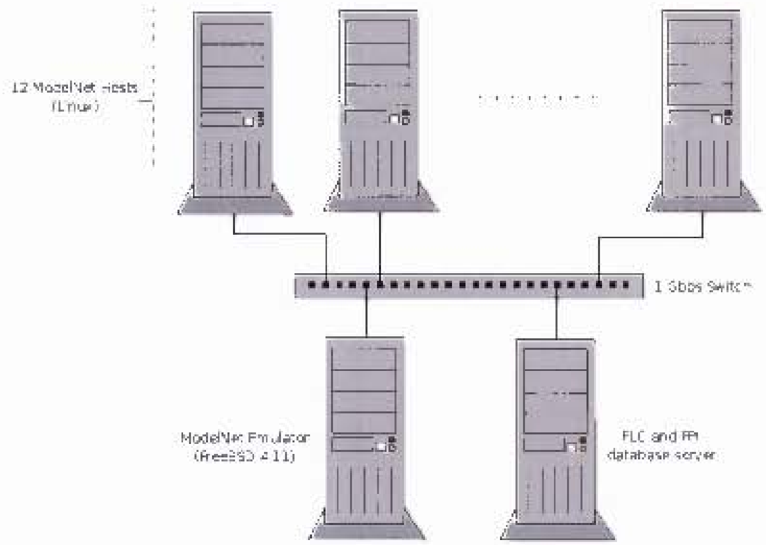


Figure 8.1: The experimental network setup

In all the experiments, the same ModelNet parameters were used. A 5000 node Autonomous System (AS) level network with 100 clients (peers) and 25 stubs was generated using the Inet Topology Generator [76] and ModelNet's `inet2xml` tool. The client-stub link bandwidth was set to 10Mbps, while stub-stub, stub-transit and transit-transit links were set to 10Gbps. All links in the network were assigned a default ModelNet latency and queue length. The link delays were automatically generated based on node distance.

8.1.2 Test Documents

A collection of 95 sample XML documents was used. Some of these documents were synthetically generated, while others were extracts of documents taken from [81, 82, 89, 90, 94]. A histogram showing the document size distribution is presented in figure 8.2. The smallest document was 515 B in size, while the largest document was 1491.61 KB. The test documents were kept intentionally small, as initial runs of the experiments resulted in out-of-memory errors when performing query processing on large XML documents. These out-of-memory issues were caused by running up to 9 instances of the prototype on each ModelNet host machine concurrently, which put increasing strain on system resources as the XML document sizes increased. Since the experiments did not evaluate the efficiency of the replication and update propagation algorithms, using small XML documents was deemed sufficient.

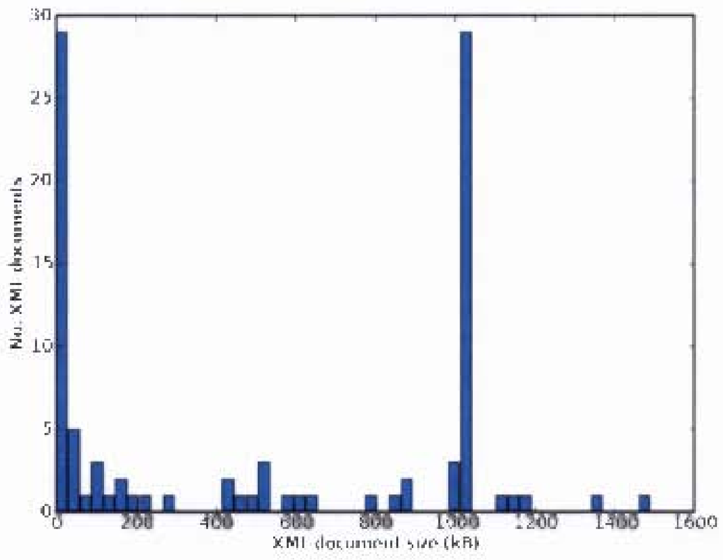


Figure 8.2: The file size distribution of the test XML documents

Peers were assigned XML documents from the sample collection in a round-robin fashion, such that, at the start of each experiment, 95 peers in the network each had one XML document in their data stores, while the remaining 5 peers had none. The same peers were always assigned the same XML documents before each experiment.

8.1.3 Test Queries

During each experiment, a set of queries was posed on the system. These queries were generated from the collection of sample XML documents, such that each query returned a non-empty result set and its results were known before-hand.

The query generation process is outlined in figure B.1 of Appendix B. For each XML document in the sample collection, a set of queries was generated. Each query in this set was evaluated on the XML document. If the queries returned non-empty result sets, they were added to a list of candidate queries. Once all the documents had been processed, a query script was generated from the list of candidate queries. This query script was executed by the prototype to pose queries on the system during the experiments. It specified the time a query should be evaluated, the query that should be evaluated and the peer in the network that should submit the query for evaluation. The queries in the script were selected at random from the candidate query list. The time between the queries was generated using a Poisson process.

The query script consisted of 300 queries, posed at an average of 3 queries per minute. The total run-time of the script (i.e. the time required to execute all 300 queries per experiment) was therefore approximately 1 hour 40 minutes. A single peer that was guaranteed to be online during the experiments (the boot peer) was chosen as the query submitter for all queries.

8.1.4 Peer Online-Offline Behaviour

The algorithms were tested using two peer online-offline behavioural models: a real-world model based an existing P2P network and a synthetically generated model using a Poisson process.

The real-world model was obtained by recording the times peers entered and departed a Direct-Connect P2P network. DirectConnect is a hybrid decentralised P2P filesharing system similar to Napster, in which peers connect to a central hub that accepts and evaluates queries on behalf of peers, and maintains information about the peers in the network. To perform these time measurements, the open-source Linux DC++ [78] DirectConnect client was modified. The DirectConnect hub informs clients whenever peers in the network connect or disconnect from the hub. The modified client captured these events and recorded the times at which they occurred.

The measurement trace was conducted on a DirectConnect network operated at the University of Cape Town from 16 August 2007 until 25 August 2007, for a period of 9 days, 6 hours and 42 minutes. Only the first 105 minutes of the measurement trace data was used for the experiments. This was long enough to allow all queries in the query script to execute. Furthermore, since the size of the ModelNet network was fixed to 100 virtual nodes, only the times for 100 peers were taken from the measurement trace. Figure 8.3 shows the changes in the size of the network for the real-world model during the experiments. The network gradually decreased in size until it reaches approximately 62% of its initial size after 66 minutes. Its size then remained relatively stable.

For the Poisson-generated behavioural model, the peer arrival and departure rates were both set to an average of 1 peer per minute. In contrast to the real-world case, the network population size in this model remained relatively high, with slight variations over the course of the test runs. This is shown in figure 8.4.

Both the real-world and Poisson-generated models were encoded as scripts that specified the type of event (whether an arrival or departure), the time an arrival or departure occurred and the peer that entered or departed the network. The prototype was extended to execute this script.

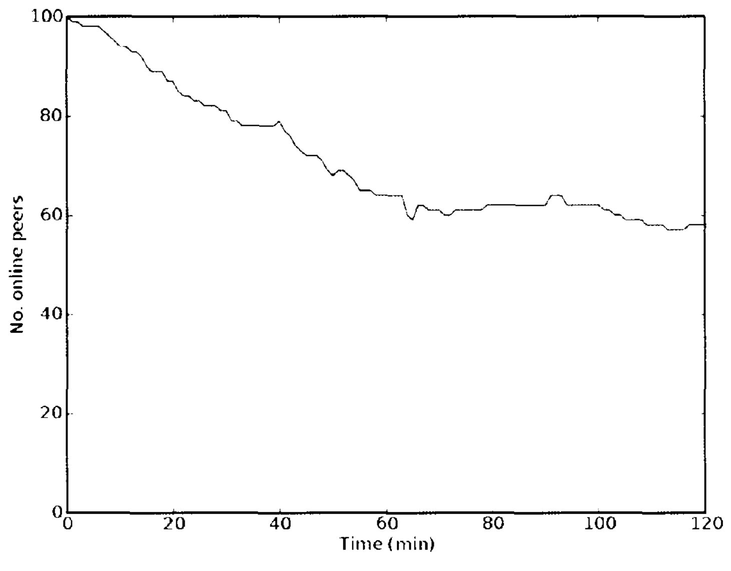


Figure 8.3: Changes in the network population size using the real-world peer online-offline behavioural model

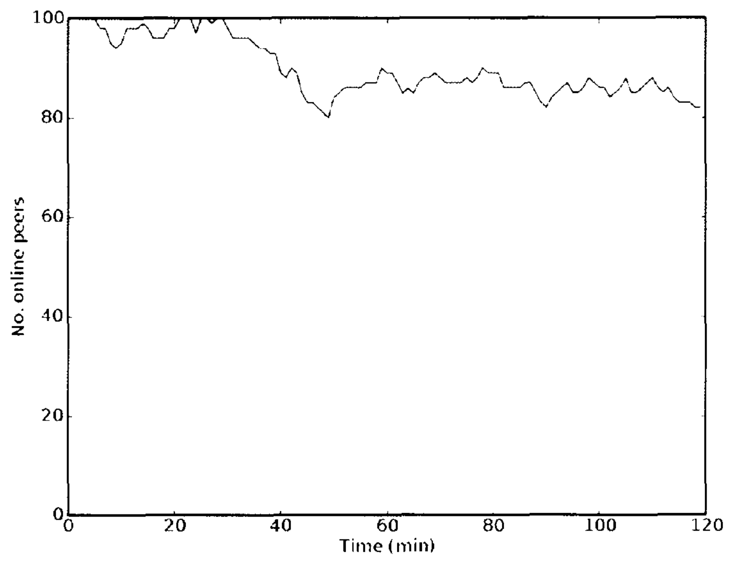


Figure 8.4: Changes in the network population size using the Poisson-generated peer online-offline behavioural model. The average rates of peer arrivals and departures were both set to 1 peer per minute.

8.2 Data Replication Algorithm Evaluation

The data replication algorithm experiments were designed to determine the following:

1. The difference in data availability in a system that performs data replication compared to a system that does not replicate data;

2. How well a system that replicates XML fragments performs compared to a system that only replicates whole XML documents;

3. Which replica selection algorithms performs best in the experimental environment; and

4. Which replica placement algorithms performs best in the experimental environment.

In all of the above cases, the various data replication approaches were evaluated against each other by considering data availability and replication cost.

8.2.1 Method

8.2.1.1 Overview

A test was performed for every combination of replica selection and placement algorithm, with and without fragmentation, for both the real-world and Poisson-generated peer online-offline behavioural models. The tests were also conducted with replication disabled. Each test run consisted of the following steps:

1. Each peer was assigned XML documents as described in section 8.1.2. Any documents or replicas in the peers' data stores remaining from previous test runs were deleted. This ensured that peers had the same initial documents at the start of each test run.
2. The FLC and FPI database tables were populated from the data in the peers' data stores. Any existing table entries from previous test runs were removed, so that each test run had the same initial FLC and FPI setup.
3. One hundred instances of the prototype were started on top of the ModelNet network.
4. After 10 seconds, the boot peer was automatically brought online by the peer online-offline behavioural model script, so that it could create a new FreePastry ring. The boot peer was the entry-point into the network for the other 99 peers.
5. Five seconds after the boot peer created the FreePastry ring, the other 99 peers were brought online, as specified in the peer online-offline behavioural model script.
6. After 5 minutes, the boot peer started executing the query script.
7. Once all 300 queries in the script were evaluated, all 100 prototype instances were shut-down. The data availability and the replication cost were calculated. The manner in which these measurements were taken is explained in the following sections.

During all the test runs, the maximum number of copies of a particular XML fragment or document allowed in the network, m, was set to 10. The replica selection interval, t, was set to 5 minutes. For the MFA and MRA replica selection algorithms, the top n entries in the access information tables that correspond to the number of fragments to replicate during a selection interval was set to 20.

8.2.1.2 Measuring the Data Availability

The data availability was determined by calculating the number of queries that returned complete results, the number of queries that returned incomplete (or partial) results, and the number of queries that returned no results.

A query result is considered complete if all possible XML trees matching the query were returned. This would be the situation if all the relevant XML trees existed in the network at the time the query was evaluated. A result is considered partial if only some of the XML trees matching the query were returned. The missing XML trees would have been due to some of the data being unavailable at the time the query was evaluated. An example of a complete and partial query result is shown in figure 8.5. A complete result for a query $/a/b$ would return the XML trees in figure 8.5(b), while a partial result may look like figure 8.5(c). A result is considered missing if no XML trees were returned for a particular query at all. In this case, all data pertaining to that query were unavailable at the time the query was evaluated.

The number of complete, partial and missing query results were determined as follows:

- *Before all the experiments*, the complete query results for all the test queries were calculated by evaluating the test queries on the collection of test documents. The results of these queries were then recorded. These query results are what the test queries are expected to return if all data were available in the network throughout the test runs.
- *During each test run*, whenever a query was processed at a peer, the query being processed was logged to file, along with the IDs of the XML documents and fragments on which the query was evaluated. The actual query results were not logged, as the additional file I/O would result in unnecessary load on the system. Peers also did not return query results to the querying peer to avoid any performance loss incurred by the serialisation, transfer and deserialisation of query results.
- *After each test run*, the queries were re-evaluated on the XML documents and fragments as specified in the log files using a standard XPath query processing tool. The query results for each query were then collected from the peers and merged, so that they could be compared with the expected query results to determine whether they were complete, partial or missing.

There were two challenges when comparing the actual query results to the expected query results. Firstly, since we were dealing with XML data, equivalent XML trees may look different.

```
<a>
  <b name="b1">
    <c/>
  </b>
  <b>
    <c/>
    <d/>
  </b>
</a>
```

```
<a>
  <b name="b2" />
  <d>
    <e/>
  </d>
</a>
```

(a) Sample XML fragments distributed in the network

```
<b name="b1">
  <c/>
</b>
<b>
  <c/>
  <d/>
</b>
<b name="b2" />
```

(b) The complete result for query $/a/b$

```
<b name="b1">
  <c/>
</b>
<b>
  <c/>
  <d/>
</b>
```

(c) A partial result for query $/a/b$

Figure 8.5: The difference between a complete and partial query result

For instance, consider the two XML documents in figure 8.6. They express the same information, but are formatted differently. Performing a simple string comparison to determine whether the actual results are equal to the expected results would not work. Secondly, since the system replicates data, the query results from the test runs may contain duplicates (the replicas). The identification and removal of these duplicates from the query results would normally be performed by a query processor. However, this was not the case in the current system, as query processing was not the focus of this work. The presence of duplicates further complicated the comparison procedure.

```
<a>
  <b name="b1">
    Some text
  </b>
  <b>
    <c/>
    <d/>
  </b>
</a>
```

```
<a>
  <b name="b1">Some text</b>
  <b><c/>
    <d/>
  </b>
</a>
```

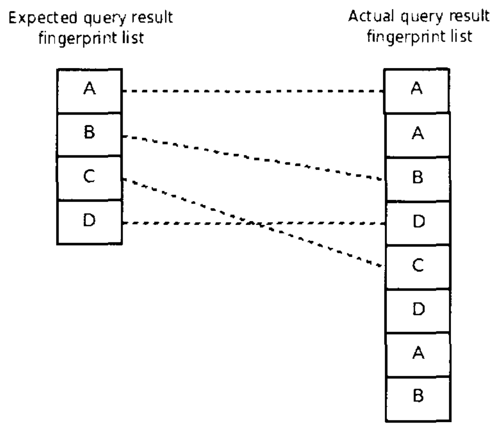
Figure 8.6: Two XML documents expressing the same information, but formatted differently

Comparison was thus performed by creating a fingerprint list for each query result. Each fingerprint in the list corresponded to a XML tree in the query result. This was done for both the expected and actual query result. Using the fingerprint lists, one can determine whether an XML tree appearing in the expected query result also appears in the actual query result. If all the fingerprints for the expected query result also appear in the fingerprint list for the actual query result, then the actual result is complete. If only some of the fingerprints for the expected query result could be found in the actual query result's fingerprint list, then the actual result is partial. If there are no fingerprints for the actual query result, then no data pertaining to that query existed in the network at the time, and a miss occurred. This procedure was repeated for each query in the test query set. The number of complete, partial and missing results for each test run were then recorded. Figure 8.7 illustrates how the comparison was performed using the fingerprint lists.

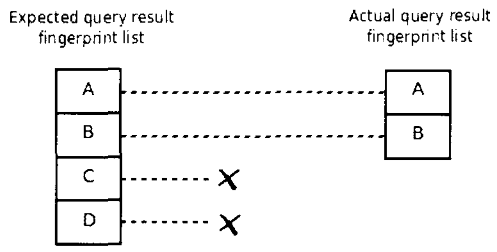
In order to generate the fingerprints, the XML trees in the query results were first converted into string representations. These strings were formatted in a manner such that any two XML trees expressing the same information, would produce the same string, even if they were originally formatted differently. The fingerprints were then obtained by calculating the MD5 hashes of the strings produced. Figures B.3 and B.4 in Appendix B present the fingerprint generation algorithm in pseudocode.

8.2.1.3 Measuring the Replication Cost

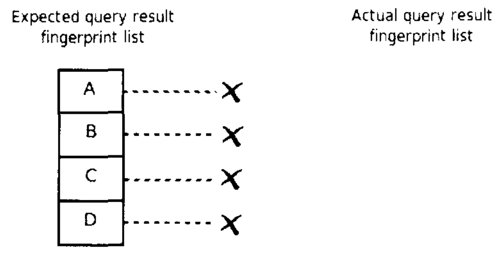
The replication cost is taken as the average amount of replica data transferred per peer. This was measured by wrapping the Java `InputStream` and `OutputStream` objects provided by the socket over which the data was transferred, and logging the amount of data that were sent and received. After the test runs, the total amount of data transferred by each peer was calculated. The replication cost was then taken as the average amount of data transferred per peer.


Expected query result fingerprint list
Actual query result fingerprint list
A
B
C
D
A
A
B
D
C
D
A
B

(a) A complete query result


Expected query result fingerprint list
Actual query result fingerprint list
A
B
C
D
A
B

(b) A partial query result


Expected query result fingerprint list
Actual query result fingerprint list
A
B
C
D

(c) A missing query result

Figure 8.7: Comparing the query result fingerprint lists to determining whether a query result is complete, partial or missing

8.2.2 Results and Analysis

This section discusses the results obtained during the data replication test runs. First, we look at the results for the real-world peer online-offline behavioural model. Then, the Poisson-generated model results are presented.

8.2.2.1 Real-World Peer Online-Offline Behavioral Model

The results for all these test runs are presented in table 8.1. The number of complete, partial and missing query results for each test run is shown, along with the average amount of replica data transferred per peer. The suffixes applied to the replica selection algorithm names in the table indicate whether fragmentation was enabled or not. If fragmentation was enabled, a "+F" is appended to the replica selection algorithm name. Otherwise, a "-F" is appended.

Replication vs No Replication

In all cases, replication improved data availability. Without replication, 92 out of 300 queries returned complete query results, while 34 queries returned nothing. Enabling replication caused the number of complete query results to rise. The highest increase was for the MRA-F/RRP test run, which gave 244 complete query results, an improvement of approximately 165%. MFA+F/CSRP gave the lowest improvement with a 40% increase.

Enabling replication also decreased the number of missing query results across all the test runs. RRS-F/RRP achieved the greatest decrease with a 35% drop, while MRA-F/AURP only achieved a 6% decrease.

Fragmentation vs No Fragmentation

Figured 8.8 to 8.10 show bar charts of the results in table 8.1, comparing the results for a system that replicates XML fragments with those obtained for a system that only replicates whole XML documents.

As shown in figure 8.8, disabling fragmentation resulted in a greater number of complete query results in all cases. This was expected, as replicating whole XML documents means that more data are replicated, which in turn, results in greater data availability. However, we are interested in determining by how much the data availability for a non-fragmenting system exceeds that of a fragmentation-enabled system. When RRS was used, the number of complete query results was on average approximately 24% higher in a non-fragmenting system. When MFA and MRA

Replica Selection Algorithm	Replica Placement Algorithm	No. Complete Query Results	No. Partial Query Results	No. Missing Query Results	Ave. Replica Data Transferred Per Peer (MB)
N/A	N/A	92	174	34	0
RRS-F	RRP	152	134	14	4.19
RRS+F	RRP	192	96	12	8.65
RRS-F	CRP	149	126	25	3.35
RRS+F	CRP	192	95	13	7.57
RRS-F	CSRP	142	134	24	2.83
RRS+F	CSRP	192	95	13	9.28
RRS-F	AURP	149	134	17	5.52
RRS+F	AURP	168	113	19	9.35
MFA-F	RRP	139	130	31	7.12
MFA+F	RRP	206	74	20	10.03
MFA-F	CRP	136	137	27	11.28
MFA+F	CRP	217	64	19	12.60
MFA-F	CSRP	129	147	24	9.55
MFA+F	CSRP	215	60	25	10.52
MFA-F	AURP	134	139	27	10.38
MFA+F	AURP	216	54	30	12.07
MRA-F	RRP	179	96	25	8.48
MRA+F	RRP	244	40	16	18.80
MRA-F	CRP	179	96	25	8.24
MRA+F	CRP	240	44	16	20.36
MRA-F	CSRP	180	98	22	7.91
MRA+F	CSRP	243	40	17	18.66
MRA-F	AURP	169	99	32	7.58
MRA+F	AURP	242	43	15	20.60

Table 8.1: The results for the data replication experiments when using the real-world peer online-offline behavioural model

were used, the number of complete query results were approximately 60% and 37% higher, respectively.

The number of missing query results is compared in figure 8.9. One would expect fewer misses when fragmentation is disabled. However, this is not always the case. In the RRS/AURP, MFA/CSRP and MFA/AURP test runs, enabling fragmentation actually resulted in fewer misses, although not by a considerable margin. Furthermore, the difference between the number of misses varies greatly across the test runs. There is no obvious indication of a pattern emerging when enabling or disabling fragmentation. This is most likely due to the dynamics of the P2P network and the varying load on the cluster machines that could have affected the number of missing results obtained during the experiments.

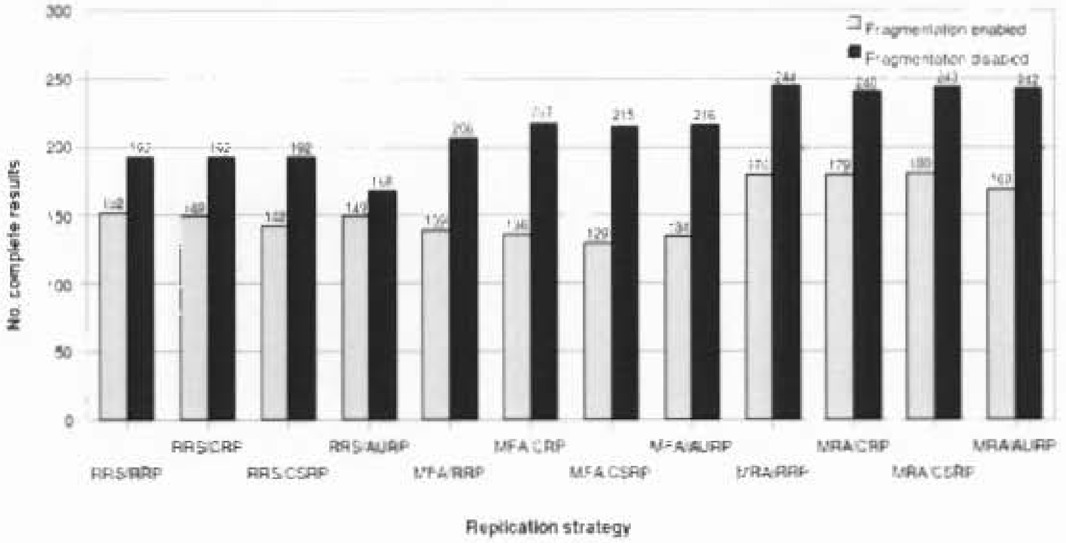


Figure 8.8: Comparing the number of complete query results obtained when fragmentation is enabled and disabled

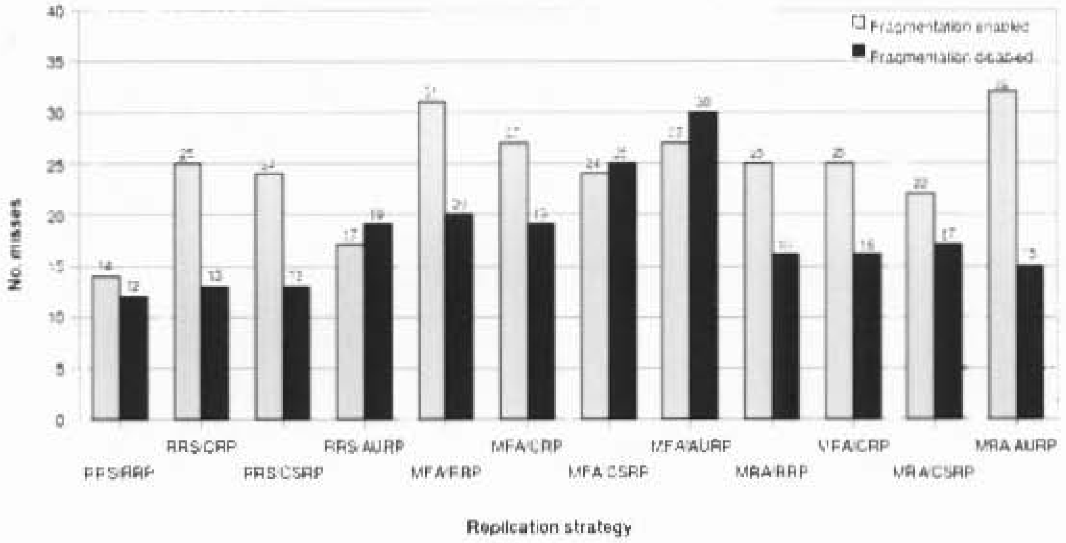


Figure 8.9: Comparing the number of missing query results obtained when fragmentation is enabled and disabled

The replication costs are compared in figure 8.10. As expected, disabling fragmentation always resulted in greater replication cost, as more data are transferred when replicating whole XML documents. However, the difference in cost varies when using different replica selection algorithms. For the RRS and MRA algorithms, the replication cost decreases on average by approximately 55% and 59%, respectively. If one were to choose between a fragmenting and a non-fragmenting system based on this, then a fragmenting system seems preferable. The large drop in replication cost is a fair trade-off for a 20% and 27% drop in the number of complete

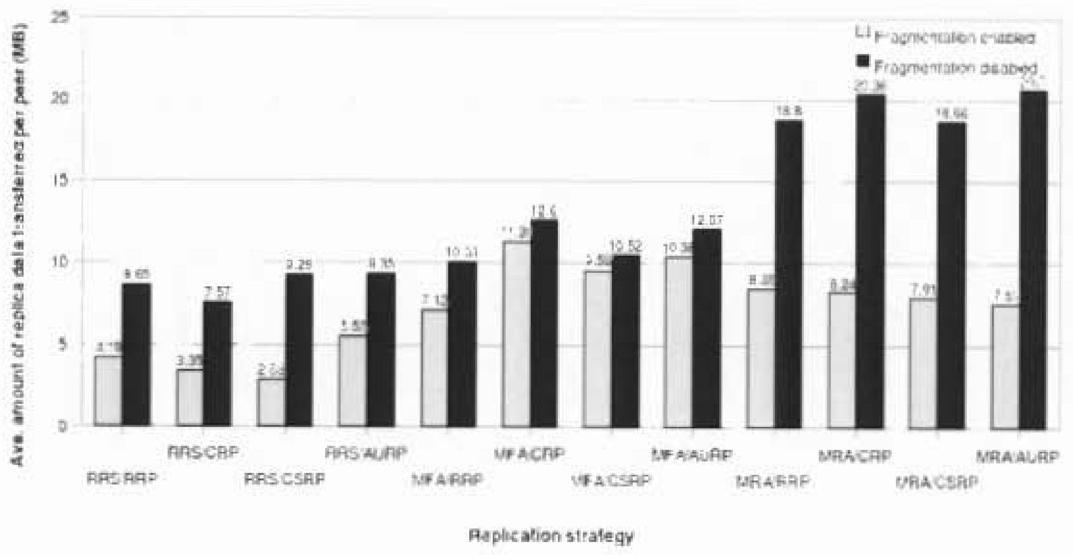


Figure 8.10: Comparing the amount of replica data transferred when fragmentation is enabled and disabled

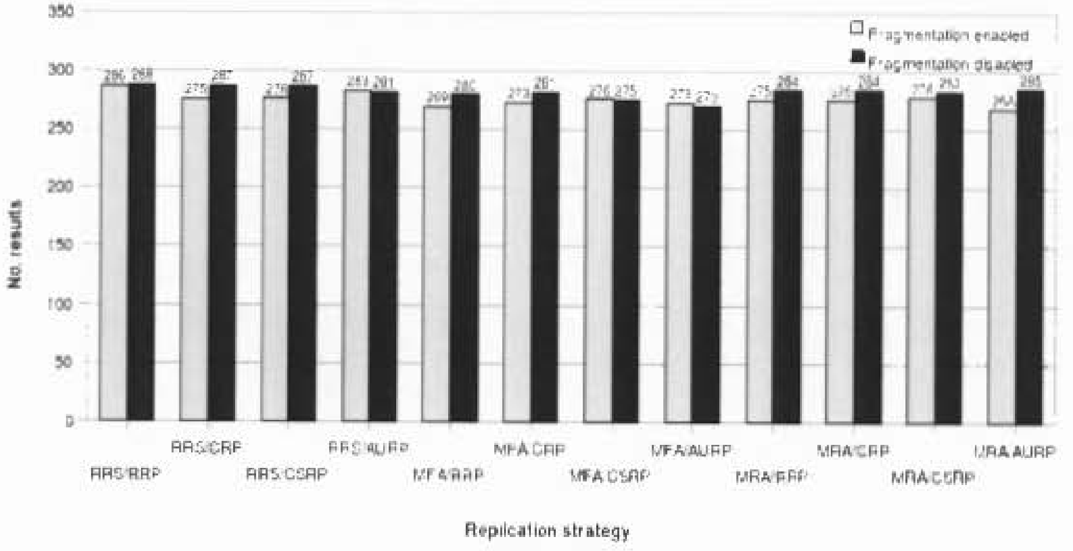


Figure 8.11: Comparing the total number of query results obtained when fragmentation is enabled and disabled

query results, respectively. When using the MFA algorithm, however, the replication cost only drops by approximately 16%. This decrease might be too low to justify a 37% loss in the number of complete results. In this case, a non-fragmenting system might be preferable to a fragmentation-enabled system.

Figure 8.11 compares the total number of query results returned during the test runs (the number of complete query results plus the number of partial results). This is of interest to applications that may not necessarily require complete query results. As shown, there is no appreciable dif-

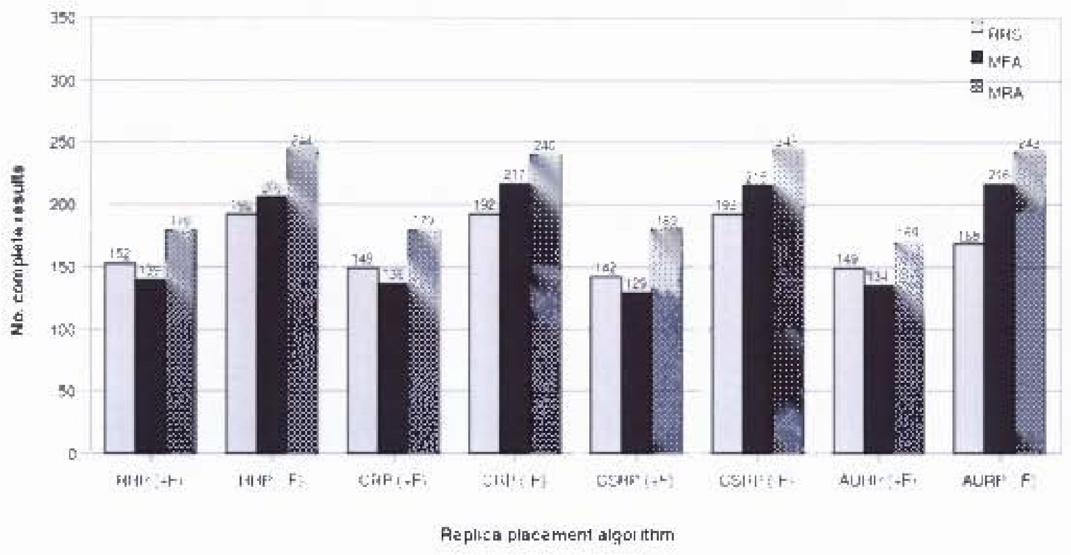


Figure 8.12: Comparing the number of complete query results obtained when using different replica selection algorithms

ference between the number of query results obtained when enabling fragmentation compared to those obtained when disabling fragmentation. In some cases, the total number of query results are even higher when fragmentation is enabled, most likely due to the dynamics of the running system as mentioned earlier. Therefore, for applications in which any query result is acceptable, regardless whether the result is complete or not, a fragmenting system would always be preferable to a non-fragmenting system due to the decreased replication cost. Unfortunately, figure 8.11 does not indicate how complete the query results are. One cannot see, for instance, whether results are 1% complete or 99% complete, which may be a factor in deciding whether to opt for fragmentation or not.

Replica Selection Algorithm Evaluation

Figure 8.12 compares the number of complete query results obtained for the three replica selection algorithms. In each case, MRA yielded the highest number of complete query results, followed by either RRS or MFA, depending on whether fragmentation was enabled or not.

When fragmentation was enabled, RRS performed better than MFA. This seems counter-intuitive. One would expect the access-based algorithms to always yield a greater number of complete query results, as there is some intelligence involved when determining what data to replicate. However, this was not the case. The are three reasons for this. Firstly, RRS has a head-start at replicating data. In contrast to the access-based algorithms, the RRS algorithm starts replicating data from the first replica selection interval, whereas the access-based algorithms have to wait until queries have been processed in order to have information for deciding what data

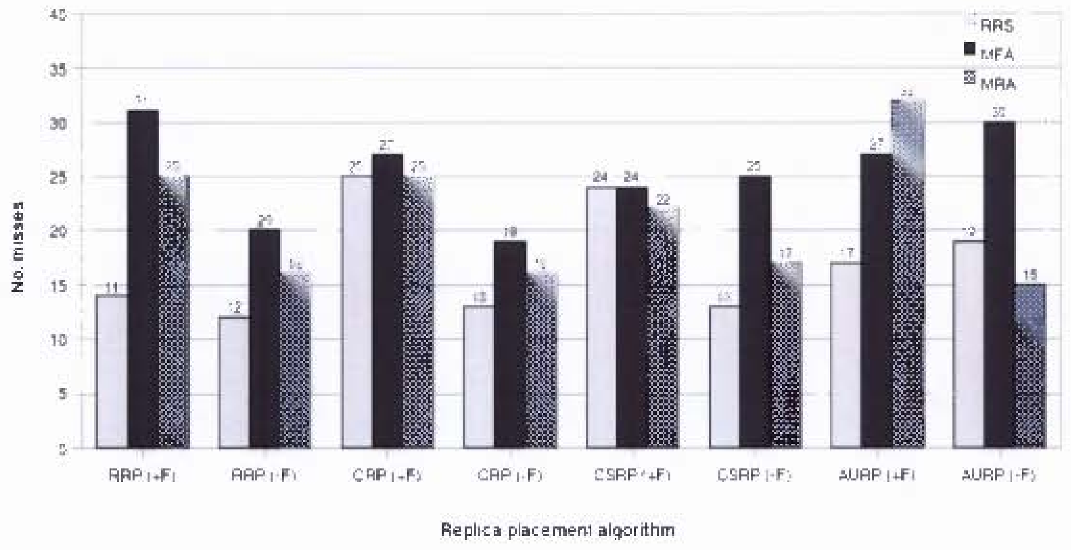


Figure 8.13: Comparing the number of missing query results when using different replica selection algorithms

to replicate. Secondly, although the MFA algorithm replicates data based on access history, the test queries were submitted at random. There was no particular query access pattern; it was as if the query pattern was constantly changing. Finally, because RRS is completely random, it might have selected complete XML documents for replication, whereas the MFA algorithm only replicated small fragments. All these factors together contributed to the lower complete query results count.

The MFA algorithm only outperformed RRS when fragmentation was disabled. In this case, the random nature of RRS caused it to replicate too much unnecessary data compared to MFA.

Looking at the number of missing query results in figure 8.13, there is no evidence to suggest that one replica selection algorithm results in a greater number of misses than another.

The replication cost is compared in figure 8.14. In all cases, RRS consumed less bandwidth than the access-based algorithms. When fragmentation was enabled, it resulted in 57% lower replication cost than MFA and 50% lower than MRA. When fragmentation was disabled, the reduction in cost was on average 25% that of MFA and 56% that of MRA. The reduced cost when using RRS seems counter-intuitive at first, because the selection of what data to replicate is completely random. However, this cost reduction may be explained by considering how the three replica selection algorithms operate. The access-based algorithms replicate the fragments corresponding to the top n entries in the access information table. In the experiments, a value of 20 was used for n. The RRS algorithm, on the other hand, randomly selects k documents from a peer's data store, where $1 \leq k \leq s$, and s is the number of XML documents in the

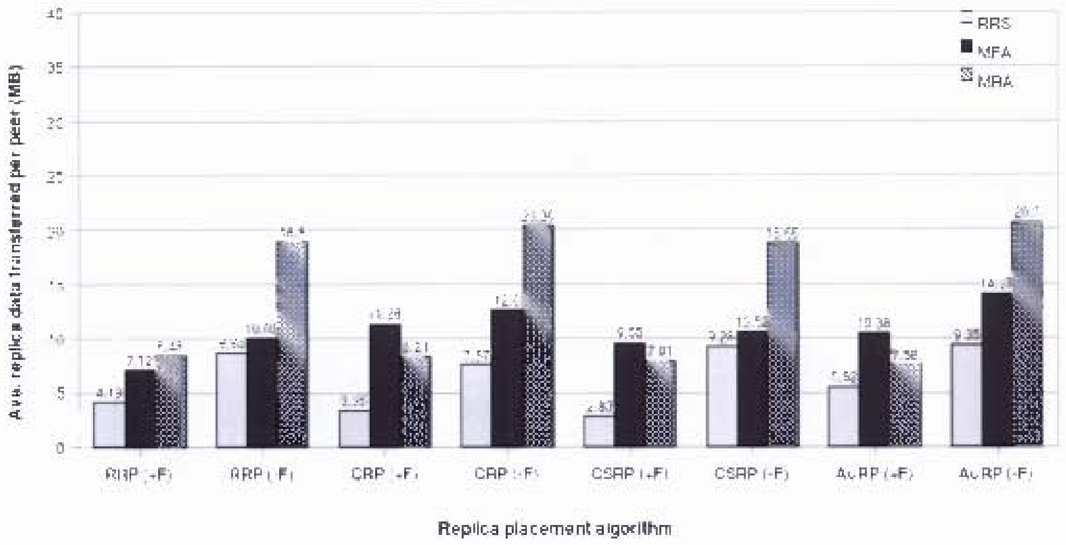


Figure 8.14: Comparing the amount of replica data transferred when using different replica selection algorithms

peer's data store. If s is small (less than 20), then the RRS will, in general, replicate less data than the access-based algorithms, resulting in a lower replication cost. The reduction in cost is greater when fragmentation is enabled, because the sizes of the access information tables for the access-based algorithms grow large very quickly compared to the number of documents in the data store (an entry exists in the access information table for each subtree accessed at a peer).

Comparing the replication costs for the access-based algorithms, we find that MRA consumes less bandwidth than MFA when fragmentation is enabled, while the opposite seems true when fragmentation is disabled. One exception to this is when RRP is used with fragmentation enabled. In this case, MFA results in less cost than MRA. This is most likely due to the random nature of the placement algorithm, which could have had an effect on the cost.

Replica Placement Algorithm Evaluation

Figures 8.15 and 8.16 compare the number complete and missing query results, respectively, when using different replica placement algorithms. There is no evidence that indicates that one placement algorithm provides greater data availability than another. Furthermore, as figure 8.17 shows, the choice of replica placement algorithm does not affect the replication cost.

The similarity in results obtained for the placement algorithms can be attributed to the peer online-offline behaviour. The network remains fairly stable after 66 minutes, as can be seen in figure 8.3. The peers remaining in the network after the initial population size decrease exhibit very similar behaviour. If there was a greater variation in the peer population size over the course of the test runs, the difference in performance between the various placement algorithms might have been more evident.

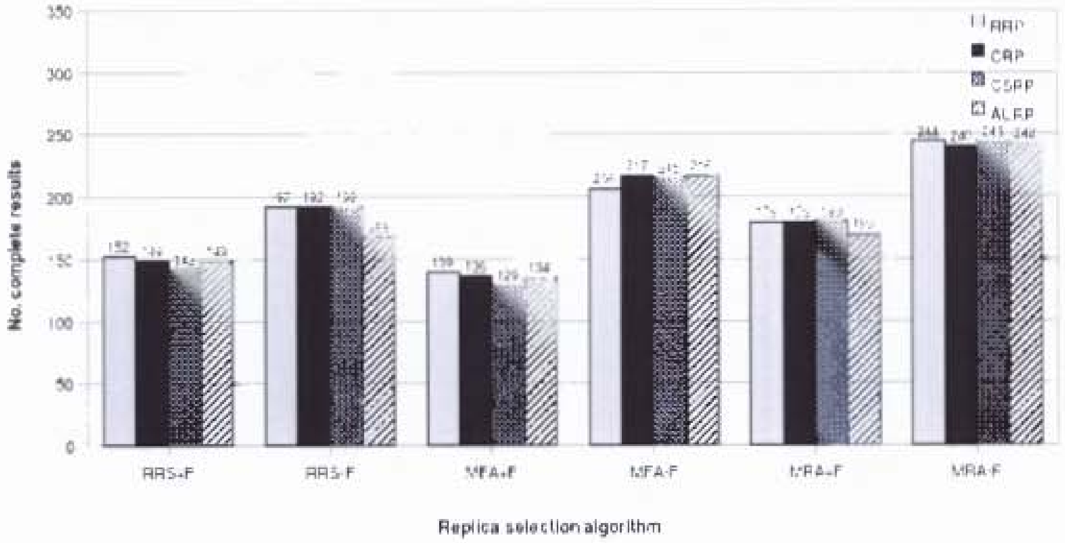


Figure 8.15: Comparing the number of complete query results obtained when using different replica placement algorithms

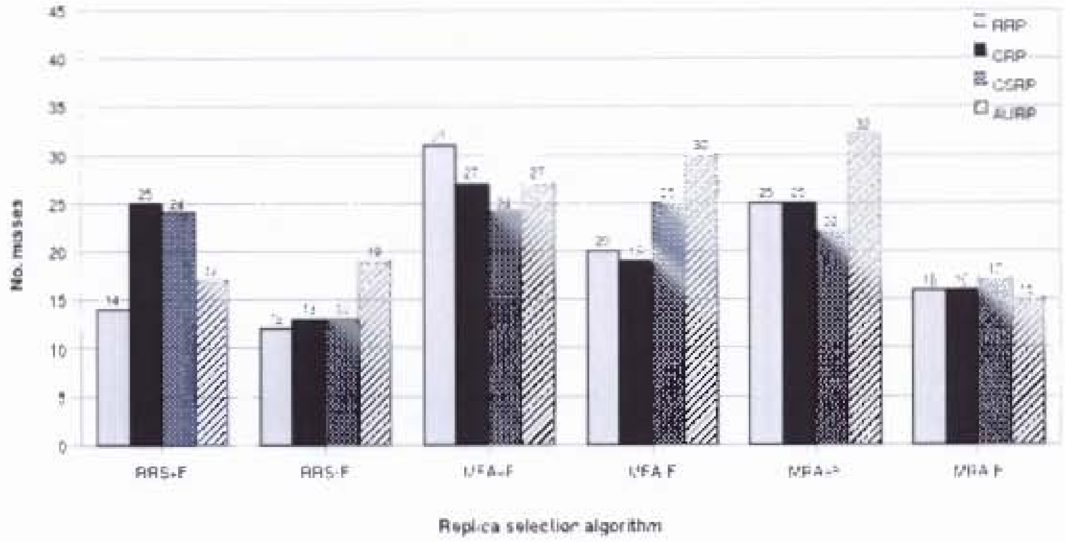


Figure 8.16: Comparing the number of missing query results when using different replica placement algorithms

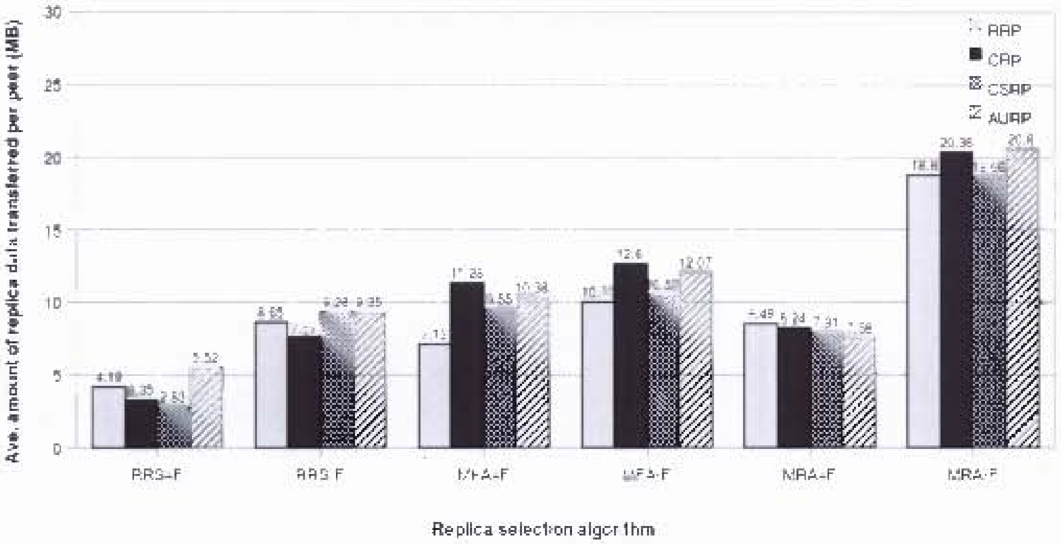


Figure 8.17: Comparing the amount of replica data transferred when using different replica placement algorithms

8.2.2.2 Poisson-Generated Peer Online-Offline Behavioural Model

The results obtained for this behavioural model are very similar to those for the real-world model. These are presented in table 8.2.

Replication vs No Replication

As before, performing data replication resulted in greater data availability. Whenever replication was enabled, the number of complete query results was higher than when replication was disabled. The highest increase was experienced during the MRA-F/AURP test run, which gave an increase of approximately 55%, while the lowest increase occurred during the MFA/CSRP and MFA/AURP test runs, which both gave increases of approximately 4%.

The number of missing query results was also lower when using replication, with the exception of two test runs: MFA-F/AURP and MRA-F/RRP. In these two test runs, the number of misses is actually higher than that of the non-replicating test run. The only explanation for this unexpected result is that extra load on the cluster machines negatively affected the stability of the running prototype. Since this negative result was only observed for the MFA and MRA algorithms, the extra load could be due to the additional work performed by those algorithms to maintain their access information tables, which could have hindered system performance. The extra load could also have stemmed from the network environment. Peers enter and leave the network more frequently in the Poisson-generated model than in the real-world model. The

Replica Selection Algorithm	Replica Placement Algorithm	No. Complete Query Results	No. Partial Query Results	No. Missing Query Results	Ave. Replica Data Transferred Per Peer (MB)
N/A	N/A	183	104	13	0
RRS-F	RRP	215	77	8	3.28
RRS+F	RRP	240	55	5	10.93
RRS-F	CRP	224	66	10	6.70
RRS+F	CRP	239	57	4	7.57
RRS-F	CSRP	227	67	6	3.61
RRS+F	CSRP	259	33	8	10.54
RRS-F	AURP	243	50	7	7.29
RRS+F	AURP	251	46	3	7.83
MFA-F	RRP	194	97	9	9.67
MFA+F	RRP	265	25	10	12.74
MFA-F	CRP	195	95	10	9.52
MFA+F	CRP	260	36	4	13.77
MFA-F	CSRP	191	98	11	8.70
MFA+F	CSRP	271	20	9	13.41
MFA-F	AURP	191	95	14	8.60
MFA+F	AURP	270	21	9	15.87
MRA-F	RRP	205	81	14	9.68
MRA+F	RRP	269	21	10	25.99
MRA-F	CRP	226	67	7	8.61
MRA+F	CRP	271	22	7	24.76
MRA-F	CSRP	220	72	8	9.82
MRA+F	CSRP	270	21	9	24.65
MRA-F	AURP	216	77	7	10.12
MRA+F	AURP	284	10	6	26.05

Table 8.2: The number of query results and amount of replica data transferred

system might therefore require more work in order to maintain the structure of the network in the face of the increased activity, placing further strain on system resources.

Fragmentation vs No Fragmentation

In all the tests, disabling fragmentation yielded a greater number of complete query results than enabling fragmentation. On average, the number of complete query results was approximately 8% higher when RRS was used, while it was approximately 28% and 21% higher when MFA and MRA were used, respectively. When considering the total number of query results returned (the number of complete plus partial query results), no appreciable difference between a fragmenting and non-fragmenting system can be found. As for the number of missing query results, disabling fragmentation caused fewer misses most of the time as expected, but there are some cases in which fewer misses were obtained for a fragmenting system. This was experienced in the real-world model as well, and is due to the dynamics of the running system. Finally, fragmentation resulted in a lower replication cost than a non-fragmenting system as expected. On average, a 39% drop in cost was obtained when RRS was used, a 34% drop was obtained when MFA was used, while a 62% decrease was experienced when MRA was used.

Replica Selection Algorithm Evaluation

The difference between the three replica selection algorithms in terms of the number of complete query results was not as great as that observed in the real-world case. This is due to the relatively high network size maintained during the test runs. When fragmentation was disabled, the algorithms rank the same as before. MRA performed the best, followed by MFA, and then RRS. However, when fragmentation was enabled, RRS performed the best. This is different than in the real-world case, in which RRS ranked second-best.

With regard to replication cost, the results are similar to the real-world experiments. When fragmentation was disabled, MFA resulted in the highest replication cost by a considerable margin: on average approximately 84% more than MFA and 179% more than RRS. RRS always resulted in the lowest replication cost, approximately 32% lower than MFA. When fragmentation was enabled, the access-based algorithms resulted in similar cost. Sometimes MFA had less cost than MRA, while at other times, MRA had less cost than MFA. RRS had the lowest cost again, approximately 43% less than MFA and 45% less than MRA.

Replica Placement Algorithm Evaluation

The replica placement algorithms gave similar results for the number of complete query results and the replication cost, despite the Poisson-generated behavioural model having greater variations in the network size over the course of the test runs than the real-world model. Nothing suggests that one algorithm performs better than another. There might still have been too many similar behaving peers in the network for there to be a noticeable difference between the placement algorithms.

8.3 Update Propagation Algorithm Evaluation

The update propagation algorithm experiment measured the percentage of XML fragment accesses that returned the up-to-date results for various XML fragment update frequencies.

8.3.1 Method

8.3.1.1 Overview

For simplicity, the experiments did not involve real XML document updates. Instead, updates were simulated by simply incrementing XML fragment version numbers.

The times at which updates took place and the peers that performed updates were specified in an update script similar to the query script described in section 8.1.3. The times between updates were generated using a Poisson process, while the peers performing the updates were randomly selected from a list of online peers as specified in the online-offline behavioural model script.

Peers performed updates by randomly selecting a locally-owned XML document in their data stores. Then, a pseudorandom number generator was used to determine which fragments extracted from the selected document were updated.

The experiment consisted of the following steps:

1. A network of 100 instances of the prototype was started in the same manner as described in steps 1 to 6 in section 8.2.1.1.

2. Approximately 5 minutes after the 100 instances were brought online, the update script was executed at each peer.

3. After all test queries were evaluated on the system, the 100 instances were shutdown. The percentage of up-to-date fragments accessed by the queries were then calculated.

These steps were repeated for various update frequencies, with the update propagation algorithm enabled and disabled. The replication strategy was fixed to Random Replica Selection with Random Replica Placement. Only the real-world peer online-offline model was used.

8.3.1.2 Calculating the Percentage of Up-to-date XML Fragments

During the experiment, each peer maintained an update log and an access log. Whenever a peer updated a local document, it logged the time of the update, the IDs of the fragments that were affected by the update, as well as the new versions of the fragments. Whenever a fragment was accessed, the access time, fragment ID, fragment version and the query ID were logged.

After the experiment, the update logs for all the peers were merged into one log. The same was done for the access logs. Then, for each entry in the merged access log, the merged update log was checked to see whether an update for the fragment was made and whether the version of the accessed fragment is the latest version. After all the access log entries were processed, the percentage of latest version fragments was calculated and recorded.

Note that if one peer accessed the latest version of a fragment while another accessed an older version during the same query, then only the latest version access was used in the calculation. The old version access was ignored. The rationale behind this is that a query processor aggregating the results sent by peers would be able to identify different versions of the same fragment in the result set and discard any old versions if a newer version is present.

8.3.2 Results and Analysis

A plot of the percentage of up-to-date XML fragments for various update frequencies is shown in figure 8.18. As expected, the update propagation algorithm resulted in a higher number of latest version accesses than a system without an update propagation mechanism.

However, the difference between an update propagation enabled system and a system without update propagation is not as great as we had hoped, especially for high update rates. This could be as a result of the manner in which fragment accesses are handled in the prototype. The update retrieval phase of the update propagation algorithm is initiated whenever a peer attempts to access a fragment that has been marked stale. Instead of waiting for the update to arrive, the peer immediately returns the stale fragment, as the update retrieval process may take an arbitrary amount of time to complete. This reduces the percentage of latest version fragments returned in the query results. Another reason for the low percentage could be that replica holders initiate the update retrieval process too late. By the time updates are requested, peers storing later versions of the fragment might have already left the network.

For systems in which low updates rates are the norm (less than 5 updates per minute), the cur-

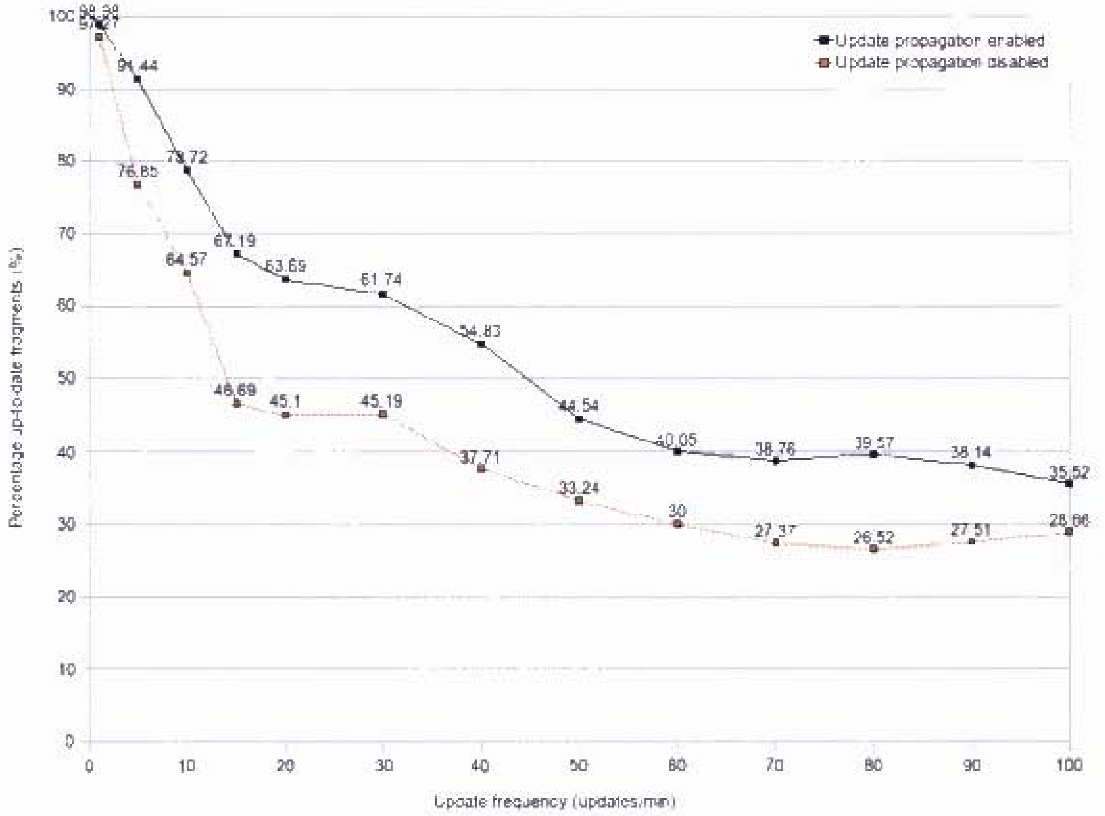


Figure 8.18: Comparing the percentage of up-to-date fragments accessed in an update propagation enabled and disabled system for varying update rates

rent update propagation algorithm may be sufficient. However, for systems that expect greater update rates, a more sophisticated algorithm may be required.

8.4 Summary

This chapter presented experiments conducted to evaluate the data replication and update propagation algorithms.

The data replication experiments were conducted to determine the following: (1) the difference in data availability in a system that performs data replication compared to a system that does not replicate data, (2) how well a system that replicates XML fragments compares to a system that only replicates whole XML documents, (3) which replica selection algorithm performs best under the experimental network conditions, and (4) which replica placement algorithm performs best in the experimental network environment. These experiments were performed for various combinations of replica selection and placement algorithms in a network of peers whose behaviours were modelled on that of a real P2P system as well as generated using a Poisson process.

With replication enabled, the number of complete query results showed an increase of between 40% and 165% for the real-world P2P model, and an increase of between 4% and 55% for the Poisson-generated model.

When comparing a system that fragments XML documents to one which does not, it was found that replicating whole XML documents resulted in a greater number of complete query results. When RRS was used, the number of complete query results were on average 24% higher in the real-world case for a non-fragmenting system than a fragmentation-enabled one, while for MFA and MRA, it was 60% and 37% higher, respectively. In the Poisson-generated model, these increases were 8%, 28% and 21% higher for RRS, MFA and MRA, respectively. However, enabling fragmentation resulted in a lower replication cost. A decrease of between 16% and 59% in replication cost was obtained in the real-world case, while a 34% to 62% decrease in cost was seen for the Poisson case.

Looking at the replica selection algorithms, MRA yielded the highest number of complete query results when fragmentation was disabled, followed by MFA and then RRS. This was observed in both the real-world and Poisson-generated network models. However, when fragmentation was enabled, the replica selection algorithms ranked differently. In the real-world case, MRA performed the best, followed by RRS and then MFA. In the Poisson-generated case, RRS ranked the highest, followed by MRA and then MFA. In terms of replication cost, the RRS always performed the best of the three replica selection algorithms.

Finally, comparing the replica placement algorithms, no appreciable difference was observed that suggested that one placement algorithm performed better than another.

The update propagation experiment looked at the percentage of XML fragment accesses for which the latest versions were returned. This experiment was conducted on a system with update propagation enabled and then repeated on a system with update propagation disabled. Enabling update propagation increased the percentage of latest versions accessed, but not by as big a margin as we had hoped. The low increase can be attributed to the system returning stale fragments as soon as they are accessed rather than waiting for the update retrieval process to complete and updates to arrive. It could also be due to the late initiation of the update retrieval process.

Chapter 9

Conclusions and Future Work

This project investigated the data replication and update propagation problem in XML P2P systems as a means of improving data availability in the face of peer departures. A survey of existing XML P2P systems revealed that this problem has not yet received much attention from the research community. Current systems instead focus on novel P2P XML indexing and query processing techniques. We believe that in order to properly support data replication and update propagation in a P2P system, such mechanisms need to be engineered into the system early in the design process. To this end, we designed a XML P2P data management framework that incorporates such mechanisms and components.

In section 1.1, we identified three mechanisms that a P2P system must provide to ensure data availability: wide-spread data replication, replica location and update propagation.

To support wide-spread data replication, our XML P2P framework was designed to sit on top of any P2P network implementation. By decoupling it from the underlying network, it does not depend on the network structure or the routing algorithms used. As a result, data items are not limited to specific locations in the network as they would be in structured P2P systems. We also proposed various data replication strategies for the framework by dividing the data replication problem into replica selection and replica placement, and allowing selection and placement algorithms to be changed independently. Our experiments showed that whatever combination of selection and placement algorithm were used, the availability of data in the network increased compared to a system without any data replication.

The framework also allows for the reduction of replication cost by accommodating the fragmentation of XML documents into smaller pieces and replicating those instead of whole XML documents. We found that this resulted in considerable cost reduction at the price of some data availability loss. However, the cost reduction is large enough to justify the drop in data availability.

To support the location of replicas in the network, the framework defines a Fragment Location Catalogue (FLC) that maps replica IDs to peer IDs. This further helps separate the framework from the underlying P2P network and facilitates wide-spread data replication. The FLC may be implemented as either a web service or built into the network (e.g. in the form of a DHT).

Finally, to tackle the update propagation requirement, we proposed a lazy update propagation algorithm that sends updates to peers in a two-stage process. Experiments showed that this algorithm increases the probability that data returned by queries are up-to-date, although the percentage of up-to-date data returned was not as high as we would like it to be, especially for high update rates. We also described how updates are handled in the framework, covering topics such as XML document update types, identifying what fragments extracted from documents have changed, and representing updates as deltas.

Having looked at all three issues - wide-spread data replication, replica location and update propagation - our framework meets our initial requirements defined in this dissertation, and we have accomplished what we set out to achieve. We have not only proposed data replication and update propagation algorithms, but also presented an entire solution for an XML P2P data management system, and developed an evaluation system that facilitates further experimentation. This work therefore serves as a good base for future research in the field.

Future Work

There are still a number of issues that could be addressed in future. This section highlights some of these.

1. *Investigating more replica selection techniques*. During the data replication experiments, Random Replica Selection (RRS) performed well compared to the access-based approaches, especially Most-Frequently Accessed Replica Selection (MFA). As mentioned in section 8.2.2.1, one of the reasons for this is that RRS had a head-start at replication data, whereas the access-based algorithms had to wait for queries to be posed on the system before having information on what fragments to replicate. While testing could have compared the approaches only after the system had been running sufficiently long, this result does raise an important issue. Future work on replica selection techniques could look at

combining RRS and the access-based algorithms, so that replication can take place before the access information tables of the access-based algorithms are filled. An adaptive selection mechanism could also be employed that changes the replica selection algorithm depending on the current system conditions. For example, initially, RRS could be used while there is still insufficient information available on the query pattern. Then, the mechanism could switch to the MFA algorithm when the query pattern has become predictable, or to the Most-Recently Accessed Replica Selection (MRA) algorithm if the query pattern is sporadic. Lastly, more heuristics for selecting fragments could be investigated, such as taking the size of fragments into account or using more sophisticated access pattern analyses and prediction techniques.

2. *Investigating more replica placement techniques.* The current replica placement algorithms base their placement decisions on the past behaviour of the peers in the network. More sophisticated algorithms could be introduced to predict the behaviour of the peers t time units into the future. The current placement algorithms could also be improved by only taking into account the last m time units. This might prevent the peers' earlier behaviour from overshadowing their current behaviour.

3. *Improving the update propagation algorithm.* Update propagation could be improved by performing periodic checks on the data store contents and initiating the update retrieval process as necessary, so that updates can be retrieved sooner. These periodic checks will, of course, introduce extra bandwidth costs that need to be investigated, as peers would make more update retrieval requests in the network. An adaptive approach could be used to control the frequency of these periodic checks. If a peer receives an increasing number of update notifications, the frequency of the checks could be increased, and vice versa. The update propagation algorithm could also be compared with an eager update propagation algorithm. Our reason for using a lazy approach over an eager algorithm that immediately propagates updates to peers was to reduce the bandwidth cost involved in transferring updates. The extent to which the lazy approach reduces this cost still needs to be investigated.

4. *Experiment with different FLC implementations.* In this work, any bandwidth and performance cost associated with the retrieval of information from the FLC has been ignored. Future work in this area could look at various implementation approaches, such as using a web service or DHT. The costs involved could then be measured to determine what effect they have on the data replication and update propagation algorithms, and on the overall performance of the system.

5. *Experiment with more network scenarios.* This work assumed an infinite amount of stor-

age space at peers. As a result, the effects of limited storage space on the data replication algorithms are not known. Different replica eviction policies could also be used to see what effect they have on the algorithms. In addition to this, experiments could be conducted with more query patterns and peer online-offline behaviour models. In particular, it would be interesting to see how well the system copes with random network splits and failures in which large parts of the network are suddenly taken offline.

Appendix A

Low-Level Prototype Implementation Details

This section describes the low-level implementation details of the prototype XML P2P data system discussed in chapter 7. A UML class diagram is presented for each component, along with a brief description of how the classes and interfaces interact.

A.1 The P2P Manager

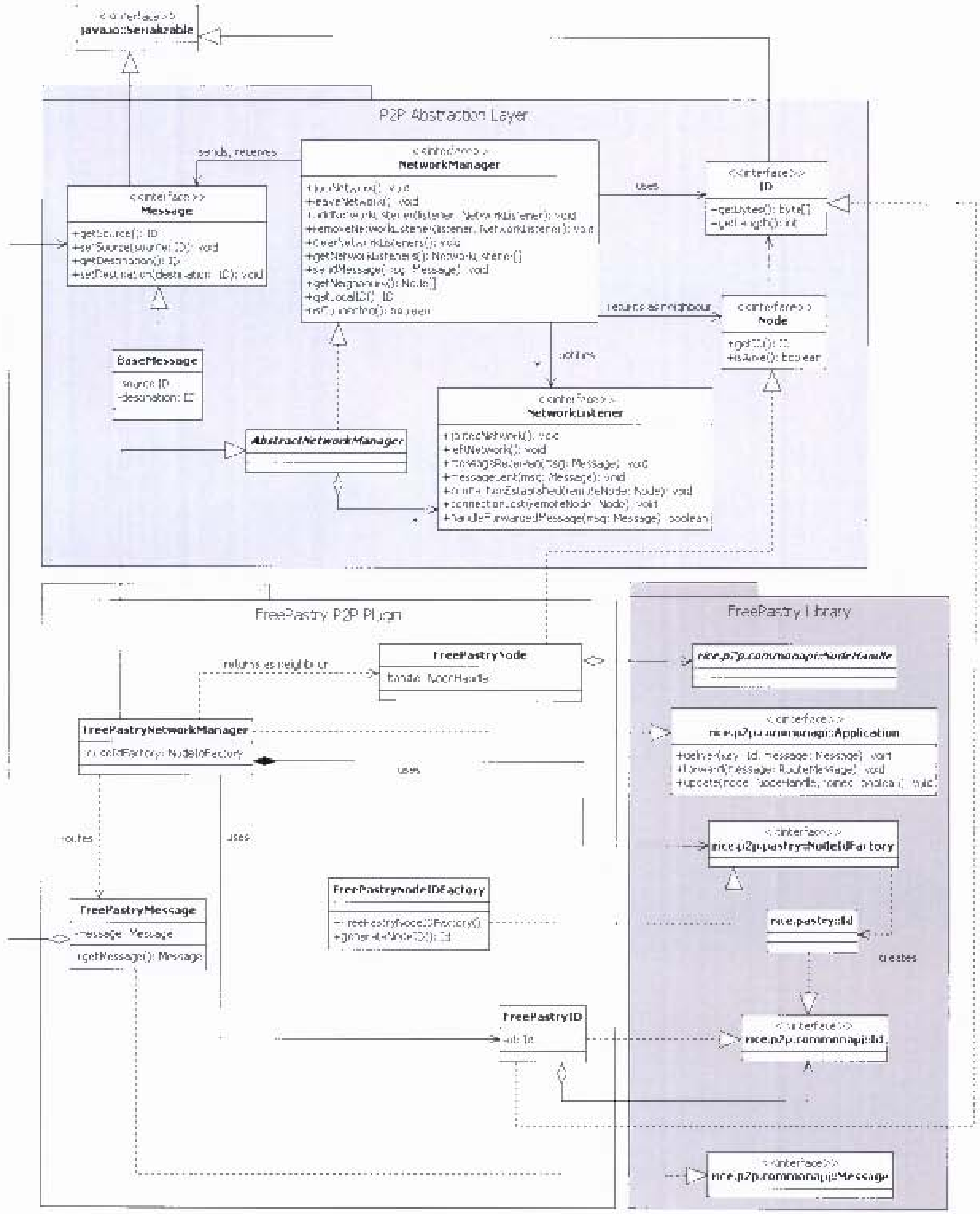


Figure A.1: UML class diagram of the P2P Manager component

The P2P Manager component consists of the following generic interfaces. These interfaces provide operations common to all P2P network implementations:

`ID` represents the ID of a peer or data item in the system.

`Node` maintains all the information about a remote peer in the network. It has operations to return the ID of the remote peer and to check whether a remote peer is online.

`Message` represents the actual message that is transferred between peers in the network.

`NetworkListener` receives notifications whenever the following network events occur:

- When the peer joins and leaves the network;
- When messages are sent and received;
- When messages are forwarded by a peer en route to their destinations; and
- When connections to remote peers are established or broken.

`NetworkManager` implements the P2P networking mechanisms and routing algorithms. It the main interface through which the all higher level components in the system interact in order to use the P2P Manager component. Operations are provided to join and leave the network, send messages, and to retrieve the local peer ID and list of neighbour peers. The `NetworkManager` is also the entity in the P2P Manager that notifies `Network-Listener` implementations of network events.

The lightly shaded package in figure A.1 is the abstraction layer, the dark shaded package is the FreePastry library, while the unshaded package is the "glue" between the abstraction layer and the FreePastry library. `FreePastryID` and `FreePastryNode` are adapter classes [31] that adapt `Id` and `NodeHandle` in the FreePastry library, respectively, to the corresponding interfaces in the abstraction layer. The `FreePastryMessage` class encapsulates `Message` objects so that they may be handed to the FreePastry routing layer. The `FreePastryNet-workManager` adapts the `Application` interface in the FreePastry library to the `Net-workManager` interface to expose the FreePastry functionality to higher-level components in the system. Finally, the `FreePastryNodeIDFactory` implements the peer ID generation algorithm mentioned previously, and is used by the `FreePastryNetworkManager` to obtain the local peer ID.

A.2 The Fragment Creator

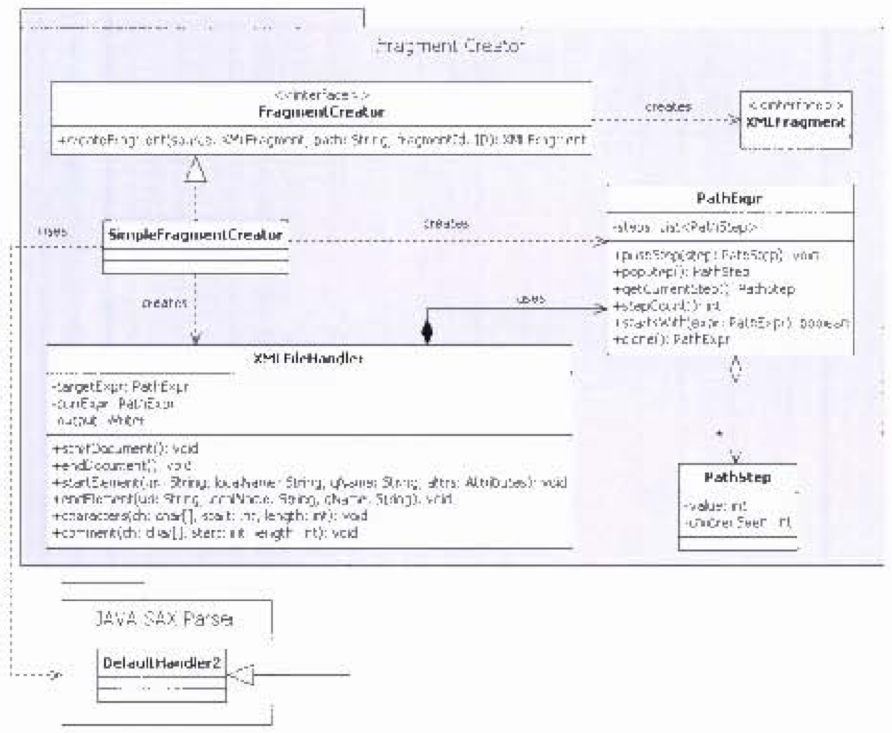

Fragment Creator
<<interface>>
FragmentCreator
+createFragment(source: XMLFragment, path: String, fragmentId: ID): XMLFragment
creates
<<interface>>
XMLFragment
PathExpr
-steps: List<PathStep>
+pushStep(step: PathStep): void
+popStep(): PathStep
+getCurrentStep(): PathStep
+stepCount(): int
+startsWith(expr: PathExpr): boolean
+clone(): PathExpr
uses
SimpleFragmentCreator
creates
creates
uses
XMLFileHandler
-targetExpr: PathExpr
-currExpr: PathExpr
-output: Writer
+startDocument(): void
+endDocument(): void
+startElement(uri: String, localName: String, qName: String, attrs: Attributes): void
+endElement(uri: String, localName: String, qName: String): void
+characters(ch: char[], start: int, length: int): void
+comment(ch: char[], start: int, length: int): void
*
PathStep
-value: int
-childrenSeen: int
JAVA SAX Parser
DefaultHandler2

Figure A.2: UML class diagram of the Fragment Creator component

The `FragmentCreator` is the main interface in the Fragment Creator component. It provides a `createFragment()` method which is called by the system to perform the fragment extraction. The `SimpleFragmentCreator` class implements the `FragmentCreator` interface and the SAX-based fragment extraction algorithm. It uses the `XMLFileHandler` class, which implements a number of callback methods that are invoked by the SAX parser whenever certain parts of the XML document are encountered while the document is being parsed.

A.3 The Fragment Location Catalogue

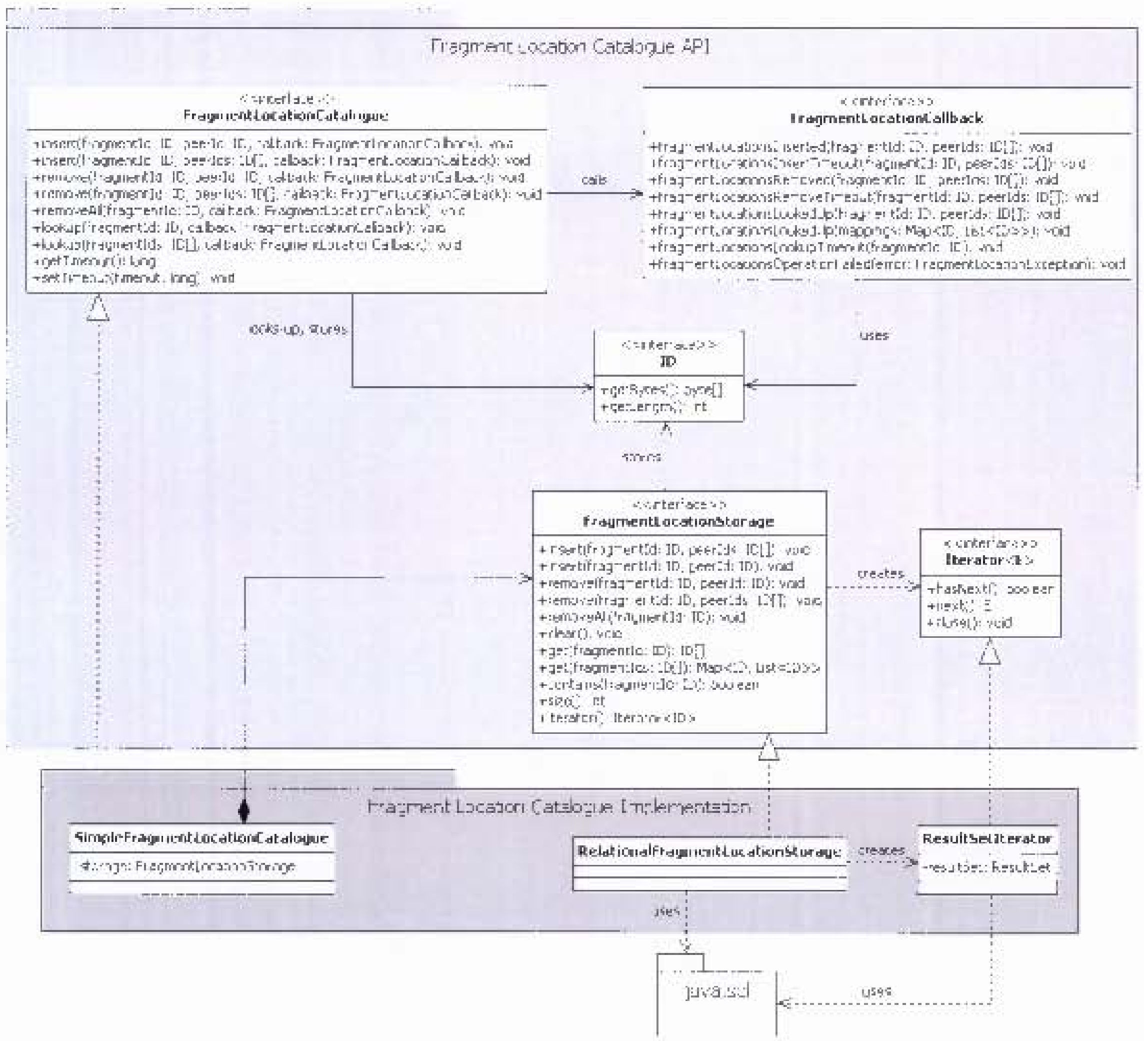

Fragment Location Catalogue API
<<interface>>
FragmentLocationCatalogue
+insert(fragmentId: ID, peerId: ID, callback: FragmentLocationCallback): void
+insert(fragmentId: ID, peerIds: ID[], callback: FragmentLocationCallback): void
+remove(fragmentId: ID, peerId: ID, callback: FragmentLocationCallback): void
+remove(fragmentId: ID, peerIds: ID[], callback: FragmentLocationCallback): void
+removeAll(fragmentId: ID, callback: FragmentLocationCallback): void
+lookup(fragmentId: ID, callback: FragmentLocationCallback): void
+lookup(fragmentIds: ID[], callback: FragmentLocationCallback): void
+getTimeout(): long
+setTimeout(timeout: long): void
calls
<<interface>>
FragmentLocationCallback
+fragmentLocationsInserted(fragmentId: ID, peerIds: ID[]): void
+fragmentLocationsInsertTimeout(fragmentId: ID, peerIds: ID[]): void
+fragmentLocationsRemoved(fragmentId: ID, peerIds: ID[]): void
+fragmentLocationsRemoveTimeout(fragmentId: ID, peerIds: ID[]): void
+fragmentLocationsLookedUp(fragmentId: ID, peerIds: ID[]): void
+fragmentLocationsLookedUp(mappings: Map<ID, List<ID>>): void
+fragmentLocationsLookupTimeout(fragmentId: ID): void
+fragmentLocationsOperationFailed(error: FragmentLocationException): void
looks-up, stores
uses
<<interface>>
ID
+getBytes(): byte[]
+getLength(): int
stores
<<interface>>
FragmentLocationStorage
+insert(fragmentId: ID, peerIds: ID[]): void
+insert(fragmentId: ID, peerId: ID): void
+remove(fragmentId: ID, peerId: ID): void
+remove(fragmentId: ID, peerIds: ID[]): void
+removeAll(fragmentId: ID): void
+clear(): void
+get(fragmentId: ID): ID[]
+get(fragmentIds: ID[]): Map<ID, List<ID>>
+contains(fragmentId: ID): boolean
+size(): int
+iterator(): Iterator<ID>
creates
<<interface>>
Iterator<E>
+hasNext(): boolean
+next(): E
+close(): void
Fragment Location Catalogue Implementation
SimpleFragmentLocationCatalogue
-storage: FragmentLocationStorage
RelationalFragmentLocationStorage
creates
ResultSetIterator
-resultSet: ResultSet
uses
java.sql
uses

Figure A.3: UML class diagram of the Fragment Location Catalogue

The `FragmentLocationCatalogue` interface provides methods to insert, retrieve and remove XML fragment ID to peer ID list mappings. `FragmentLocationCatalogue` implementations are responsible for the FLC-specific logic such as network communication and caching. They do not handle the actual storage of the mappings. Instead, this responsibility is factored into a `FragmentLocationStorage` interface. This allows the FLC-specific logic and the logic that manages the storage of the mappings to vary independently. Additionally, it allows the same storage mechanisms to be reused in different `FragmentLocation-Catalogue` implementations. For instance, a DHT-based FLC would implement the P2P communication logic specific to a particular P2P protocol, but may use a `FragmentLoca-`

`tionStorage` implementation backed by a relational database to actually store the mappings locally.

The operations in the `FragmentLocationCatalogue` interface were designed to be non-blocking. Once they are called, they do not block until the operation completes. To enable such non-blocking method invocation, a `FragmentLocationCallback` interface was introduced. The `FragmentLocationCallback` interface contains methods that are called by `FragmentLocationCatalogue` implementations upon completion of the corresponding operations in the `FragmentLocationCatalogue` interface. `FragmentLocation-Callback` objects are passed to FLC operations as method arguments.

A.4 The Update Manager

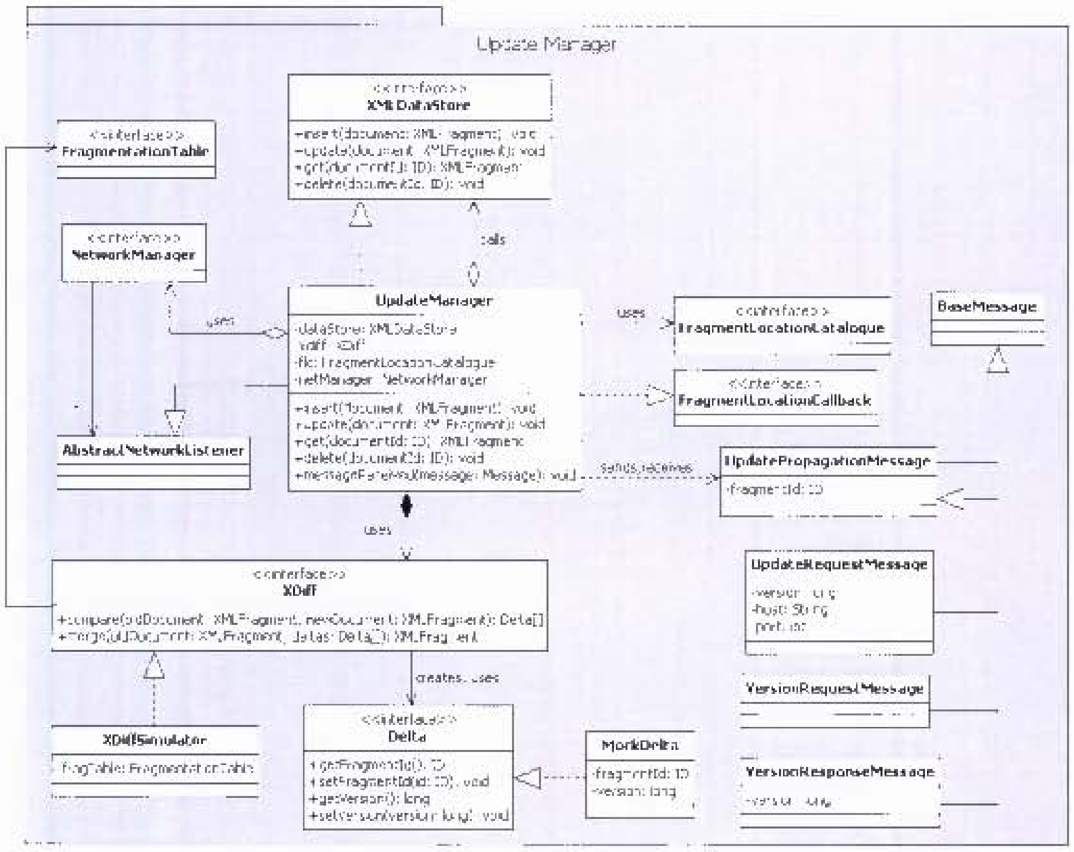

Update Manager
<<interface>>
XMLDataStore
+insert(document: XMLFragment): void
+update(document: XMLFragment): void
+get(documentId: ID): XMLFragment
+delete(documentId: ID): void
<<interface>>
FragmentationTable
<<interface>>
NetworkManager
calls
uses
UpdateManager
-dataStore: XMLDataStore
-xdiff: XDiff
-flc: FragmentLocationCatalogue
-netManager: NetworkManager
+insert(document: XMLFragment): void
+update(document: XMLFragment): void
+get(documentId: ID): XMLFragment
+delete(documentId: ID): void
+messageReceived(message: Message): void
uses
<<interface>>
FragmentLocationCatalogue
BaseMessage
<<interface>>
FragmentLocationCallback
uses
AbstractNetworkListener
sends, receives
UpdatePropagationMessage
-fragmentId: ID
uses
<<interface>>
XDiff
+compare(oldDocument: XMLFragment, newDocument: XMLFragment): Delta[]
+merge(oldDocument: XMLFragment, deltas: Delta[]): XMLFragment
UpdateRequestMessage
-version: long
-host: String
-port: int
creates, uses
VersionRequestMessage
<<interface>>
Delta
+getFragmentId(): ID
+setFragmentId(id: ID): void
+getVersion(): long
+setVersion(version: long): void
XDiffSimulator
-fragTable: FragmentationTable
MockDelta
-fragmentId: ID
-version: long
VersionResponseMessage
-version: long

Figure A.4: UML class diagram of the Update Manager component

The `UpdateManager` class implements the `XMLDataStore` interface, so that it provides the same set of operations to higher-level system components. Within the `UpdateManager` class is a reference to an `XMLDataStore` instance. The `UpdateManager` delegates data store operations to this `XMLDataStore` instance once it has completed its own work.

The `XDiff` interface is used by the `UpdateManager` to determine what XML fragments in an XML document have been updated. It provides a `compare()` method, which takes as input the old and new version of an XML document, and returns an array of `Delta` objects, one for each updated fragment found in the document. The `Delta` objects describe how a fragment was updated. The `XDiff` interface is also responsible for applying `Deltas` to XML fragments to generate the new version of a fragment. This is done using the `merge()` method, which receives the old version of the XML document and an array of `Delta` objects as input, and returns the new version of the XML document.

A.5 The Query Processor

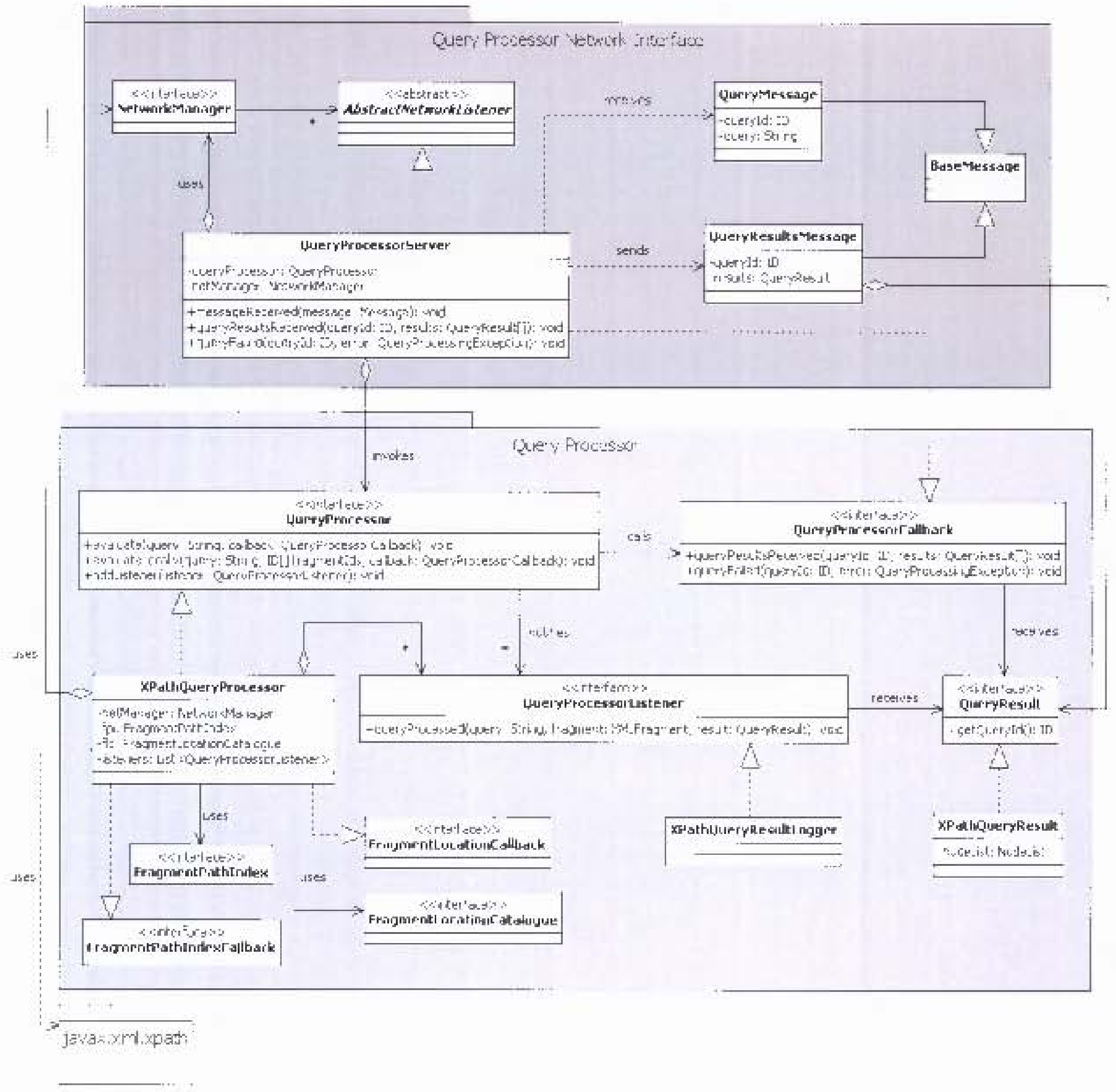

Query Processor Network Interface
<<interface>>
NetworkManager
<<abstract>>
AbstractNetworkListener
receives
QueryMessage
-queryId: ID
-query: String
BaseMessage
uses
QueryProcessorServer
-queryProcessor: QueryProcessor
-netManager: NetworkManager
+messageReceived(message: Message): void
+queryResultsReceived(queryId: ID, results: QueryResult[]): void
+queryFailed(queryId: ID, error: QueryProcessingException): void
sends
QueryResultsMessage
-queryId: ID
-results: QueryResult
Query Processor
invokes
<<interface>>
QueryProcessor
+evaluate(query: String, callback: QueryProcessorCallback): void
+evaluateLocally(query: String, ID[] fragmentIDs, callback: QueryProcessorCallback): void
+addListener(listener: QueryProcessorListener): void
calls
<<interface>>
QueryProcessorCallback
+queryResultsReceived(queryId: ID, results: QueryResult[]): void
+queryFailed(queryId: ID, error: QueryProcessingException): void
notifies
receives
uses
XPathQueryProcessor
-netManager: NetworkManager
-fpi: FragmentPathIndex
-flc: FragmentLocationCatalogue
-listeners: List<QueryProcessorListener>
<<interface>>
QueryProcessorListener
+queryProcessed(query: String, fragment: XMLFragment, result: QueryResult): void
receives
<<interface>>
QueryResult
+getQueryId(): ID
uses
<<interface>>
FragmentLocationCallback
XPathQueryResultLogger
XPathQueryResult
-nodeList: NodeList
<<interface>>
FragmentPathIndex
uses
uses
<<interface>>
FragmentLocationCatalogue
<<interface>>
FragmentPathIndexCallback
javax.xml.xpath

Figure A.5: UML class diagram of the Query Processor component

The Query Processor is represented by the `QueryProcessor` interface. This interface has two methods for evaluating queries: `evaluate()` and `evaluateLocally()`. The `evaluate()` method performs query evaluation on XML fragments in the network as described in section 5.6.2. The `evaluateLocally()` method only performs query processing on local XML fragments. It is called by the `evaluate()` method to perform local query processing.

It is also invoked whenever a query from a remote peer is received. `QueryProcessor-Callback` objects are passed to the `evaluate()` and `evaluateLocally()` methods as arguments, and are notified by the `QueryProcessor` of query results via callback methods. The `queryProcessed()` method in the `QueryProcessorListener` interface is called by the `QueryProcessor` whenever a query was processed on any locally stored XML fragment.

The `QueryProcessorServer` class represents the Query Processor component's network interface. It overrides the `messageReceived()` method in the `AbstractNetwork-Listener` class to capture `QueryMessages` received from remote peers, and passes the queries specified in those `QueryMessages` to the `evaluateLocally()` method in the `QueryProcessor` interface. The `QueryProcesserServer` receives query results from the `QueryProcessor` by implementing the `QueryProcessorCallback` interface. Once it receives a query result, it sends a `QueryResultMessage` using the `NetworkManager` to the remote peer from which it received the query.

The functionality of the FPI is separated into a `FragmentPathIndex` and a `Fragment-PathStorage` interface to allow FPI-specific logic and the logic that manages the storage of the FPI mappings to vary independently. The `XMLFragmentSummary` interface represents the $fragmentSummary$ data structure in the $(fragmentId, fragmentSummary)$ tuple. It has two methods: `evaluate()` and `serialize()`. The `evaluate()` method takes the query string as an argument, and returns a boolean value indicating whether a non-empty result set could be returned if the given query is evaluated on the XML fragment summarised by the `XMLFragmentSummary` object. The `serialize()` method converts the `XMLFrag-mentSummary` object to an array of bytes for storage. Since calculating XML fragment summaries is beyond the scope of this work, the prototype uses the `NullXMLFragmentSummary` class, which always returns `true` from its `evaluate()` method, regardless of the value of the argument. The `serialize()` method of the `NullXMLFragmentSummary` always returns an empty array.

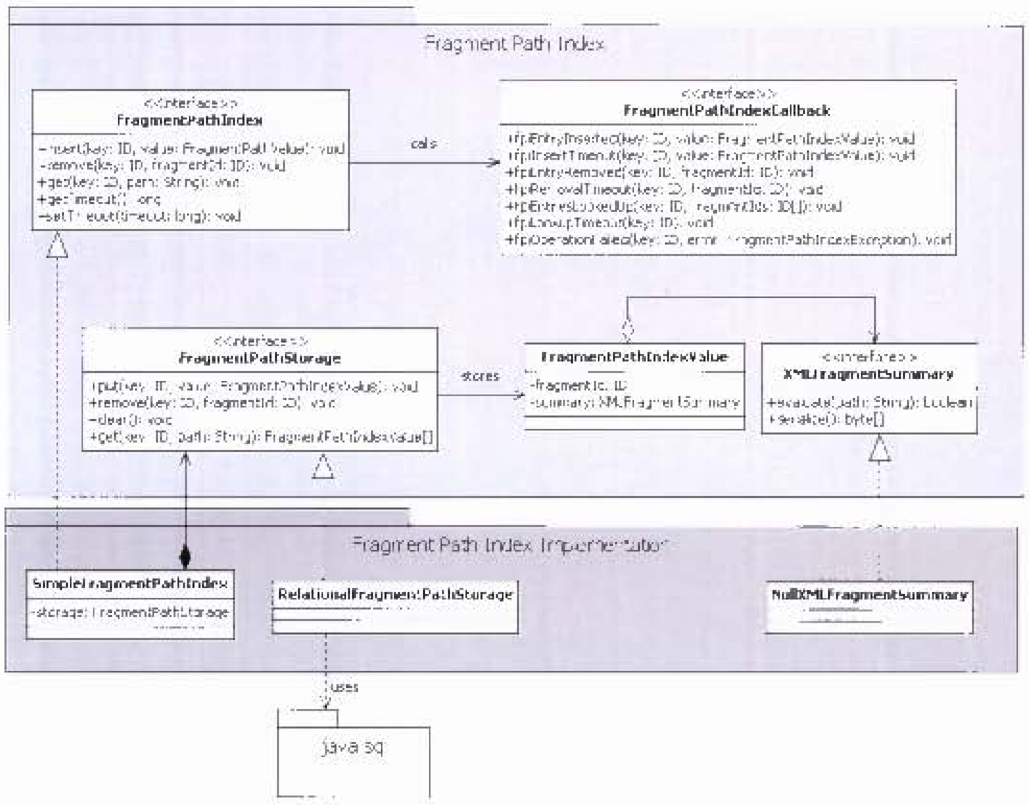

Fragment Path Index
<<interface>>
FragmentPathIndex
+insert(key: ID, value: FragmentPathIndexValue): void
+remove(key: ID, fragmentId: ID): void
+get(key: ID, path: String): void
+getTimeout(): long
+setTimeout(timeout: long): void
calls
<<interface>>
FragmentPathIndexCallback
+fpiEntryInserted(key: ID, value: FragmentPathIndexValue): void
+fpiInsertTimeout(key: ID, value: FragmentPathIndexValue): void
+fpiEntryRemoved(key: ID, fragmentId: ID): void
+fpiRemovalTimeout(key: ID, fragmentId: ID): void
+fpiEntriesLookedUp(key: ID, fragmentIds: ID[]): void
+fpiLookupTimeout(key: ID): void
+fpiOperationFailed(key: ID, error: FragmentPathIndexException): void
<<interface>>
FragmentPathStorage
+put(key: ID, value: FragmentPathIndexValue): void
+remove(key: ID, fragmentId: ID): void
+clear(): void
+get(key: ID, path: String): FragmentPathIndexValue[]
stores
FragmentPathIndexValue
-fragmentId: ID
-summary: XMLFragmentSummary
<<interface>>
XMLFragmentSummary
+evaluate(path: String): boolean
+serialize(): byte[]
Fragment Path Index Implementation
SimpleFragmentPathIndex
-storage: FragmentPathStorage
RelationalFragmentPathStorage
NullXMLFragmentSummary
uses
java.sql

Figure A.6: UML class diagram of the Fragment Path Index

A.6 The Peer Discover and Pinger Components

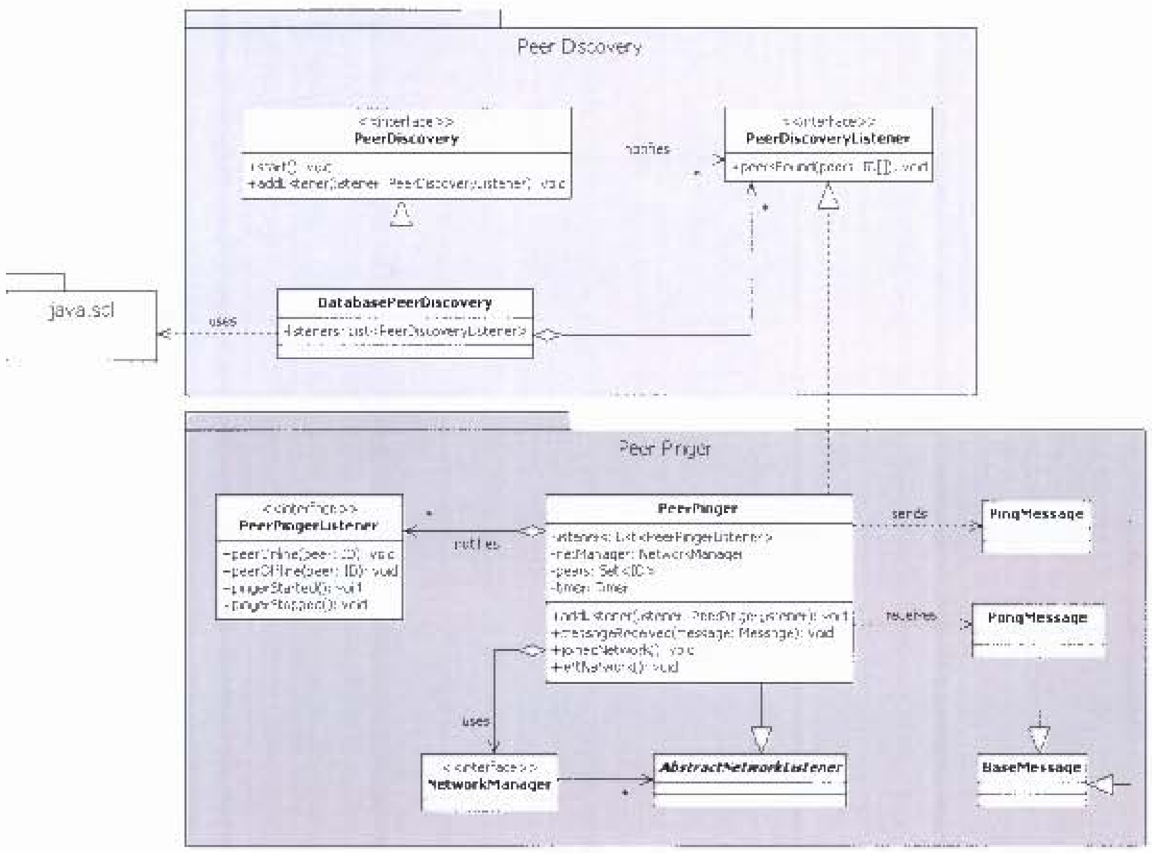


Figure A.7: UML class diagram of the Peer Discovery and Pinger components

The `PeerDiscover` interface represents the peer discovery mechanism. It informs the system of discovered peers by calling the `peersFound()` method in the `PeerDiscoveryListener` interface. The `DatabasePeerDiscovery` class is a mock implementation of the `PeerDiscovery` mechanism that maintains a list of peer IDs in a database.

The `PeerPinger` class implements the `PeerDiscoveryListener` interface. It informs the system of a peer's online/offline status by invoking the `peerOnline()` and `peerOffline()` methods in the `PeerPingerListener` interface, respectively.

A.7 The Replication Manager

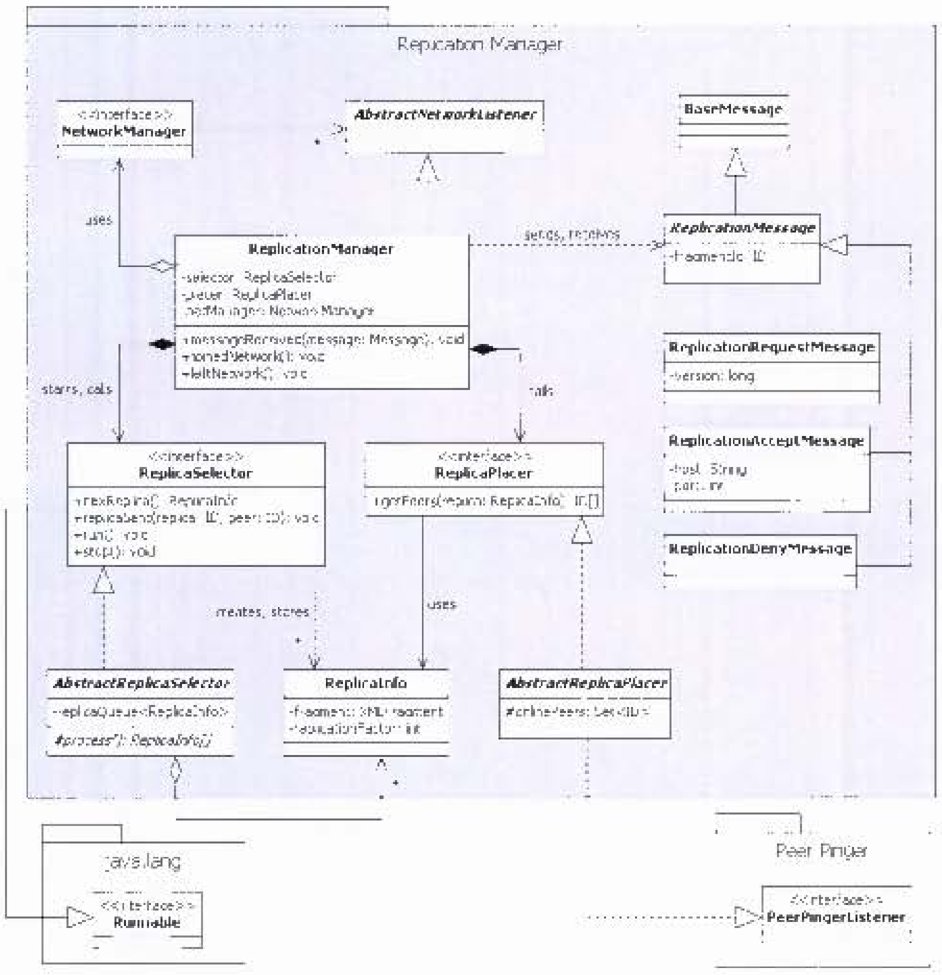


Figure A.8: UML class diagram of the Replication Manager component

The `AbstractReplicaSelector` class implements the `ReplicaSelector` interface and maintains a replica queue onto which fragments selected for replication are placed. All the replica selection algorithms discussed in section 6.1.2 are implemented as `AbstractReplicaSelector` subclasses that override the abstract `process()` method. Fragments selected for replication are represented as `ReplicaInfo` objects, each holding a reference to the selected fragment and a number indicating how many copies of the selected fragment should be created in the network. These `ReplicaInfo` objects are returned by the `process()` methods implemented by the `AbstractReplicaSelector` subclasses and are added to the replica queue in the `AbstractReplicaSelector` class.

`ReplicaInfo` objects are retrieved from the queue by the `ReplicationManager` class using the `nextReplica()` method in the `ReplicaSelector`. When a replica has been successfully transferred to a remote peer, the `ReplicationManager` calls the `replicaS-ent()` method to inform the `ReplicaSelector` where the replica was sent.

The `AbstractReplicaPlacer` class implements the `ReplicaPlacer` interface and stores the list of online peers from which the replica placement algorithms select peers to which replicas should be sent. The replica placement algorithms discussed in section 6.1.3 are implemented as `AbstractReplicaPlacer` subclasses that implement the `getPeers()` method in the `ReplicaPlacer` interface. This method is called by the `ReplicationManager` to retrieve the IDs of the peers on which a given replica should be placed.

The `AbstractReplicaPlacer` receives notifications of online and offline peers from the `PeerPinger` class in figure A.7 by implementing the `PeerPingerListener` interface.

Appendix B

Procedures Used During Experimental Evaluation

This section presents the procedures and algorithms used for measurement gathering during the experimental evaluation in chapter 8.

```
candidateQueries := new list

foreach xmlDoc in xmlDocCollection do:
    queries := generate xmlDoc XPath queries
    foreach query in queries do:
        resultSet := evaluate query on xmlDoc
        if resultSet is not empty do:
            add query to candidateQueries
        end if
    end foreach
end foreach

n := no. queries to pose on system
for i := 0 to n do:
    query := select random query from candidateQueries
    queryTime := generate time using Poisson process
    output queryTime, query and submitting peer
end for
```

Figure B.1: Pseudocode for generating the set of test queries

```
<query>             ::= '/' <element-name> <step>*
<step>              ::= <axis> <element-name> <predicate-list>?
<axis>              ::= '/' | '//'
<predicate-list>    ::= '[' [ <predicate> ' or ' ]* <predicate> ']'
<predicate>         ::= <element-predicate> | <attr-predicate>
<element-predicate> ::= <element-name> '=' <alphanumeric-string>
<attr-predicate>    ::= '@' <attr-name> '=' <alphanumeric-string>
<element-name>      ::= <alphanumeric-string>
<attribute-name>    ::= <alphanumeric-string>
```

Figure B.2: BNF notation for the form of the XPath queries used in the experiments

```
procedure fingerprint(Element element):
    string := serialize(element)
    return md5hash(string)
end procedure

procedure serialize(Element element):
    string := '$' + element.getNodeName()

    // Process element attributes
    string := string + serializeElementAttributes(element)

    // Process child elements and text nodes
    nodeList := element.getChildNodes()
    if nodeList.length() > 0:
        string := string + "<"
        childrenProcessed := 0
        foreach node in nodeList:
            if node is an element node:
                if childrenProcessed > 0:
                    string := string + ","
                end if
                string := string + serialize(node)
                childrenProcessed := childrenProcessed + 1
            else if node is a text node:
                text := node.getNodeValue().trim()
                if text.length() > 0:
                    if childrenProcessed > 0:
                        string := string + ","
                    end if
                    string := string + text
                    childrenProcessed := childrenProcessed + 1
                end if
            end if
        end foreach
        string := string + ">"
    end if

    return string
end procedure
```

Figure B.3: The XML tree fingerprinting algorithm used for comparing XML trees as described in section 8.2.1.2

```
procedure serializeElementAttributes(Element element):
    string := ""

    attributesList := element.getAttributes()
    if attributesList.length() > 0:
        string := string + "{"
        for i := 0 to attributesList.length():
            attribute := attributesList[i]
            string := string + attribute.getName() + "="
            string := string + attribute.getValue()
            if i < attributesList.length() - 1:
                string := string + ","
            end if
        end foreach
        string := string + "}"
    end if

    return string
end procedure
```

Figure B.4: Serializing XML element attributes to strings for fingerprinting in figure B.3

Appendix C

List of Acronyms

AIO	Access Information Object
AURP	Average Uptime Replica Placement
CRP	Counter-based Replica Placement
CSRP	Counter-State Replica Placement
DHT	Distributed Hash Table
DIN	Document Information Node
FIN	Fragment Information Node
FLC	Fragment Location Catalogue
FPI	Fragment Path Index
MFA	Most-Frequently Accessed Replica Selection
MFA-F	Most-Frequently Accessed Replica Selection Without Fragmentation
MFA+F	Most-Frequently Accessed Replica Selection With Fragmentation
MRA	Most-Recently Accessed Replica Selection
MRA-F	Most-Recently Accessed Replica Selection Without Fragmentation
MRA+F	Most-Recently Accessed Replica Selection With Fragmentation
P2P	Peer-to-Peer

RRP	Random Replica Placement
RRS	Random Replica Selection
RRS-F	Random Replica Selection Without Fragmentation
RRS+F	Random Replica Selection With Fragmentation
XML	eXtensible Markup Language

Bibliography

[1] Aboulnaga, A., Alameldeen, A. R., and Naughton, J. F. Estimating the selectivity of xml path expressions for internet scale applications. In *Proceedings of the 27th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2001), Morgan Kaufmann Publishers Inc., pp. 591–600.

[2] Al-Ekram, R., Adma, A., and Baysal, O. diffx: an algorithm to detect changes in multi-version xml documents. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research* (2005), IBM Press, pp. 1–11.

[3] Androutsellis-Theotokis, S., and Spinellis, D. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv. 36*, 4 (2004), 335–371.

[4] Bayer, R., and McCreight, E. Organization and maintenance of large ordered indexes. *Acta Informatica 1* (1972), 173–189.

[5] Bernstein, P., Giunchiglia, F., Kementsietsidis, A., Mylopoulos, J., Serafini, L., and Zaihrayeu, I. Data management for peer-to-peer computing: A vision. In *Proceedings of the Workshop on the Web and Databases (WebDB)* (2002).

[6] Bhagwan, R., Savage, S., and Voelker, G. M. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)* (2003).

[7] Bhagwan, R., Tati, K., Cheng, Y.-C., Savage, S., and Voelker, G. M. Total recall: system support for automated availability management. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 25–25.

[8] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[9] Bonifati, A., Matrangolo, U., Cuzzocrea, A., and Jain, M. Xpath lookup queries in p2p networks. In *Proceedings of the 6th annual ACM international workshop on Web information and data management* (New York, NY, USA, 2004), ACM Press, pp. 48–55.

[10] Cai, M., Chervenak, A., and Frank, M. A peer-to-peer replica location service based on a distributed hash table. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 56.

[11] CARLSSON, B., AND GUSTAVSSON, R. The rise and fall of napster - an evolutionary approach. In *Proceedings of the 6th International Computer Science Conference on Active Media Technology* (London, UK, 2001), Springer-Verlag, pp. 347–354.

[12] CERI, S., AND PELAGATTI, G. *Distributed Databases: Principles and Systems*. McGraw-Hill, Inc., USA, 1984.

[13] CHAWATHE, S. S., AND GARCIA-MOLINA, H. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1997), ACM, pp. 26–37.

[14] CHAWATHE, S. S., RAJARAMAN, A., GARCIA-MOLINA, H., AND WIDOM, J. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1996), ACM, pp. 493–504.

[15] CHEN, Y., MADRIA, S., AND BHOWMICK, S. Diffxml: Change detection in xml data. *Database Systems for Advanced Applications 2973/2004* (February 2004), 289–301.

[16] CHEN, Z., JAGADISH, H. V., KORN, F., KOUDAS, N., MUTHUKRISHNAN, S., NG, R. T., AND SRIVASTAVA, D. Counting twig matches in a tree. In *Proceedings of the 17th International Conference on Data Engineering* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 595–604.

[17] CHERVENAK, A., DEELMAN, E., FOSTER, I., GUY, L., HOSCHEK, W., IAMNITCHI, A., KESSELMAN, C., KUNSZT, P., RIPEANU, M., SCHWARTZKOPF, B., STOCKINGER, H., STOCKINGER, K., AND TIERNEY, B. Giggle: a framework for constructing scalable replica location services. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 1–17.

[18] CHIEN, S.-Y., TSOTRAS, V. J., AND ZANIOLO, C. Copy-based versus edit-based version management schemes for structured documents. In *Eleventh International Workshop on Research Issues in Data Engineering on Document Management for Data Intensive Business and Scientific Applications* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 95–102.

[19] CHIEN, S.-Y., TSOTRAS, V. J., AND ZANIOLO, C. Version management of xml documents. In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases* (London, UK, 2001), Springer-Verlag, pp. 184–200.

[20] CLARKE, I., MILLER, S. G., HONG, T. W., SANDBERG, O., AND WILEY, B. Protecting free expression online with freenet. *IEEE Internet Computing 6*, 1 (2002), 40–49.

[21] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM Press, pp. 202–215.

[22] DABEK, F., ZHAO, B., DRUSCHEL, P., KUBIATOWICZ, J., AND STOICA, I. Towards a common api for structured peer-to-peer overlays. *Lecture Notes in Computer Science 2735* (2003), 33–44.

[23] DATTA, A., HAUSWIRTH, M., AND ABERER, K. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems* (Washington, DC, USA, 2003), IEEE Computer Society, p. 76.

[24] DOUCEUR, J., AND WATTENHOFER, R. Large-scale simulation of replica placement algorithms for a serverless distributed file system. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)* (Washington, DC, USA, 2001), IEEE Computer Society, p. 311.

[25] DOUCEUR, J., AND WATTENHOFER, R. Optimizing file availability in a secure serverless distributed file system. *20th IEEE Symposium on Reliable Distributed Systems 00* (2001), 4–13.

[26] FLESCA, S., MANCO, G., MASCIARI, E., AND PONTIERI, L. Fast detection of xml structural similarity. *IEEE Transactions on Knowledge and Data Engineering 17*, 2 (2005), 160–175. Student Member-Andrea Pugliese.

[27] FOSTER, J. N., GREENWALD, M. B., KIRKEGAARD, C., PIERCE, B. C., AND SCHMITT, A. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences 73*, 4 (2007), 669–689.

[28] FRANCONI, E., KUPER, G., LOPATENKO, A., AND ZAIHRAYEU, I. Queries and updates in the codb peer to peer database system. In *Proceeedings of the 30th International Conference on Very Large Databases (VLDB)* (2004), pp. 1277–1280.

[29] FREIRE, J., HARITSA, J. R., RAMANATH, M., ROY, P., AND SIMÉON, J. Statix: making xml count. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2002), ACM, pp. 181–191.

[30] GALANIS, L., WANG, Y., JEFFREY, S. R., AND DEWITT, D. J. Locating data sources in large distributed systems. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)* (Berlin, Germany, 2003).

[31] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[32] GIFFORD, D. K. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles* (New York, NY, USA, 1979), ACM, pp. 150–162.

[33] GOLDMAN, R., AND WIDOM, J. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)* (Athens, Greece, August 1997), Morgan Kaufmann, pp. 436–445.

[34] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1996), ACM, pp. 173–182.

[35] GRIBBLE, S., HALEVY, A., IVES, Z., RODRIG, M., AND SUCIU, D. What can databases do for peer-to-peer? In *Proceedings of the 4th International Workshop on the Web and Databases (WebDB)* (2001).

[36] HALEVY, A. Y., IVES, Z. G., MADHAVAN, J., MORK, P., SUCIU, D., AND TATARINOV, I. The piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering 16*, 7 (2004), 787–798.

[37] HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I. Querying the internet with pier. In *Proceedings of the 29th international conference on Very large data bases* (2003), VLDB Endowment, pp. 321–332.

[38] JACOB, J., SACHDE, A., AND CHAKRAVARTHY, S. Cx-diff: a change detection algorithm for xml content and change visualization for webvigil. *Data & Knowledge Engineering 52*, 2 (2005), 209–230.

[39] JIN, H., WANG, C., AND CHEN, H. Boundary chord: A novel peer-to-peer algorithm for replica location mechanism in grid environment. In *Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 262–267.

[40] JUNQUEIRA, F., BHAGWAN, R., MARZULLO, K., SAVAGE, S., AND VOELKER, G. M. The phoenix recovery system: rebuilding from the ashes of an internet catastrophe. In *Proceedings of the 9th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2003), USENIX Association, pp. 73–78.

[41] KEMENTSIETSIDIS, A., ARENAS, M., AND MILLER, R. J. Mapping data in peer-to-peer systems: semantics and algorithmic issues. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2003), ACM Press, pp. 325–336.

[42] KOLONIARI, G., PETRAKIS, Y., AND PITOURA, E. Content-based overlay networks for xml peers based on multi-level bloom filters. In *Proceedings of the International VLBB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)* (2003).

[43] KOLONIARI, G., AND PITOURA, E. Peer-to-peer management of xml data: issues and research challenges. *SIGMOD Rec. 34*, 2 (2005), 6–17.

[44] KOLONIARI, G., AND PITOURI, E. Coontent-based routing of path queries in peer-to-peer systems. *Lecture Notes in Computer Science 2992* (2004), 29–47.

[45] LEE, K.-H., CHOY, Y.-C., AND CHO, S.-B. An efficient algorithm to compute differences between structured documents. *IEEE Transactions on Knowledge and Data Engineering 16*, 8 (2004), 965–979.

[46] LINDHOLM, T. Xml three-way merge as a reconciliation engine for mobile data. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access* (New York, NY, USA, 2003), ACM, pp. 93–97.

[47] LINDHOLM, T., KANGASHARJU, J., AND TARKOMA, S. Fast and simple xml tree differencing by sequence alignment. In *Proceedings of the 2006 ACM symposium on Document engineering* (New York, NY, USA, 2006), ACM, pp. 75–84.

[48] MARIAN, A. Detecting changes in xml documents. In *Proceedings of the 18th International Conference on Data Engineering* (Washington, DC, USA, 2002), IEEE Computer Society, p. 41.

[49] MARIAN, A., ABITEBOUL, S., COBENA, G., AND MIGNET, L. Change-centric management of versions in an xml warehouse. In *Proceedings of the 27th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2001), Morgan Kaufmann Publishers Inc., pp. 581–590.

[50] MASCOLO, C., CAPRA, L., ZACHARIADIS, S., AND EMMERICH, W. Xmiddle: A data-sharing middleware for mobile computing. *Wireless Personal Communications 21*, 1 (2002), 77–103.

[51] NG, W. S., OOI, B. C., TAN, K.-L., AND ZHOU, A. Peerdb: A p2p-based system for distributed data sharing. In *Proceedings of the 19th International Conference on Data Engineering* (2003), pp. 633–644.

[52] OOI, B. C., SHU, Y., AND TAN, K. L. Web technologies and applications. *DB-Enabled Peers for Managing Distributed Data* (2003), 596–596.

[53] POLYZOTIS, N., AND GAROFALAKIS, M. Structure and value synopses for xml data graphs. In *Proceedings of the 28th international conference on Very Large Data Bases* (2002), VLDB Endowment, pp. 466–477.

[54] POLYZOTIS, N., AND GAROFALAKIS, M. Xcluster synopses for structured xml content. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)* (Washington, DC, USA, 2006), IEEE Computer Society, p. 63.

[55] RANGANATHAN, K., IAMNITCHI, A., AND FOSTER, I. Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2002), IEEE Computer Society, p. 376.

[56] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)* (New York, NY, USA, 2001), ACM Press, pp. 161–172.

[57] RIPEANU, M., AND FOSTER, I. A decentralized, adaptive replica location mechanism. In *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 24.

[58] RISSE, T., AND KNEZEVIC, P. A self-organizing data store for large scale distributed infrastructures. *21st International Conference on Data Engineering Workshops (ICDEW) 0* (2005), 1211.

[59] RODRÍGUEZ-GIANOLLI, P., KEMENTSIETSIDIS, A., GARZETTI, M., KIRINGA, I., JIANG, L., MASUD, M., MILLER, R. J., AND MYLOPOULOS, J. Data sharing in the hyperion peer database system. In *Proceedings of the 31st international conference on Very large data bases* (2005), VLDB Endowment, pp. 1291–1294.

[60] RÖNNAU, S., SCHEFFCZYK, J., AND BORGHOFF, U. M. Towards xml version control of office documents. In *Proceedings of the 2005 ACM symposium on Document engineering* (New York, NY, USA, 2005), ACM, pp. 10–19.

[61] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science 2218* (2001).

[62] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), ACM Press, pp. 188–201.

[63] SAROIU, S., GUMMADI, K. P., AND GRIBBLE, S. D. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems 9*, 2 (2003), 170–184.

[64] SARTIANI, C. A query algebra for xml p2p databases. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters* (New York, NY, USA, 2004), ACM Press, pp. 258–259.

[65] SARTIANI, C., MANGHI, P., GHELLI, G., AND CONFORTI, G. Xpeer : A self-organizing xml p2p database system. *Lecture Notes in Computer Science 3268* (Jan 2004), 456–465.

[66] SILBERSCHATZ, A., KORTH, H. F., AND SUDARSHAN, S. *Database System Concepts*, 4th ed. McGraw-Hill, 2002.

[67] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2001), ACM Press, pp. 149–160.

[68] WANG, W., JIANG, H., LU, H., AND YU, J. X. Bloom histogram: path selectivity estimation for xml data with updates. In *Proceedings of the Thirtieth international conference on Very large data bases* (2004), VLDB Endowment, pp. 240–251.

[69] WANG, Y., DEWITT, D. J., AND CAI, J.-Y. X-diff: An effective change detection algorithm for xml documents. *icde 00* (2003), 519.

[70] WANG, Z., DAS, S. K., KUMAR, M., AND SHEN, H. Update propagation through replica chain in decentralized and unstructured p2p systems. In *Proceedings of the Fourth International Conference on Peer-to-Peer Computing (P2P'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 64–71.

[71] WATANABE, T., HARA, T., KIDO, Y., AND NISHIO, S. An update propagation strategy for delay reduction and node failure tolerance in peer-to-peer networks. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 103–108.

[72] Freepastry website. http://freepastry.rice.edu.

[73] Freepastry tutorial. http://freepastry.org/FreePastry/tutorial/index.html.

[74] Globus toolkit website. http://www.globus.org/toolkit.

[75] Gnutella protocol specification wiki. http://gnutella-specs.rakjar.de/index.php/Main_Page.

[76] Inet topology generator. http://topology.eecs.umich.edu/inet/.

[77] J2se. http://java.sun.com/javase/.

[78] Linux dc++ website. http://linuxdcpp.berlios.de.

[79] Modelnet howto. http://modelnet.ucsd.edu/howto.html.

[80] Modelnet. http://modelnet.ucsd.edu.

[81] The mondial database. http://www.dbis.informatik.uni-goettingen.de/Mondial/.

[82] National vulnerability database cve feeds. http://nvd.nist.gov/download.cfm.

[83] Rfc 1952: Gzip file format specification version 4.3. http://www.ietf.org/rfc/rfc1952.txt.

[84] Rfc 3174: Us secure hash algorithm 1 (sha1). http://www.ietf.org/rfc/rfc3174.txt.

[85] Rfc 4648: The base16, base32, and base64 data encodings. http://tools.ietf.org/html/rfc4648.

[86] Rfc 791: Internet protocol. http://www.ietf.org/rfc/rfc791.txt.

[87] Simple api for xml (sax). http://www.saxproject.org/.

[88] Soap version 1.2 primer. http://www.w3.org/TR/soap12-part0.

[89] Uniprot: The universal protein resource. http://www.uniprot.org.

[90] UW XML data repository. http://www.cs.washington.edu/research/xmldatasets/.

[91] World wide web consortium (w3c). http://www.w3.org.

[92] Web services description language (wsdl). http://www.w3.org/2002/ws/desc.

[93] Extensible markup language (xml). http://www.w3.org/XML.

[94] Xml binary characterization working group test data. http://www.w3.org/XML/Binary/2005/03/test-data.

[95] XML Document Object Model (DOM). http://www.w3.org/DOM.

[96] XML document type definitions (DTD). http://www.w3.org/TR/2006/REC-xml-20060816/#dt-doctype.

[97] XML schema. http://www.w3.org/XML/Schema.

[98] Xml path language (xpath) version 1.0. http://www.w3.org/TR/xpath.

[99] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.