# PEER TO PEER CATALOG MANAGER FOR NETTRAVELER MIDDLEWARE SYSTEM

By

*Oliver Antonio Moreno-Puello*

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER ENGINEERING

University of Puerto Rico
Mayagüez Campus
2007

Approved by:

| | |
|---|---|
| Bienvenido Vélez-Rivera, Ph.D.<br>Member, Graduate Committee | Date |
| Pedro Rivera-Vega, Ph.D.<br>Member, Graduate Committee | Date |
| Manuel Rodríguez-Martínez, Ph.D.<br>President, Graduate Committee | Date |
| Oswald N. C. Uwakweh, Ph.D.<br>Representative of Graduate Studies | Date |
| Isidoro Couvertier-Reyes, Ph.D.<br>Chairperson of the Department | Date |

# ABSTRACT

# PEER TO PEER CATALOG MANAGER FOR NETTRAVELER MIDDLEWARE SYSTEM

By

*Oliver Antonio Moreno-Puello*

Distributed and mobile database systems hold the promise of interconnecting mobile devices, workstations, and enterprise servers to share data and computational resources. In these environments, query optimizers will be as good as the metadata they use in the optimization process. These metadata are kept in the catalog (data dictionary) of the system. Several approaches currently in use for metadata management assume that the catalog is either: a) centralized, b) fully replicated system-wide, or c) distributed among various well-known sites. However, the dynamic nature of mobile and wide-area networks results in constant changes to the metadata, as well as changes in the sites holding such metadata. Hence, it is of paramount importance to have a catalog system that dynamically adapts to these changes. This work presents a decentralized framework for metadata management that copes with this situation. Our approach is based on a Peer-to-Peer (P2P) catalog management organization, which uses consistent hashing as the mechanism to locate metadata objects throughout the system. Each metadata object is associated with a well-known key. Our framework makes the system more scalable since there is no central metadata repository, and metadata can be found through an efficient search mechanism. It also provides efficient mechanisms to handle the arrival and departure of hosts and the metadata associated with these hosts.

# RESUMEN

# SISTEMA PEER TO PEER PARA EL MANEJO DEL CATALOGO EN EL SISTEMA TIPO MIDDLEWARE NETTRAVELER

Por

*Oliver Antonio Moreno-Puello*

Los sistemas de bases de datos móviles distribuidos presentan el potencial de interconectar una amplia gama de dispositivos tales como móviles, computadores de alto rendimiento y servidores empresariales con el fin de intercambiar datos y/o recursos computacionales. Los optimizadores de consultas de estos sistemas dependen en gran medida de la calidad de la metadata disponible. Esta metadata es almacenada dentro del catalogo del sistema, siendo los modelos mas comunes para este almacenamiento: a) el modelo centralizado, b) completamente redundante y c) distribuido entre diferentes sitios conocidos. Sin embargo, debido a la naturaleza dinámica de las redes inalámbricas como las de hoy en día, la metadata almacenada esta sujeta a cambios constantes, los cuales de una manera u otra deben verse reflejados directamente en el catalogo. Es aquí donde se centra la gran importancia de desarrollar un sistema de catalogo capaz de manejar estos cambios con la finalidad de mantener la integridad de la metadata en todo momento. Esta investigación propone un sistema descentralizado para el manejo de metadata como solución a este problema. Esta solución se basa en un esquema peer-to-peer (P2P) para el manejo del catalogo, haciendo uso de estructuras de tipo consistent-hash como mecanismo para localizar la metadata a través del sistema. A cada ítem de metadata le es asignado código en particular para agilizar su localización. El hecho de utilizar una arquitectura descentralizada disminuye el riesgo de perder toda la información debido a fallas, debido a que en un modelo centralizado, como su nombre lo indica, toda la información yace en un mismo lugar. Nuestra arquitectura incorpora un método de búsqueda eficiente no exhaustivo para localizar cualquier ítem de metadata dentro del catalogo. Finalmente, esta solución provee

mecanismos para el manejo eficiente de entradas y salidas de nodos en el sistema.

To parents Elsy† and Raimundo, brothers Herlin and Didier.

# ACKNOWLEDGMENTS

First of all, I will like to thank God for giving me the strength and courage to complete this thesis. I will also like to thank to my advisor Dr. Manuel Rodríguez Martínez for giving me the opportunity to work with him, for his support and especially for his friendship during these years. Also, I want to thank Dr. Bienvenido Vélez, Dr. Pedro Rivera and Dr. Oswald Uwakweh for serving on my graduate committee and for reviewing my work. To my brothers, Herlin and Didier thank for you support and motivation. I deeply thank to my friends of Advance Data Management Group - ADMG (Eliana, Elliot, René, Juan, Abdiel, Pablo, Juddy and Oswaldo) for their academic contributions and their friendship. My very special thanks to Angel and Yuji for their friendship and constantly support.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# Introduction

The emergence of wireless networks and powerful mobile computers pose a challenge to the assumptions upon which existing distributed databases are implemented. A Distributed Database System (DDBS) is a collection of multiple Database Management Sytems (DBMS), logically interrelated, distributed geographically and connected via a computer network. Commonly, these DBMS are heterogeneous, with software systems from different vendors, running on computer with different architectures and different operating systems.

Nowadays, people wish to store and share data not only on servers but also an a wide range of mobile devices, such a laptops, PDAs and smart-phones. Figure 1.1 depicts some of the top-notch mobile devices on the market. Peer-to-Peer (P2P) systems such as BitTorreent and Kazza have helped cement a culture of loosely coupled interactions and rapidly changing content. Hence, users usually add or remove data from their systems without any restrictions or central controlling authority. As a result, the contents on a given source are unpredictable and subject to constant change. This change also applies to the locations and environment from which users interact with others in the network. Wireless environments enable computers to move to different geographical locations and continue sharing their data collections. But the connectivity information for such computers can change because the owners might choose to re-connect to the local wireless networks, and this often results in a change of network address. Despite the promise of MobileIP for allowing

seamless packet forwarding to mobile hosts, it is most often the case that people simply change IP addresses as they switch networks. In addition, data sources go online and off-line because the users voluntarily join and leave the network. As a result, the availability of a host and its associated data collections is not guaranteed.

Figure 1.1: Modern mobile devices

In order to address this problem, middleware systems for data integration have been proposed as feasible solutions to form federations of cooperative sites [1]. But these systems need to be dynamically adaptive to the changes in the context of execution and must take advantage of contextual information (such as network bandwidth, CPU usage, user location and user activity) of mobile devices. The NetTraveler system [1] is designed to support data management over dynamic Wide-Area Networks (WAN). Three specific issues are addressed with NetTraveler:

1. Context-aware query processing.
2. Server-side and Client-side query recovery.
3. System self-configuration.

By attacking these three specific areas, NetTraveler is intended to deal with query execution in mobile devices taking into consideration the nature of host for query processing purpose.

In lieu of this new working environment, it is necessary to reconceptualize query process-

esing architectures that rely on well-behaved environments. One important aspect that must be considered is the management of metadata throughtout the system. This metadata is used by query optimizers and query processing modules to select the sites on which relational operators will be scheduled for execution. These metadata include descriptions about local databases, tables, columns, data types, available query operators, description of physical resources (e.g., CPU speed, memory size, current load), access privilegies, and so on. Keeping these matadata up-to-date and readily available is crucial to efficiently execute submitted queries. Traditionally, each participating site in a distributed system keeps a catalog with information about its data collections, use restrictions, performance characteristics, and other policies required to participate in query processing. These catalogs can be distributed and located at each participating host, or there can be a set of well-known sites that store the catalog. Several approaches currently in use for metadata management assume that the catalog is either: a) centralized, b) fully replicated system-wide, or c) distributed among various well-known sites. However, the dynamic nature of mobile and wide-area networks results in constant changes to the metadata, as well as changes in the sites holding such metadata

Clearly, these existing catalog (metadata) management schemes are simply inadequate for the new large-scale and mobile environments, for example the centralized approach is vulnerable to failures and bottlenecks of performance, the partitioned approach requires exhaustive searching protocol to among the system, finally the fully replicated approach can be prohibitive in terms of resource usage.

## 1.1 Problem Statement

The problem being addressed in this thesis is the development of a framework for the efficient storage and dissemination of metadata about the resources in a distributed database middleware system. Specially, we study the problem of building a distributed catalog system that can answer request for metadata posed by client applications. These metadata include tables, columns, users, query operators, computing resources (e.g., CPU speed, RAM, disk space), and so on, all of

which must be made available to a query optimizer and query execution engine in order to produce a data processing plan to answer a query posed by a user. Thus, when a client application submits a query request, the catalog system must find the metadata records that help the query optimizer and execution engine decide how to solve that query. Moreover, the catalog must operate in a de-centralized mode, where no central authority controls content nor becomes a performance bottleneck. In addition, the catalog system must adapt to constant changes in the metadata, and also accommodate for the entrance of new data sources into the middleware system. Likewise, the catalog system must handle the departure of existing data sources from the middleware system.

## 1.2 Proposed Solution

This work proposes a *Peer-to-Peer (P2P) Catalog Manager Scheme* that takes in consideration rapid changes in the data contents and the de-centralized nature of current networked systems. The solution is based on the Distributed Hash Table (DHT) algorithm Chord presented in [2]. The reason behind our decision to use Chord, is that it provides a decentralized lookup system that matches our requirements for next-generation catalog systems for middleware environments. We foresee that centralized solutions are simply unfeasible in future data integration environments. By using Chord our approach is far more scalable since the metadata lookup operations are distributed among the sites in the system, and the processing load is naturally balanced to prevent hot spots. We pair the Chord lookup functionality with a metadata cache manager, designed to speed up the metadata lookup process by exploiting frequent access to related metadata. In our approach, the percentage of queries served in the presence of failures is comparable to the ideal scenario of a fully replicated catalog approach, and much better than the centralized catalog and partitioned catalog approaches used elsewhere. The improvements in the response time of the system, introduced by using data caching techniques, allows us to stabilize the response time metric and offset the overhead inherent in having the catalog distributed among many sites. Our results show that without the usage of data caching mechanisms response time suffers from a nearly exponential growth as the number of clients in the system increases.

## 1.3 Objectives of this Thesis

The main goals of this thesis are:

- Design and implementation of a fully de-centralized catalog management system for metadata dissemination and resource location in the NetTraveler Middleware System.
- Provide a deterministic metadata lookup service. We define deterministic metadata lookup as a lookup operation that will assure that an answer for a metadata query will be given. Moreover, the same answer will be given during repeated calls, unless the metadata item is purged or updated.
- Provide an efficient mechanism to manage arrival and departures of nodes.
- Develop the necessary interface to allow easy interconnection with others components of NetTraveler Middlerware System.

## 1.4 Contributions

The major contributions of this thesis can be summarized as follows:

- A dynamic and scalable peer-to-peer architecture for the catalog management system that takes in consideration the rapid changes and decentralized nature of the current networks.
- A scheme for fault tolerance in the system by replicating the metadata over difference nodes. In presence of nodes failures, the metadata does not become unavailable, and can be found in alternative locations.
- Technique to support metadata caching in the system to improve the response time to future client queries.
- A experimental study that validates our proposed system. This study shows that our system has better availability than other approaches since the percentage of queries served in the presence of failures is above 30 percent better than the centralized catalog approach, and 20 percent (on average) better than the partitioned catalog approach. Also, this performance

is comparable to the ideal scenario of a fully replicated catalog system. This study also illustrates the need for metadata caching to reduce response time and offset the overhead incurred in having the metadata dispersed over several nodes.

## 1.5 Organization of this Thesis

The remainder of this thesis is structured as follows. Chapter two presents a survey of the more relevant works that served to develop this thesis, arranged in three main subtopics: i) Distributed Databases and Catalogs; ii) Middleware systems and the NetTraveler System; and iii) Peer-to-Peer Networks and Chord. Chapter three presents a overview of the proposed solution, the Catalog Manager. Chapter four discussed the implementation details of the proposed solution. Chapter five presents the results from experimental study done to evaluate the performance of the system. And finally, chapter six present a summary of contributions, conclusions and directions regarding future work.

# CHAPTER 2

# Related Work

This chapter presents relevant work in the areas that form the basis of this thesis. These areas include: Distributed Database Systems, Catalogs Manager, Database Middleware Systems, NetTraveler Middlerware System, Peer-to-Peer Systems and Chord.

## 2.1 Distributed Database Systems

In a Distributed Database System (DDBS), data is physically stored across several sites, and each site is typically managed by a DBMS capable of running independent of the other sites [2]. These sites do not share main memory or disk and are interconnected via computer network. The Distributed Database System uses the communication facilities of a computer network to offer to distributed users the same advantages obtained in a common Database System. Thus, from a user's perspective there is no logical difference between a Distributed Database System and a single Database System. The element responsible of this transparency to the users is the Distributed Database Management System (DDBMS) which is defined as the software system that control the management of the DDBS, dealing with all distribution concerns: concurrency, performance, recovery manager, distributed catalog management, distributed transactions and so on. The Figure 2.1, depicts a simplified view of Distributed Database System.

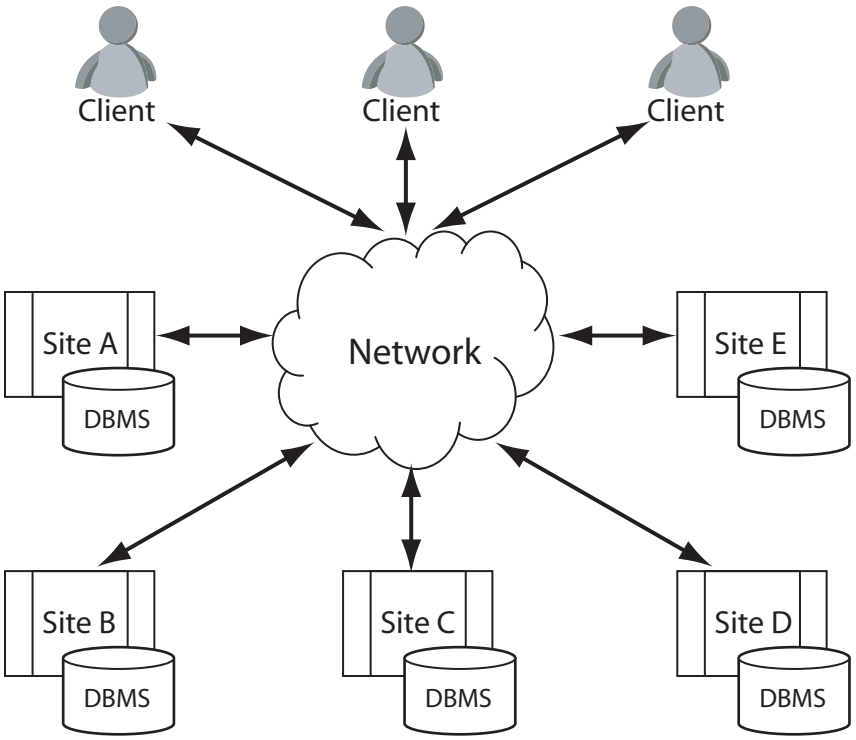



Figure 2.1: Typical Distributed Database Architecture

There are a number of advantages to using a Distributed Database System, among which the following stand out: reliability and availability, naturally modeling the organization structure, economy and easy growth. First, failures over a portion of the system (i.e. various local sites) does not make the system entirely inoperable. Additionally, the data of local sites can be replicated in different sites allowing access to replicated copies when failures are present. Distributed Database Systems reflect the organization structure. The data is organized across many departments in different independent databases. This structure of how the information is managed is itself a natural extension of DDBS. The cost of creating a DDBS with many small computer is more economic that create a single Database System in a powerful computer. Finally, new components are easily to added to a existing DDBS without affecting other components in the system.

But, there are also disadvantages such as system complexity, security and integrity control. Distributed Database Systems are more complex that a centralized DBMS. DDBS inherit the problems of DBMS approach and the problems of a distributed system. A efficient handling of these problems involved a increase of the complexity. In a centralized DBMS the control of the

data resides in a central site while in a DDBS this control are spread out over several separate sites. Also, the integrity control (data validation and consistency) inside a DDBS entails a high communication and processing costs that may be prohibitive.

Through the years, several functional DDBS prototypes has been developed of which we have to accentuate: R* [3], Distributed Ingres [4], Mariposa [5] and SDD-1 [6]. The R* system was developed between 1979 and 1984 at the IBM San Jose Research Laboratory. R* system was the evolution of the centralized DBMS System R, it was built with the follow key objectives: i) distribution transparency ii) site autonomy, and iii) good performance. Distributed Ingres was developed by University of California at Berkeley. It uses a master-slave approach (master Ingres and slave Ingres). The master Ingres is a process that run s at the site where the user's application and the slave Ingres is a process that runs on each site in the DDBS which the data must be accessed. Mariposa was developed by the University of California at Berkeley at 1995. It has a microeconomic model with bids and bidding process for query and storage optimization. Where bids represents computational load, store data or any computational cost. SDD-1 was developed by Computer Corporation of America (CCA) at 1978. SDD-1 is a collection of two servers: Transactions Modules (TM) and Data Modules(DM). The TMs and DMs are interconnected by the Reliable Network (RelNet) in a robust fashion.

The Distributed Database System can be classified to according of nature of DBMS that each site execute: homogenous and heterogenous. In homogenous distributed database systems, all sites run the same database management system software, are aware of one another, and agree to cooperate to solve a user query. In heterogeneous distributed database system, different sites can use different database management system software and different schemas. The sites may not be aware of one another, and may exist some limitation to cooperate to solve a specific query.

Query processing in centralized DBMS is focused in measuring the cost of particular strategy in terms of disk accesses. In Distributed Database Systems, not only the cost of disk accesses must be considered, but also we need take in consideration: the cost of transmission over the network (i.e accessing data from a remote site) and the advantage of parallel query processing

(i.e sites cooperate with parts of query to speed up the query answer). The basic challenge is the design and development of efficient query processing techniques that minimize the cost of communication and maximize the advantage of parallel processing.

Four principal techniques have been used for query optimization in Distributed Database Systems: *query shipping*, *data shipping*, *hybrid shipping* and *code shipping*. In *query shipping* the queries are submitted for computation at the sites where the data reside and the clients directly receive the results. *Query shipping* reduce the cost of network communication, shift the load of the clients to the servers but might cause the servers to become overloaded because the amount of data processing. This technique is useful when the system has clients with resource constraints, powerful servers and high cost in network communication. In contrast, the *data shipping* technique does not perform all query processing at the server. Instead, the query processing is mostly done at the client machines. All data is shipped from the servers to the clients and all query operators are executed at the client. The *data shipping* approach makes much better use of client resources than query shipping, but it increasing the amount of load in the communication network and fails to take advantage of server resources. A combination of two techniques mentioned previously is *hybrid shipping*. In this technique some query operator are performed in the client side and others sever side. *Hybrid shipping* permits exploit both client resources and server resources. Finally, *code shipping* technique which dynamically extends the capabilities of the remote sites by shipping new functions/database operations. Code Shipping enables hybrid shipping in system in which remote servers or clients do not have the capability to perform one or more query operators.

## 2.2 Catalog Management

The catalog in a database system contains the metadata that describe the objects that are of interesting to the database system itself. Example of this objects are: relations, query operators, views, indexes, statistics, users and so on. Since all object metadata reside into the catalog, its plays an important role in many aspects of database systems such as: query processing and optimization, access user control, and statistical information. For example, to process a query, the system must

first validate and parse the query using the catalog information, next its is presented to the query optimizer which generate possible execution plans to evaluate the query. Each possible execution plan have an associated cost which is computed with the help of the information stored in the catalog. The query optimizer will choose the plan with the low cost and proceeds to execute the query.

In a distributed database system, the catalog becomes a distributed database. The DDBS catalog entries must specify site(s) at which data is being stored, which is not necessary in the case of the catalog for a centralized DBMS. The catalog in a DDBS can be managed in different ways. Three basic approach are commonly used: i) centralized, ii) fully replicated and iii) partitioned.

- **Centralized catalog**: one site is exclusively dedicate to maintain the catalog and to serve catalog lookup requests. Although this approach offers simplicity of implementation and centralized access control, it has a disadvantage that is vulnerable to failures and bottlenecks of performance. As seen on figure 2.2, each site sends their request to a central and unique catalog server.

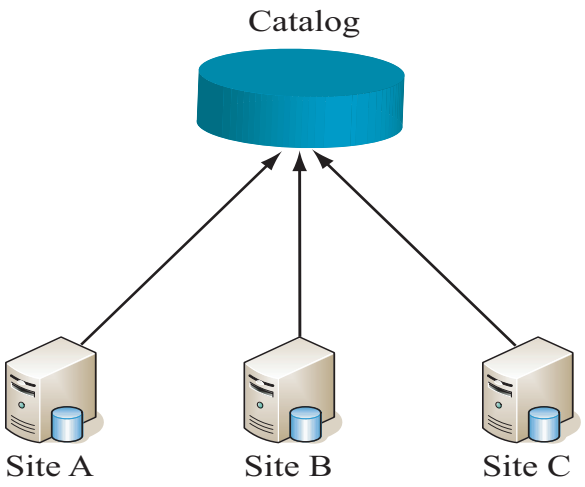



Figure 2.2: Centralized catalog

- **Fully replicated catalog**: each site maintains a local copy of the entire catalog. Although this approach is not vulnerable to single point of failure, the use of multiples copies implies that all copies must behave like a single-copy. Thus, a change in a local copy should be

propagated to the rest of copies. If the catalog size is large the cost of maintaining each copy of the catalog can be prohibitory. As seen in figure 2.3, each site has a copy of the entire catalog. Distributed Ingres [4] and DSS-1[6] are systems that implemented this kind of catalog.

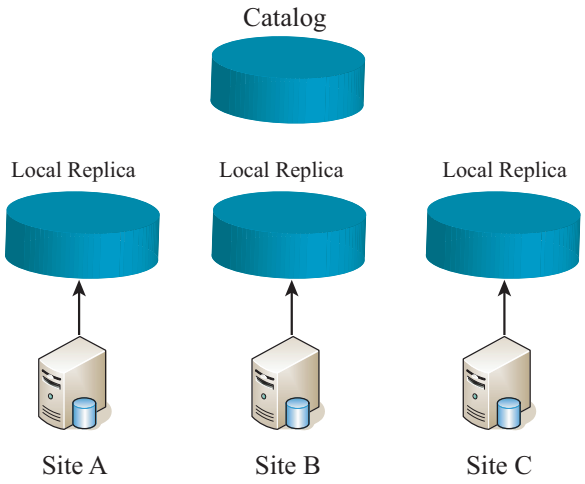



Figure 2.3: Fully replicated catalog

- **Partitioned catalog**: the entire catalog information is partitioned between all members of distributed database system. This approach does not need to propagate updates of each catalog to other sites, but queries over the catalog are more expensive, since finding a catalog entry requires communication between the nodes in the system. As seen in figure 2.4, each site maintains only a portion of catalog. R* [3] system is an example that implements this kind of catalog.

## 2.3 Database Middleware Systems

*Middleware* is a generic term referring to the layer of software whose purpose is to overcome the heterogeneity problem faced when different systems must communicate with each other to exchange data. The main goal of middleware is to mask the heterogeneity to users and to provide a standard programming interfaces and protocols to application programmers. With the same purpose Database Middleware Systems are used to integrate a collections of *data sources* distributed

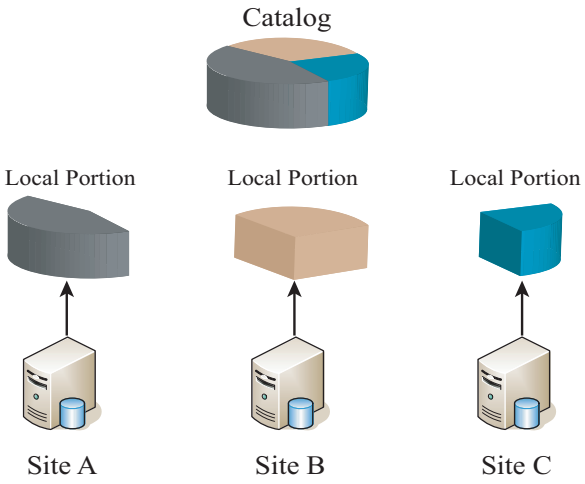



Figure 2.4: Partitioned catalog

over a computer network in a common DBMS-style framework. These data sources may not only be multi-vendor DBMS with different SQL dialects and data models, but also non-relational databases such as data organized in file systems (e.g. Mac OS, Linux, Windows), XML, or even object-oriented databases. To be effective, Database Middleware Systems must provide one or more integrated schemas, and must be able to transform data from different sources to answer queries against these schema [7].

There are two main approaches for Database Middleware Systems: *database mediator* and *database gateway*. In the first approach, a server application known as the *mediator* acts as the integration server for the clients. The mediator is specifically designed to handle data translation and schema mapping, and it provides many of the services of a typical DBMS such as query parsing, query optimization and query execution. The mediator uses the functionality of *wrappers* to access to the information stored in each data source. Wrappers typically reside as a stand-alone application or stored procedure near the data sources. They receive requests from the mediator and convert them into queries or procedure calls that the data source can handle in order to get the data needed. Some example of mediator systems are Pegasus [8], TSIMMIS [9], Garlic [10] and MOCHA [11]. Figure 2.5 shows a typical middleware mediator database system.

In the second approach, the database gateway establishes a point-to-point connection

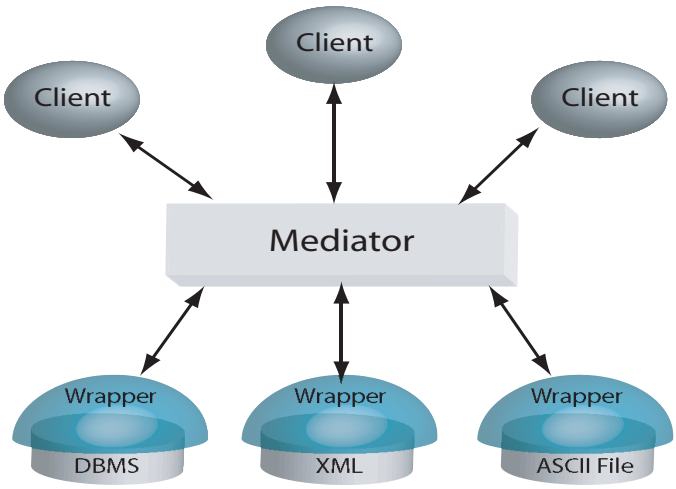



Figure 2.5: Typical Middleware Mediator Database System

between a local DBMS and one remote data source. The database gateway accepts client commands (using one set of APIs) to translate into remote data source commands (using a completely different set of APIs), also transforms the data returned from the remote data source into client format. Database gateways are provided by the major database vendors such as Oracle [12] and Sybase [13]. Figure 2.6 shows a typical mediator-based middleware database system.

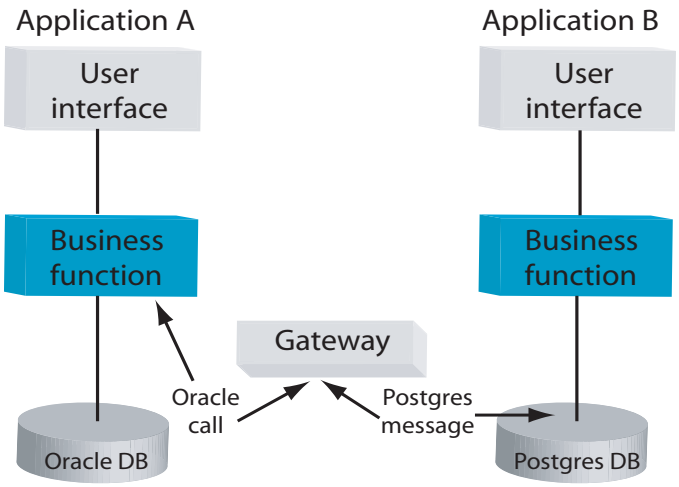



Figure 2.6: Typical Middleware Gateway Database System

## 2.4 NetTraveler Middlerware System

The NetTraveler Database Middleware System [1] is currently being developed at the University of Puerto Rico, Mayagüez Campus. NetTraveler is designed to support efficient data management over dynamic Wide-Area Networks (WANs) where data sources go off-line, change location, have limited power capabilities and form ad-hoc federations of sites.

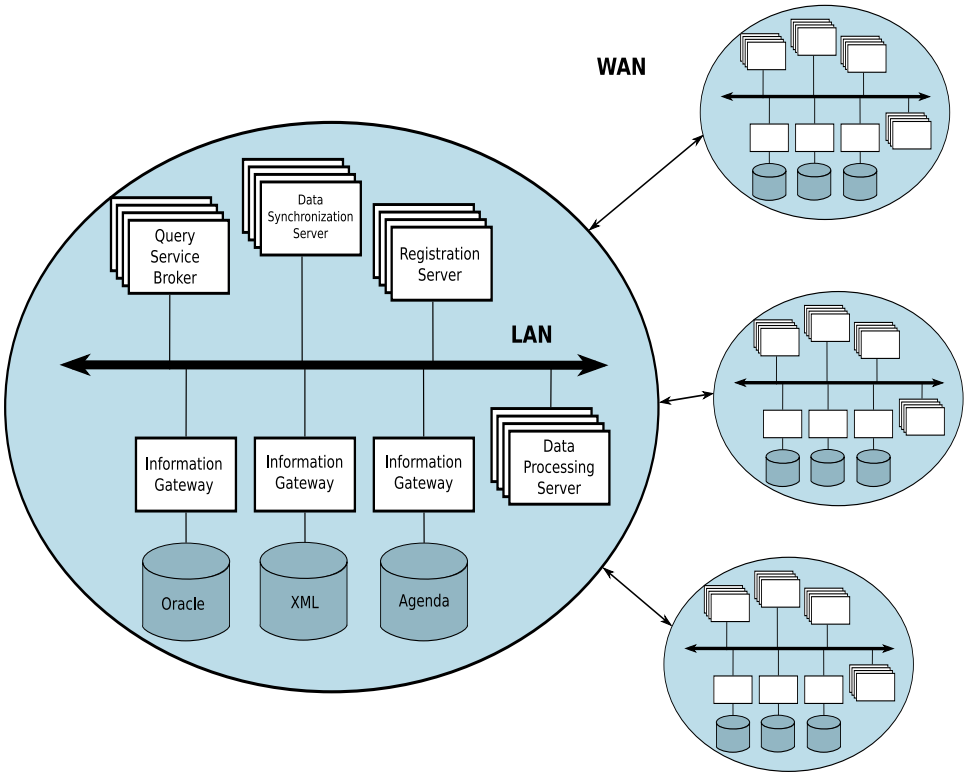



Figure 2.7: NetTraveler Architecture

Figure 2.7 depicts the diferent components of the NetTraveler architecture. Each circle represent a Local Area Network (LAN) which the connectivity can be wired or wireless. The cylinders represent the data sources which can be a DBMS, XML-Server or another customized data server. The basic organization unit in NetTraveler is known as ad-hoc federation. A federation is a collection of computational devices than have agreed to work together and share data and computational resources. A federation can spawn more that one LAN, and a LAN can have elements that belong to more than one federation. The simplest federation that can be build is called a Local

Group ($LG$). A $LG$ consist of one $QSB$, one or more data sources and their associate $IG$s, one or more clients, one $RS$, one $DSS$, and one $DPS$.

- The Query Service Broker ($QSB$) is a server application responsible for coordinating the execution of queries that are submitted by clients. $QSB$s exhibit a P2P behavior since a $QSB$ might contact other $QSB$s to help it solving the task at hand.
- The Information Gateway ($IG$) is responsible of giving access to the $QSB$s to the wealth of information contained in data sources.
- The Data Synchronization Server ($DSS$) is responsible for keeping synchronized copies of data on behalf of the client. The $DSS$ also takes the role of a proxy on the client's behalf to fetch query results when a client leaves a work session before a running query is completed.
- The Data Processing Server ($DPS$) provides a commodity service for computational tasks during query processing. These tasks include query execution, sorting, or any other type of special computational operation required.
- The Registration Server ($RS$) is responsible for maintaining metadata that describes federation elements and available resources. Two or more $RS$s work as peers to exchange metadata within local groups in a federation.

## 2.5 Peer-to-Peer Systems

A Peer-to-Peer (P2P) System is a distributed systems in which each participant (i.e. peer or node) have identical functionalities, responsibilities and is totally independent from the others. There is no central authority to control or manager how the resources of each participant in the network are utilized. Each participant may act as either a server or a client in different communication relations. Commonly, these participants are referred to as *servents* (derived from the terms client and server).

In [14] Roussopoulos, Baker et al. consider that a P2P system must satisfy the following three characteristics :

- **Self-organizing**: the nodes organize themselves into a network through a discovery process. Thus, the system automatically adapts to join, departures and failures in nodes.
- **Symmetric communication**: peers are considered equals; they may act either as a server or as a client.
- **Decentralized control**: peers determine their level of participation and their course of action autonomously. There is no central controller that dictates any behavior to individual nodes.

Content distribution probably is the application most widely known for P2P systems. Several P2P systems designed for this purpose have been developed and deployed in recent years. Examples of such systems are: Limewire [15], Kazaa [16] and eMule [17]. Others applications for P2P systems include a wide variety of purposes: distributed computation (Seti@home [18]) where computational processes are done on peers that volunteer for that, instant messaging (P2PMessenger), telephony services (Skype) and distributed filesystem (CFS [19] and N3FS [20]).

P2P networks can be classified in terms of their structure into two categories: *unstructured* and *structured*. Unstructured P2P networks organize the nodes in a random graph and without any rule which defines where data is stored. Gnutella [21] and FasTrack are examples of this P2P network type. Structured P2P networks are formed by following strict rules, and the resulting topology of the network is one that is controlled. Therefore, the neighbors of a peer are well-defined and the data is stored in a well-known site. Distributed Hash Tables (DHT) are a well-known type of this class of P2P system. Examples of implemented systems of this kind of P2P network are: Content Addressable Network (CAN) [22], Chord [23], and Pastry [24].

Typically, unstructured P2P networks use two search mechanisms to find data items: search by flooding and search with time-to-live (TTL). The flooding method is the naive approach, where a query message is broadcast to all network nodes until it reaches its target (i.e. a node with the request data). In the search with TTL, the query message encasulapes an additional TTL value indicating how long the message should stay in the network. The query message is forwarded

between the nodes until the query is either answered or the message TTL value has run off. In the later case, the message is discarded from the system and the sender is notified. Notice that either approach implies a relative large number of message forwards, which results in large response times for data lookup operations. The search mechanism in structured P2P networks is determined by the network topology chosen, because the topology determines where the data is stored. For example, DHT provide a fast lookup functionality based on hash functions to locate a data item in the network.

Several recent works [25, 26, 27] propose the integration between databases and P2P networks. Bernstein et al [25] propose the Local Relational Model (LRM) that allow semantic interoperability between peer relational databases by the assistance of coordination formulas. LRM assumed that the set of all data in a P2P network consist of local databases, each with a set of "acquaintances" (peers). For each pair of peers there are translation rules between data items and semantic definitions of dependencies between the databases of each peer. Siong et al [26] present a distributed data sharing system called PeerDB. PeerDB is built on top of BestPeer [28]. The main features of PeerDB are: first, it is a full-fledge data management system that supports content-based search; second, users can share data without a shared global schema; third, it adopts mobile agents to assist in query processing and finally PeerDB supports mechanisms to dynamically keep promising (or best) peers in close proximity based on some criterion. Arenas et al [27] present the Hyperion project that allows specifying and managing logical metadata that enable data sharing and coordination between peer DBMSs. This project is novel because it considers data coordination where peers are used as access points for both local and shared data. It also considers data sharing both within and across domains using mapping tables [29]. Finally, the project considers the use of mapping tables and mapping expressions to support information exchange between peers.

## 2.6 Chord

Chord is a structured P2P network that supports one main operation: given a key $k$ find the node $n$ that maps to key $k$. This node $n$ is the candidate place to store the data item(s) $X$

associated with key $k$. Chord uses a variant of consistent hashing to assign a $m$-bit identifier to keys and nodes. Roughly speaking, Chord can be considered as a distributed hash table where each node represents a bucket that stores values associated with a set of keys. The main idea is that buckets and nodes become the same thing, and the hashing process aims at mapping a key to the node where its associated data item should be stored.

Each node has an identifier (ID) which is computed by hashing the node's IP address. This ID becomes the "bucket number" for the node. Each data item has to be uniquely identifiable by a key computed by hashing the data item. Chord uses a cryptographic one-way hash function such as SHA-1 [30] to generate these $m$-bit identifiers for the keys and the node IDs. Based on this hashing techniques the idenfiers get assigned (hashed) to the nodes (the buckets).

In Chord, the node identifiers are sorted into an identifier circle called the Chord Ring, that cannot exceed more than $2^m$ nodes. Chord assigns a key $k$ to the first node in the ring whose identifier is greater or equal to $k$. This node is called the successor node of key $k$, denoted by $successor(k)$. In practice, this is how the data item finds its way to the node. The idea being that the item shall be stored in the node in which the hash function naturally assigns it. That node, however, might not be available. Thus, another node must be used. This process is similar to the assignment that occurs in closed hashing systems when a collision occurs. The figure 2.8 depicts an example that how keys are assigned to nodes in a Chord Ring. In this example we have an identifier circle with m = 3 and three nodes (1, 4, 6). Four keys (1, 2, 3, 6) are assigned to the nodes following the chord assignment mechanism. Successors are found in the clockwise direction of identifier circle. Since key 1 is assigned to node 1, keys 2 and 3 are assigned to node 4 and key 6 are assigned to node 7.

Each node $n$ maintains a routing table called the finger table. This table has $m$ entries, and the $ith$ entry ($1 \leq i \leq m$) contains the identifier of the first node $s$ that succeeds $n$ by at least $2^{i-1}$ positions on the Chord Ring, i.e. $s = successor(n + 2^{i-1})$ [23]. All arithmetic operations to calculate the finger table are modulo $2^m$. The node $s$ is also denoted by $n.finger[i].node$ and the identifier generated by the operation $(n + 2^{i-1})\ mod\ 2^m$ is denoted by n.finger[i].start.

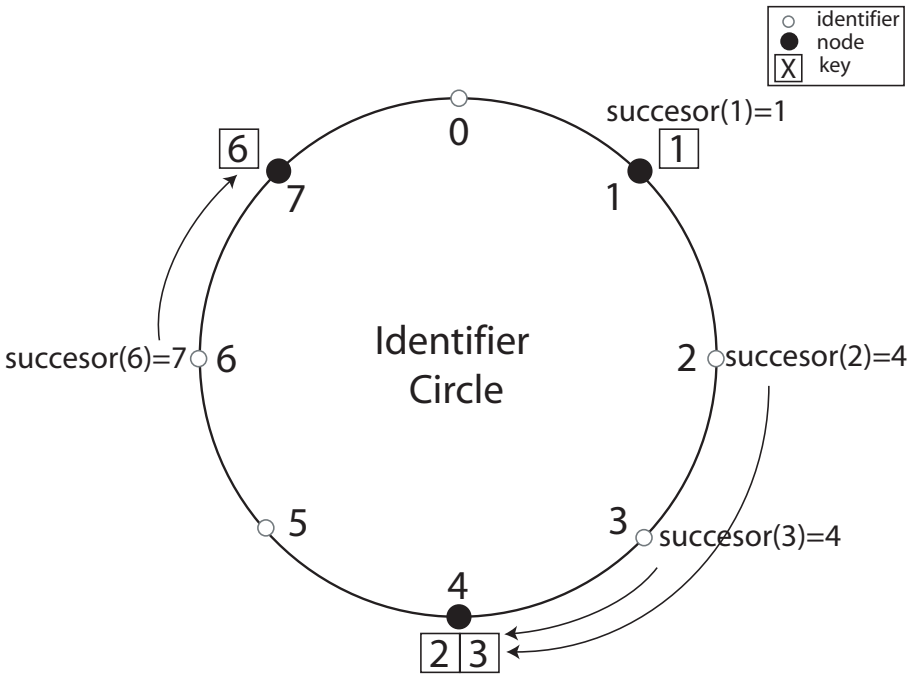



Figure 2.8: Chord assignment mechanism

For example, in the figure 2.9 the node with identifier 4 maintains information about the successors of the following identifiers:

$(4 + 2^0)$ ***mod*** $2^3 = 5$

$(4 + 2^1)$ ***mod*** $2^3 = 6$

$(4 + 2^2)$ ***mod*** $2^3 = 0$

The successor of identifier 5 and 6 is node 7 and the successor of identifier 0 is node 1. In this way, nodes store a small amount of information, and know more about peers that are closer in the Chord ring than other nodes. Additionally each node stores a reference to its predecessor node to enable an operation that traverses to nodes in the Chord Ring in a counterclockwise direction.

Chord provides us the means to obtain the node $n$ responsible for a key $k$ in no more than $Olog(N)$ messages, where $N$ is the total number of nodes in the network. The figure shows the pseudo-code to find the responsible node for a given key. The main loop of algorithm is *find-predecessor* which locates the immediate predecessor node $p$ for a given key $k$, then the successor of $p$ must be the successor node of $k$. The intervals used into the algorithm are adapted to a circular

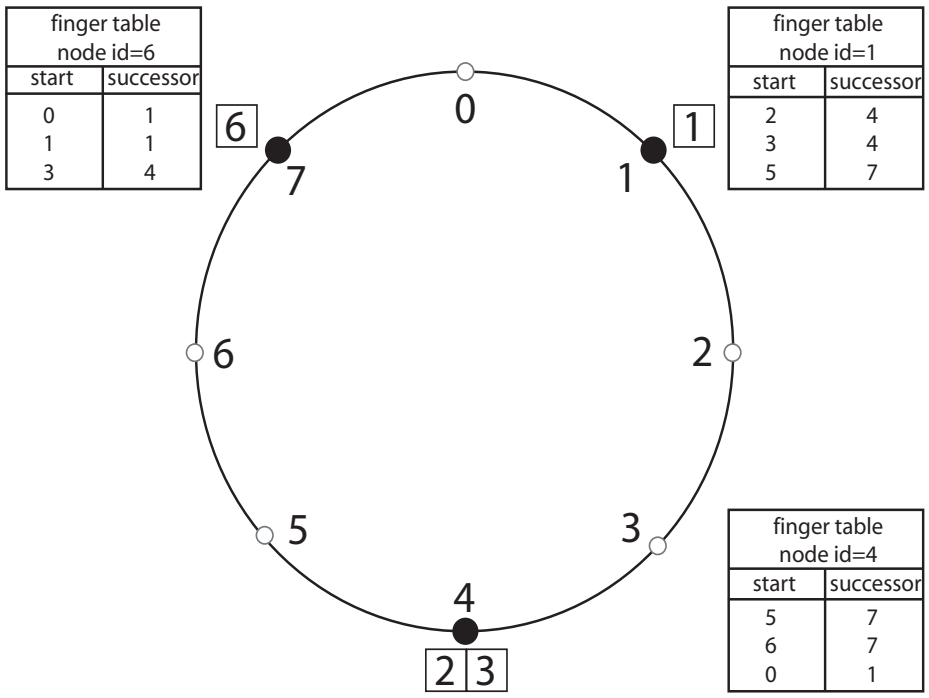



Figure 2.9: Finger Table

space, that is, the expressions $1 \in (0, 2]$ , $1 \in (0, 0)$ and $1 \in (6, 3)$ are all true. *find-predecessor* uses the *closet-preceding-finger* function which returns the node identifier inside the finger table that is nearest to the successor of key $k$. *closet-preceding-finger* can never return a node greater than the target identifier, therefore the algorithm never can overshoot the correct node. Each interaction of the loop in *find-predecessor* halves the distance to the target identifier.

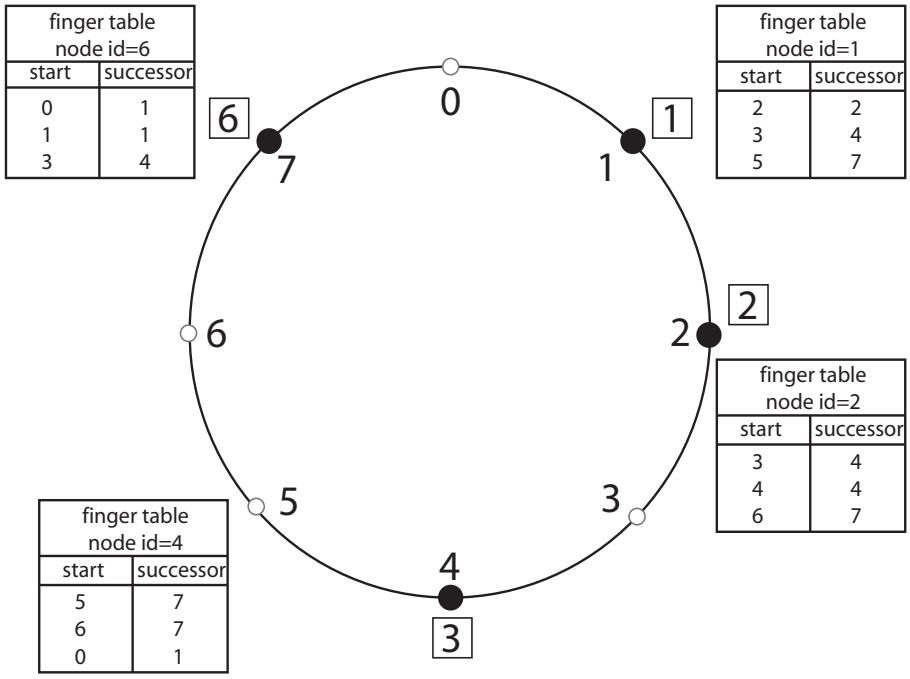
Chord is able to adapt to the ever changing network, allowing entrance and departures of nodes. When a node $n$ joins the network, certain keys previously assigned to $n$'s successor now need to be assigned to $n$. When a node $n$ leaves the network, it transfers its keys to $n$'s successor and tells to $n's$ predecessor that it has left the network. Figure 2.11(a) depicts the situation when the node 2 is added, and compare with Figure 2.9. The key 2 initially was assigned to node 4 but with the arrival of node 2, this key must be transfers to the new node to preserve the consistency of key assignment schema. The predecessor of node 4 and the successor of node 1 are adjusted to node 2. Also, the finger table of all nodes reflect the arrival of node 2. Figure 2.11(b) depicts when the node 4 leaves the Chord Ring (compare with Figure 2.11(a)). The keys assigned to the node 4

are transfered to its successor, node 7. The node 7 updates its predecessor and the node 2 updates its succesor. Also, the finger table of all nodes reflect the departure of node 4.

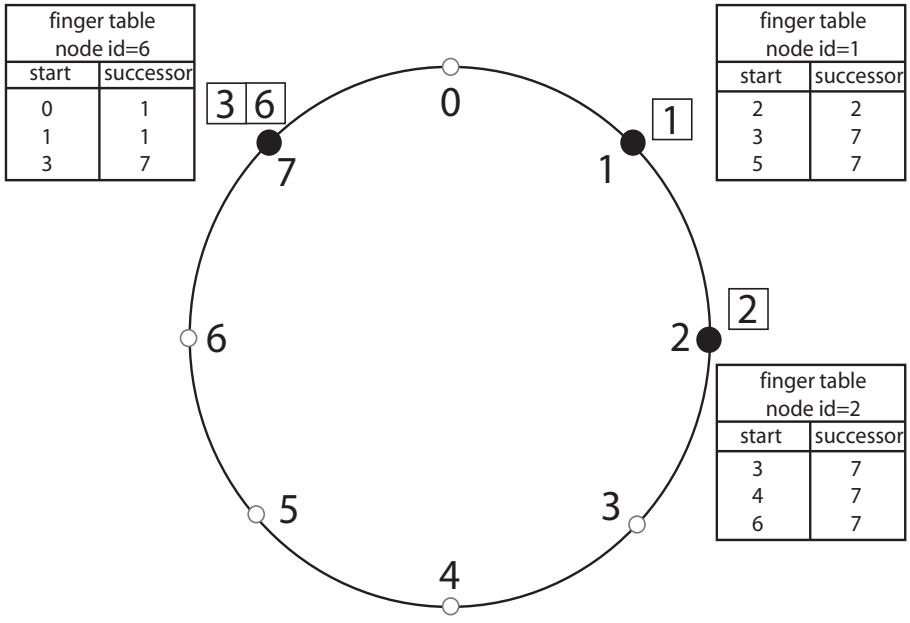
```
find_successor(key)
   n = find_predecessor(key);
   return n.successor;

find_predecessor(id)
   n = this;
   while (id ∉ in (n, n.successor])
     n = n.closet_preceding_finger(id);
   return n;

closet_preceding_finger(id)
   for i = m downto 1
   if (finger[i].node ∈ (n,id) )
     return finger[i].node;
   return n;
```

Figure 2.10: Pseudo code to find the successor node of a specific key

Chord also defines a mechanism to increase system robustness by arbitrarily selecting the size $r$ of the *succesor list* that each Chord node maintains. This list stores the first nearest $r$ successor in the Chord Ring. If a node's immediate successor does not respond, this node can explore the rest of the list until a responding node is found.




(a) Join node in the system




(b) Leave node in the system

Figure 2.11: Nodes joins and departures

# CHAPTER 3

# Catalog Manager System Design

This chapter presents the catalog manager system developed to aid in the process of metadata dissemination on the NetTraveler middleware system. It starts by displaying the basic organization of our approach and how the metadata is modeled with it. This chapter continues by discussing the mechanisms for searching, deleting, and updating of metadata records in the system. Next, the management of arrival and departures of nodes is explained in detail. Finally, this chapter discusses the cache manager developed as a mechanism to speed up search operations.

## 3.1 Catalog Manager Organization

A distributed systems integrated with NetTraveler can be envisioned as a set of Federations $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$. Each federation $F_i \in \mathcal{F}$ can be modeled as a set of local groups $\mathcal{LG} = \{LG_1, LG_2, \ldots, LG_m\}$. Each Local Group $LG_i \in \mathcal{LG}$ contains at least one Registration Server $RS$ which maintaints the metadata that describes resources and objects of its Local Group $LG_i$. All of these $RS$ servers become peers in a peer-to-peer network to exchange the metadata, and form a distributed catalog manager for all the metadata in the system. In terms of logical organization, all the $RS$s in the system are organized into a Chord Ring. Figure 3.1 illustrates the basic organization of our approach. Specifically, this figure shows four local groups and their respective registration servers.

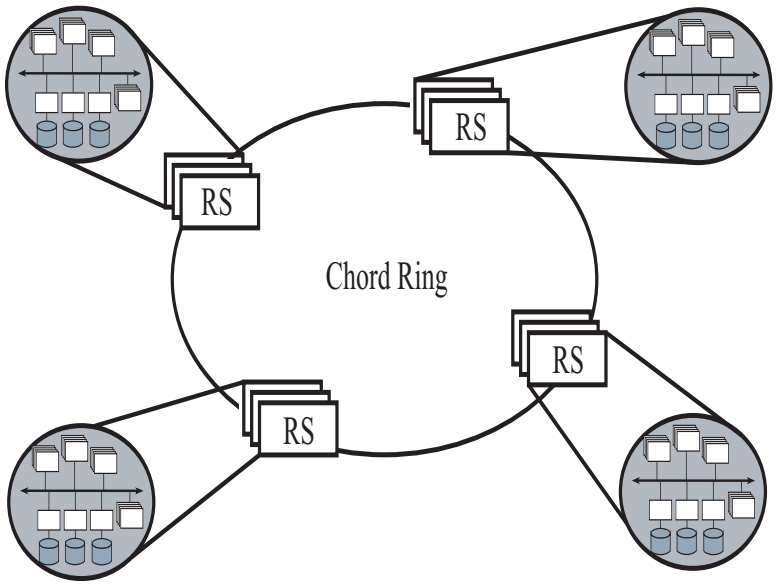



Figure 3.1: Organizations of Registration Servers

Each metadata record $r$ managed by an RS is an $n$-tuple $r = (a_1, a_2, ..., a_n)$ representing information about a particular type of resource in the system. The structure and number of attributes in the tuple depend on the type of resource being represented. Given an tuple $r = (a_1, a_2, ..., a_n)$, an attribute $a_i$ is actually an ordered pair $a_i(u, v)$, where $u$ is a property name and $v$ is property value. For example, suppose we need to represent the host name and IP address of a node in the system. We can have a collection $Hosts$ which contains tuples with this information, and each tuple can be of the form $((NAME, myhost.com), (IP, 136.145.1.1))$. These are the records of information then the RS needs to exchange to propagate the metadata throughout the system.

To simplify and standardize this exchange, we can encode the record in XML and instantly make the record easy to exchange and manipulate across several NetTraveler components. We also use XML Schema definition to define metadata types and constraints to support validation. As another example, suppose that we have a relational scheme to represents airline operations. A record that represent the column *passenger-id* of table *passengers* is modeled as follow:

```
((ColumnName, passenger_id),
 (Type, ((SourceType,int8),(Size,8),
             (ImplType,Integer))),
```

```
(isKey,true),
(TableName, passengers),
(Position, 0),
(Schema, airline))
```

This record represents the metadata about a column in a table used in the middleware system. The table could either be a local table in a data source, or a global table in the integrated schema. Using our XML syntax we can encode the record as follows:

```
<ColumnName>passenger_id</ColumnName>
  <Type>
    <SourceType>int8</SourceType>
    <Size>8</Size>
    <ImplType>MIInteger</ImplType>
  </Type>
  <isKey>true</isKey>
  <TableName>passengers</TableName>
  <Position>0</Position>
  <Schema>airline</Schema>
```

This approach has many advantages that we cannot overemphasize. First, the use of XML shields the system from the choice of storage system used in the RS to store the records. Second, the exchange of metadata becomes independent of the data types available in the programming language used to implement the RS, and hides any architectural issues (e.g. 32-bit vs 64-bit integers). Third, the code to process the XML-encoded record can leverage on existing tools to convert from XML to objects (e.g., Java Objects) and vice-versa. This latter feature has become a time saver in our case, since we were able to use the JAXB toolkit for this purpose.

In Chord, the data items to be store need to have a key, and since we use Chord as the foundation to our system, we must find a way to build a key for each metadata record. We

have defined an unique *id* for the metadata records, which is fed into the Chord hash function to produce the key value for the a given metadata record. This *id* for the metadata record is built from a subset of its property-value pairs. For example, if we take the column metadata object previously discussed, we can build its *id* by combining the values of its database name, table name and the column name properties:

```
(schema_name = airline;
 table_name = passengers;
 column_name = passenger_id;)
```

Basically, the values of these properties are concatenated to form a unique id, and then hashed to produce a Chord key. This key can then be used by the system to perform the lookup operations necessary to locate the node that the full metadata record associated with this key. The next subsection, we provide more details of this process.

## 3.2 Searching for Metadata

Given a *QSB b* that receives a query request $Q$ from a client $c$, the *QSB b* initally will determine from the query $Q$ which tables must be accessed, what kind of physical resources must be allocated, which query operators need to be used, and so on. These tasks are accomplished by examining catalog metadata under the control of system's *RS*s.

In order to search a metadata entry $e$, the *QSB b* submits a query message to the *RS r* in its local group. This message contains the *id* of the requested metadata entry $e$. The *RS r* receives this message and hashes the *id* to obtain the key $k$ of the requested item. With this key, the RS $r$ looks in its finger table for the successor node of $k$. If the RS $r$ happens to be the successor node of $k$, then it looks for the metadata in its local database of entries and returns the result. If not, the RS $r$ checks if the successor is in the finger table. If the successor is found in the finger table, the RS $r$ will ask this node for the metadata, and returns the result. Else, if RS $r$ does not know the successor, it asks this information from the node that is closer to key $k$ in the identifier circle

of the Chord Ring. The reason for this is because this node will know more about the key $k$. This process is repeated until the node that contains the key $k$ is found. At this point the node will reply with either the metadata entry $e$ (if it exists) or a message indicating that the entry is not present in the system at this point.

Figure 3.2 depicts the chain of events that occur when a $QSB$ asks an $RS$ ($N5$) for the metadata associated with table *students* from database *university*. The name $N5$ stands for the node with *id* 5, and we follow this convention in the rest of this work. The events shown in the figure unfold as follow:

- First, the $QSB$ sends to the $RS$ $N5$ a lookup message with the metadata $id = (database : university, tablename : students)$ corresponding to the metadata object that is trying to find (*step 1*). The $RS$ receives the lookup message and hashes the *id* to obtain the *key* associated with the metadata object. For this example, let us suppose that the *key* value is 85. The $RS$ uses its finger table to find the successor node for *key* 85. In the finger table for $N5$ the value $N70$ is the largest, meaning the that finger table holds the *id* 70 as its largest value. This means that the $RS$ cannot determine directly the succesor node because the successor *id* in the finger table has to be the first one that is greater or equal to 85. Hence the RS forwards the lookup message to the node in its finger table with highest identifier not exceding the *key* value of 85.
- Continuing, for this particular case we are referring to node $N70$, whose *id* is 70 (*step 2*). The node $N70$ inspects its finger table, and determines that the successor for *key* 85 is the node $N90$ (whose *id* is 90).
- So, node $N70$ forwards again the lookup message to node $N90$ (*step 3*).
- Next, the node $N90$ returns to the originating $RS$ the metadata associate with the *key* 85 (*step 4*).
- Finally, the $RS$ returns to the $QSB$ the metadata associate with the table *students* (*step 5*).

Notice that the process is deterministic and the number of lookup messages is bound by $O(log(N))$, where $N$ is the number of nodes in the system [2]. Thus, at the end of the process

the QSB will get an answer which is either the metadata being searched for, or an empty value indicating that the metadata is not present in the system at this time.

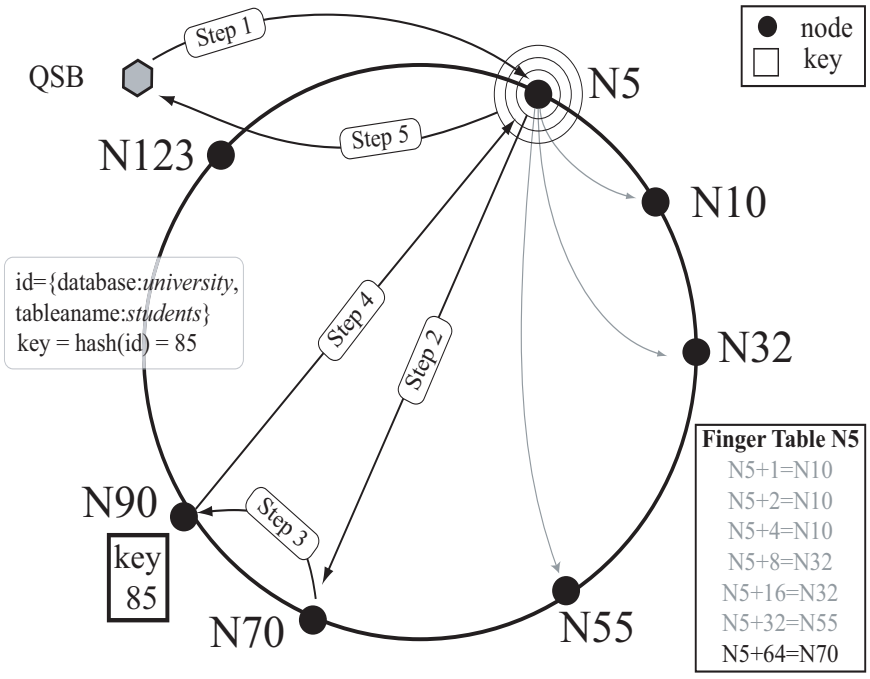



Figure 3.2: Searching data in the sytem

## 3.3 Deleting and Updating Metadata

A metadata deletion takes place only when a $QSB$, $IG$ or a $DSS$ node starts a request for deletion. This deletion process works as follows. First, a node $n$ submits a message to a specific $RS$ $r$ invoking the deletion of the selected metadata element referenced by a specific $id$. Next, the $RS$ $r$ hashes the corresponding $id$ to obtain the $key$ $k$ for the referenced entry. Then the $RS$ $r$ locates the $succesor(k)$ node where the metadata is held. Finally, the $succesor(k)$ node proceeds with the deletion of the data associated with $key$ $k$ and a confirmation message is sent to node $n$ to confirm that the deletion operation was successful.

Consider the scenario illustrated in the figure 3.3 which consists of the sequence of steps to eliminate a metadata entry whose $id$ is $(database : university, tablename : students)$. First, the

*QSB* sends to *RS* *N*5 a deletion message with this *id* (*step 1*). The *RS* receives the deletion message and hashes the id to obtain the *key* of this metadata entry. In the example, the *key* value is 85. A search proccess is then initialized to locates the *successor*(85) (see section 3.2). The *steps 2* and *steps 3* are identical to the previous search example. When *successor*(85) is located, in our example *N*90, the metadata entry associated with this key is deleted, and a confirmation message is returned to the *RS* (*step 4*). Finally, the *RS* forwards to the *QSB* the confirmation message (*step 5*) indicating that the metadata was deleted. If the metadata entry is not found, the *QSB* receives an error message.

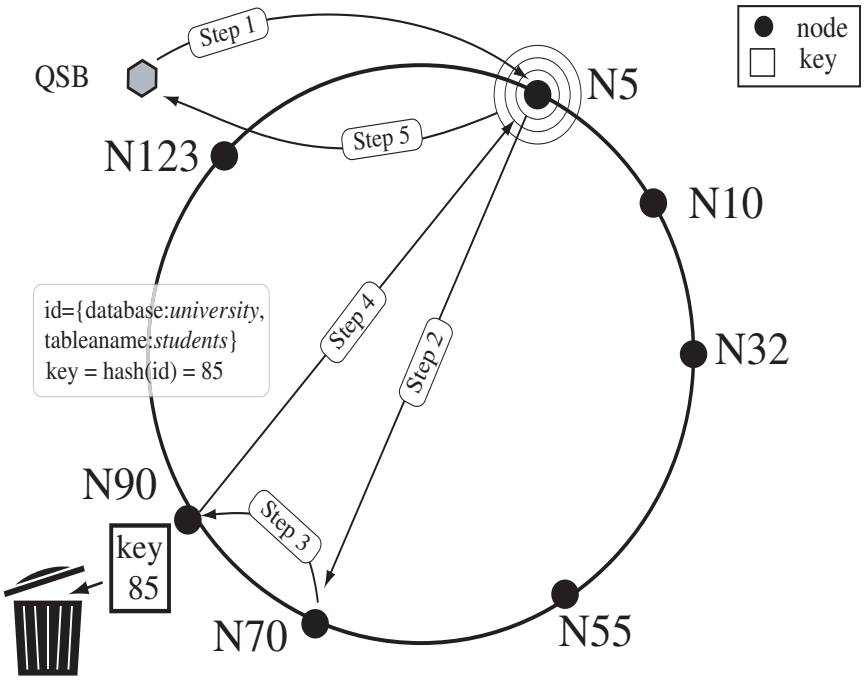



Figure 3.3: Deleting data in the sytem

Contrary to other solutions such as CFS [19] which consider the metadata or data as read-only (immutable), our approach supports updates. The metadata update operation is carried out in two steps: a deletion of the existing metadata followed by the insertion of the updated metadata. The reason for this two-step approach has to do with the fact that the *id* of the object might change as a result of the changes in value. As a result, the metadata item might now belong to a node different from the one that held the item before the update operation.

Figure 3.4 depicts the chain of events that occur when a $QSB$ updates the metadata associated with table *students* from database *university*. A change on these metadata is the table name which will become *sophomores*. First, the $QSB$ submits a message to a $RS$ $N5$ invoking an update operation. This message contains the $id$ of the old metadata entry and the value of the new metadata. Notice that steps from 2 to 4 are the same as in the deletion process. Steps 5, 6 and 7 are new additions, which are explained as follows. *Step 5* begins when the $RS$ computes the corresponding $id$ for the new metadata entry ($database : university, tablename : sophomores$), and the resulting $key$ is 30 . The RS then searches for the node that will hold the metadata associated with the $key$ 30, which for this particular example is the node $N32$ (whose is $id$ 32). This completes the *step 5*, and *step 6* gets underway. RS $N5$ awaits until the insertion operations is completed at node $N32$ and receives a confirmation message. Once the confirmation message is delivered to $N5$, this RS returns an acknowledgment message to the $QSB$ (*step 7*). Notice that after the update process is completed, the updated metadata entry is located in a different node ($N32$) from the original node ($N90$) due to changes in the $id$'s attribute that cause the mapping of the metadata entry to a new $key$.

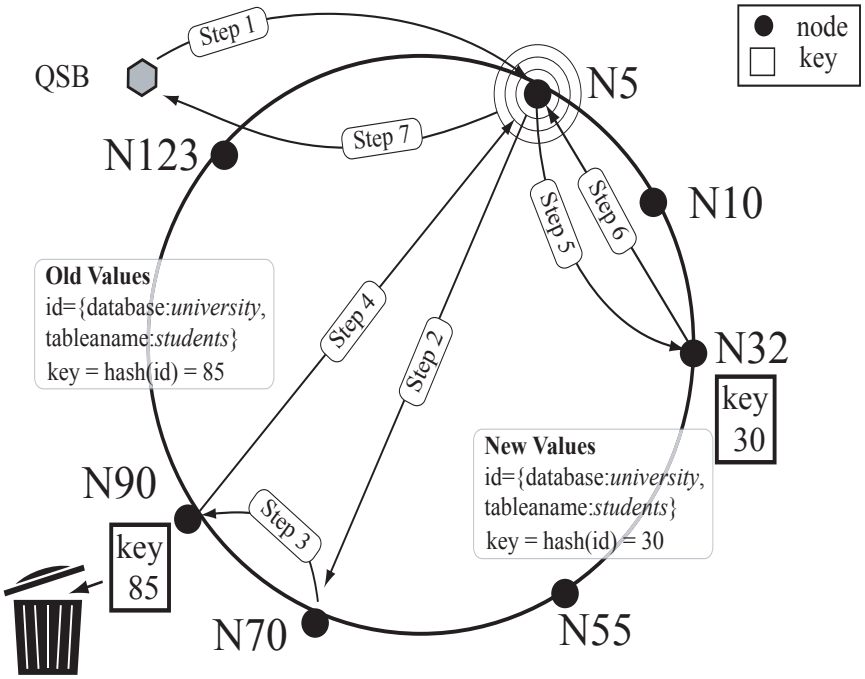



Figure 3.4: Updating data in the sytem

## 3.4 Arrival and Departures of Nodes

The structure of the database middleware system can be very dynamic as new nodes constantly enter and leave the system. This raises the issue of how to manage the metadata associated with the node that is arriving or leaving the system. This dynamic behavior is valid for metadata producer nodes ($QSB$'s, $IG$'s, $DPS$'s and $DSS$'s) as well as metadata storage (holder) nodes ($RS$'s).

When a metadata producer node $n$ joins the system, first it must register to a specific Local Group $L$ through the $RS$ of $L$. Next, $n$ submits to the $RS$ its metadata information such as: tables, schema mapping rules and physical resources. These metadata are valid only for a certain period of time, therefore, $n$ has to re-register periodically to validate these metadata into the system. The addition of a metadata holder is a little more complicated and it is handled by Chord. When a new metadata holder node $m$ enters the system it contacts an existing metadata holder node $m'$. Through node $m'$, $m$ initializes its predecessor node and its finger table. Likewise, existing nodes in the ring reflect the addition $m$, updating its finger table and predecessors node. Finally, $m$ contacts its successor node $s$ and begins a process in which $s$ transfers to $m$ all the keys and the entries associated with this keys that should be stored by $m$. Basically, node $s$ sends all the keys and its entries that belong to $m$ based on the assignment of keys in the Chord ring.

A metadata producer node might leave the system gracefully or abruptly. In the first case, the node has time to inform to the $RS$ of its Local Group about its departure. Next, the $RS$ proceeds to invalidate the metadata node into the distributed catalog. This might be the case when someone shuts down his/her device or simply disconnects from the network. In the abrupt case, a failure of the device or network causes an unexpected departure from the system, with no time to contacts the $RS$. In this case, the $RS$ waits for a re-registration process by the node, if this steps is not completed a specific time then the $RS$ proceed to invalidate the metadata node.

When a metadata holder node leaves the system gracefully, Chord transparently transfers the keys and entries associated with it keys of the leaving node to other nodes in the system. In the

case of abruptly departure, we use replication to enhance the basic Chord algorithms for departures to guarantee that metadata are not lost.

We decided to create replicas of the same metadata entry and store them on different nodes. Each replica has the same key as the original entry and theoretically must be stored in the same node, but we need these replicas to be stored in different nodes without affecting the search mechanism.

To successfully store those replicas on different nodes, each node uses its successor list to identify possible nodes to send those replicas. There is a need to be careful with this approach, because synchronization issues regarding the management of the metadata and its replicas could arises. For example, when a deletion is presented, not only the original data must be deleted, but we also need to delete the replicas present in the node's successor list.

## 3.5 Caching of Metadata Entries

Caching can be used as an additional mechanism to improve response time of the search operations in the system. Notice that since the catalog is distributed, the metadata lookup process must incur in a little overhead to locate the entries. To hide this overhead, we implemented mechanisms for metadata caching on our *RS* nodes as a way to decrease response time for future queries. In our approach, each *RS* node caches metadata search results as part of the lookup process. These search results contain metadata entries and their corresponding keys. In this way, if a client requests to an RS for a previously requested metadata entry, the client will probably receive the cached metadata from the current *RS* node, and the overhead in the search process is averted.

Since our system is dynamic, the changes that occur over time in the metadata store might result in metadata inconsistencies between the entries in the cache and the actual entries in the storage nodes. To mitigate this problem, each entry in the cache is assigned a time-to-live (TTL) value, which indicates how long the cache entry should be valid in the cache.

During a metadata search operation, if a cached entry is hit, the system first verifies its validity by comparing the TTL value with the difference between the current timestamp and the entry cache insertion timestamp. If the TTL is still is valid, the cached entry is returned to requester $QSB$. Otherwise, the cached entry is invalidated, a normal search operation is conducted as discussed in section 3.2, and the result of this search is stored in the $RS$ cached (replacing the invalid entry). The figure 3.5 depicts an example of how the cache stores some entries. The entries with keys 10 and 30 are obtained from the cache instead of searching other $RS$ nodes to find the items.

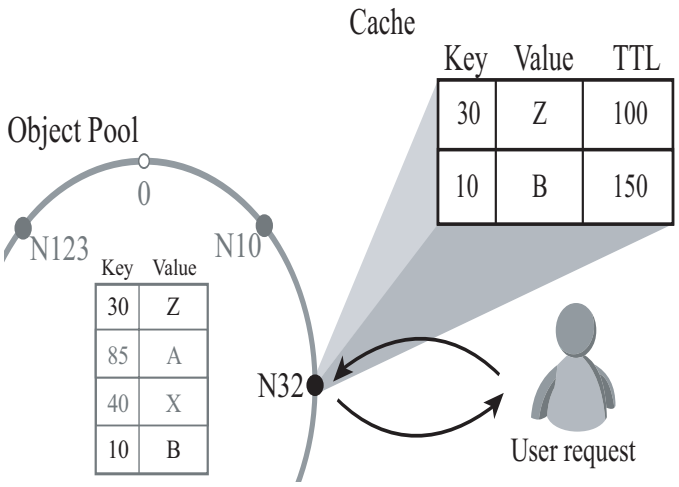



Figure 3.5: Cache example

Due to the limited size of the cache with respect to the size of the distributed catalog, we must define a cache replacement strategy to handle the situation in which a new entry must be added but the cache is full. In our approach we selected the well-known Least-Recently-Used (LRU) replacement strategy, in which the object evicted from the cache is the one that has been unused for the longest time. This strategy is very popular in in the context of cache memory management.

Figure 3.6 shows in pseudo-code the search procedure with metadata caching in place. This metadata caching scheme does not change the way of forwarding request between the $RS$ nodes, but rather tries to answer the metadata requests before the responsible node for the metadata item needs to be contacted. Other systems based on Chord, such as CFS [19] and PeerOLAP [31],

$M_i$ : a metadata entry requested for the client
$t$ : time-stamp when $M_i$ is requested
$t_i$ : time-stamp when $M_i$ was inserted into the cache
$TTL_i$ : time-to-live of $M_i$
**case:**
$M_i$ **in cache**
**if** $TTL_i \leq t - t_i$ **then**
  invalidate $M_i$ {$TTL_i$ expired}
  fetch $M_i$ from responsible $RS$ node
  cache $M_i$ {replaces old copy}
**end if**
$M_i$ **not in cache**
**if** cache is full **then**
  evict a entry $N_j$ from cache
**end if**
fetch $M_i$ from $RS$ node
cache $M_i$ {replaces evicted entry $N_j$}

**return** $M_i$

Figure 3.6: Pseudo code to search procedure with metadata caching

implement caching techniques to speed up search operations.

# CHAPTER 4

# Implementation of the Catalog Manager System

## 4.1 Overview

This chapter discusses the implementation details of the Catalog Manager System. It begins by discussing the architecture of the main component of our system: the *Registration Server* (*RS*). This architecture is presented in a layered structure, where each layer have a defined specific role. The chapter centers on describing the functionality and implementation details of each layer.

## 4.2 Registration Server Layers

The main component of our catalog manager, the RS, is organized into several layers of functionality, each responsible for a specific set of tasks. Figure 4.1 depicts the current layered architecture of the *RS*. These layers are: EndPoint Layer, Query Abstraction Layer, Distribute Storage Layer, Cache Manager Layer and Persistence Layer.

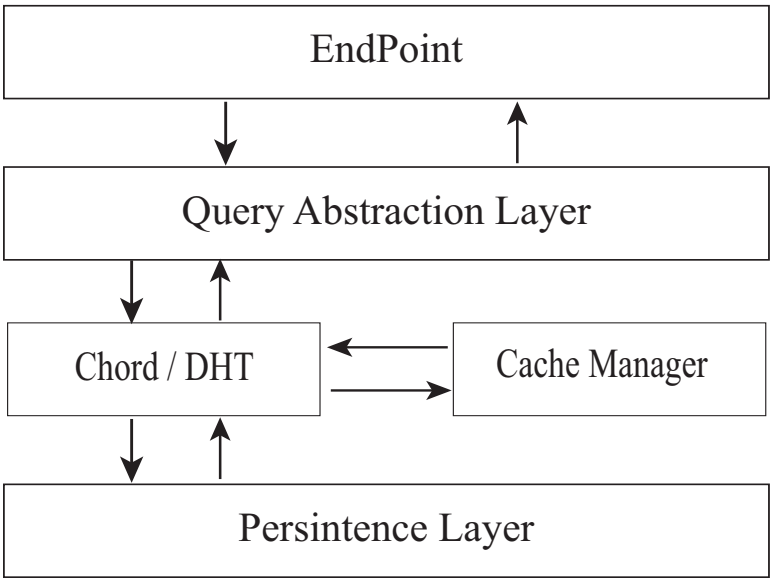



Figure 4.1: RS Architecture

### 4.2.1 EndPoint Layer

The Endpoint layer is an abstraction mechanism to provide clients and peers with a mechanism to invoke the functionality of the RS component, thus enabling them to issue request for metadata items. This layer was implemented as a Web Service and the clients/peers interaction is based on the Simple Object Access Protocol (SOAP), using HTTP as the transport protocol. The EndPoint layer functions at high level are:

1. Receive metadata lookup requests and other operation by listening for SOAP calls, handling the extraction of Java objects contained in SOAP messages and passing them to the Query Abstraction Layer.
2. Answer metadata and other request by creating SOAP response messages with Java objects from the Query Abstraction Layer.

In SOAP messages, all data and application-specific data types are represented as a XML document. SOAP use type mappings to determine how to convert objects and their values between the native programming language (Java in our specific case) and the XML representation. The type mapping of objects is performed by serializing them into XML so that they can be

transmitted in the SOAP message, and this process is known as marshalling. The reverse process, de-serializing, neccesary to convert XML to native language objects is also known as un-marshaling. Most SOAP toolkit implementations provide their own translators (marshalling and un-marshalling mechanisms) to map primitives data types to XML and vice-versa.

SOAP only has support to simple types (i.e. integer, float, date, string, enumerations, etc.), binary encoding, compound types (only two type of composition are available: struct and array) and generic compound types (when the compound data value is not known in advance). In more complex objects like Hash-Tables, the developer has to implement his/her own SOAP serializer and des-serializer mechanisms, to convert the object to XML and vice-versa.

Since our catalog manager was designed to store/retrieve any kind of objects, we can not create custom serializers and deserializers for them. To deal with this situation, we decide to use SOAP with Attachments (SwA). With this mechanism, all request and response message objects are sent as attachments in the SOAP message. SwA utilizes the Multipart/Related MIME [32] framework to package the SOAP message together with the binary attachments, and utilizes URI based mechanisms to reference the MIME parts.

With the use of SwA, the EndPoint layer process a SOAP request as follows. First, the attachment is read from the incoming SOAP message. Second, the system extracts the bytes from the attachments and reconstructs the objects using Java de-serialization mechanism. Finally, the reconstructed object request is sent to the Query Abstraction Layer for processing. In the same way, when the EndPoint layer sends a SOAP message, all objects that will be send with the message must be serialized and attached to the SOAP outgoing message. The object serialization is done by using the Java serialization mechanism.

There are numerous frameworks and toolkits for Java that deal with web services, amongst which we can highlight: Apache Axis 2, Apache Axis 1.4, Java Web Services Developer Pack (JWSDP) and XFire. For our development process we choose Apache Axis 1.4 for several reasons: 1)at the development stage of this work Apache Axis 2 still was not at its first stable release, 2)

JWSDP project had been discontinued, and 3) up to this date XFire still does not support SOAP with attachments. As the application container, we used Apache Tomcat version 5.5. The Axis engine provides both client and server side message processors as client and server side handlers. Figure 4.2 presents the communication between the clients and RS through of Web Services.

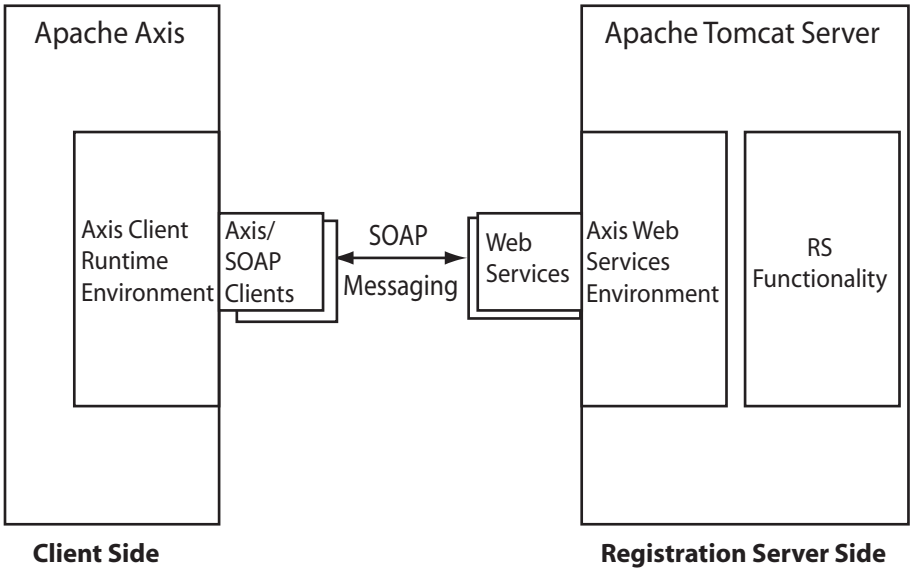



Figure 4.2: Communication between client and RS server

### 4.2.2 Query Abstraction Layer

The Query Abstraction Layer (QAL) abstracts upper levels from the complexity of the Chord/DHT layer, by transforming client request to Chord/DHT operations. The QAL receives from upper layers information concerning the method or command to execute and a set of input parameters related to the method being requested and executed at the Chord/DHT layer. Also the QAL encapsulates results of requested calls by the upper layers. This mapping from upper layer commands to Chord/DHT operations is not always a one-to-one match. For example, an *update* operation correspond to two Chord/DHT operations; first a *remove* operation and then an *insert* operation.

QAL was implemented following two design patterns: factory and command. The factory pattern defines a common mechanism for the creations of objects instances. This instances must

have a common parent class, but different specialized functionality. The resulting instance is decided by the factory depending on one or more parameters. The main benefit of this pattern relies on the unifying mechanism for derived classes' creation. but each of them performs a task differently and is optimized for different kinds of data. The Factory Pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code.

In our case, the objects returned by the factory pattern, follow the common pattern known as Command pattern. The Command design pattern encapsulates the concept of the command into an object. The issuer holds a reference to the command object rather than to the recipient. The issuer sends the command to the command object by executing a specific method on it. The command object is then responsible for dispatching the command to a specific recipient to get the job done.

Figure 4.3 shows a simplified class diagram of the QAL. First, we begin describing the class *Parameter* which represents the parameters sending by the client to execute a specific command. The Parameters objects are sent by the users, serialized in a SOAP message, extracted by the EndPoint Layer and sent to QAL.

The class *Command* is an abstract class that encapsulate the specific actions requested by the clients. This class encapsulates information concerning the name of the command to be executed, the list of parameters required to execute a command, and the result obtained after the command is executed. The specific command functionality is implemented by children classeses of this *Command* class. Children classes encapsulate the concrete logic of the command to be executed by lower layers. The classes: *Join*, *Leave*, *Delete*, *Update*, *Insert* and *Lookup*, represent the concrete commands that can be executed by the system. All these class inherit the attributes and methods from the class *Command*. *QueryAbstraction* class contains all necessary logic to deals with the request from upper layer. The *Factory* contains all the logic to parse, validate and create a correct command from its parameters and name.

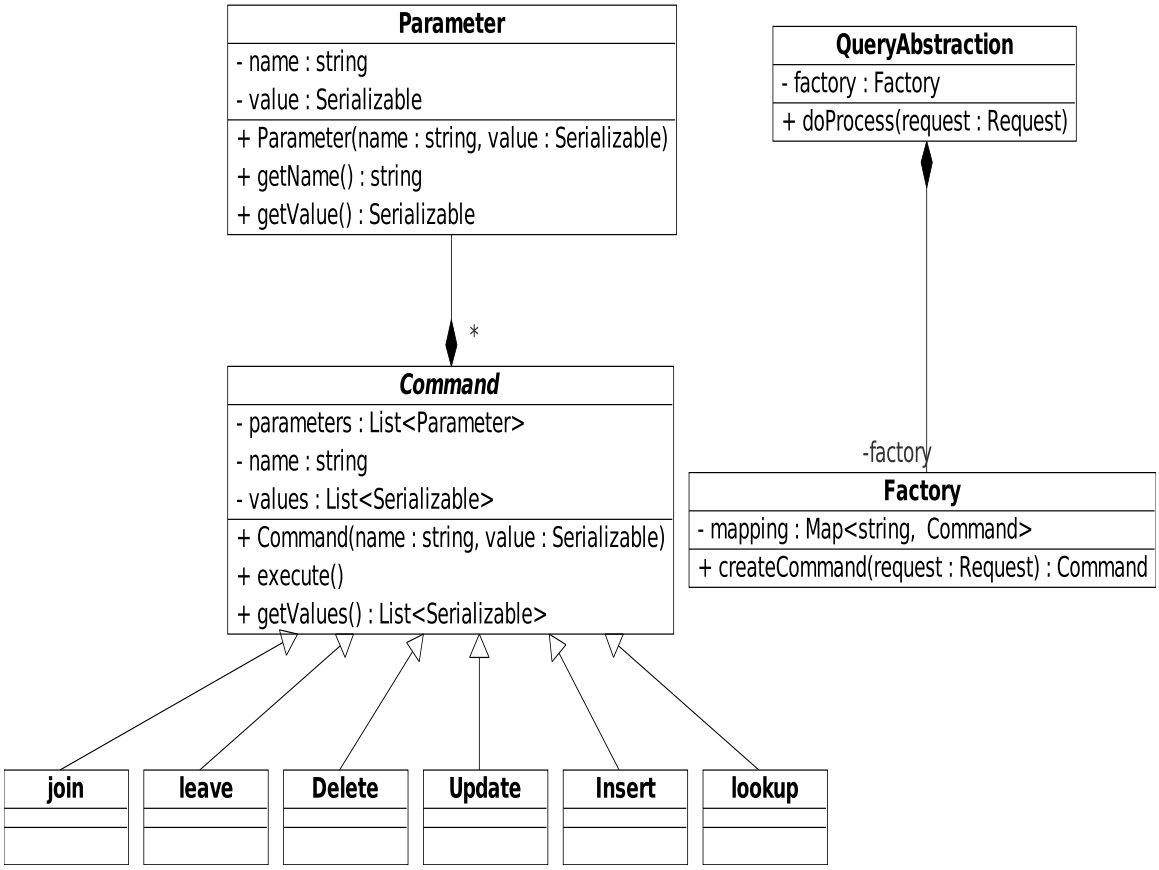



Figure 4.3: Simplified diagram class of Query Abstraction Layer

### 4.2.3 Chord/DHT Layer

Our Chord/DHT layer is based on the current implementation of the Open Chord [33] project. Open Chord is an open source Java-based library developed by the Distributed and Mobile Systems Group of Bamberg University, Germany, and is distributed under the GNU Public License (GPL). Figure 4.4 represents the architecture of the Open Chord project. [1]

The image shows an architecture composed of four layers, on which the Chord/DHT implementation is composed of the last three layers. The lowest layer of their architecture corresponds to the implementations of the communication protocols. Currently, this layer only provides support for two different protocols. The first protocol provides communication between a single instance of the Java Virtual Machine (JVM). This protocol is used only to provide a mechanism for testing and debugging. The second, and the one we are using, correspond to a socket-based TCP/IP communication protocol. Although, they only provide these two protocols, this layer provides the necessary mechanism and interface to provide the capability to extend Chord/DHT to additional protocols.

The middle layer provides an abstraction layer for peer communication, independent from the actual communication protocol in use. This layer also provides mechanism to support synchronous and asynchronous calls between peers. The next layer contains the logic of the Chord overlay network. This layer provides the necessary mechanism to find, store and delete objects, and also for joining, creating and leaving of a particular network. This layer abstract applications from the actual implementation of the routing mechanisms within the Chord/DHT. This layer is also in charge of handling crashes and desertion of peers, although it is recommended for a peer to announce its decision to leave a network.

A synchronous behavior would block each request until the operation has been performed and a results are obtained. This scheme would not be suitable for our purposes, so a non-blocking

[1] This image was re-printed with permission from the Open Chord project. It originally appears on their web site.

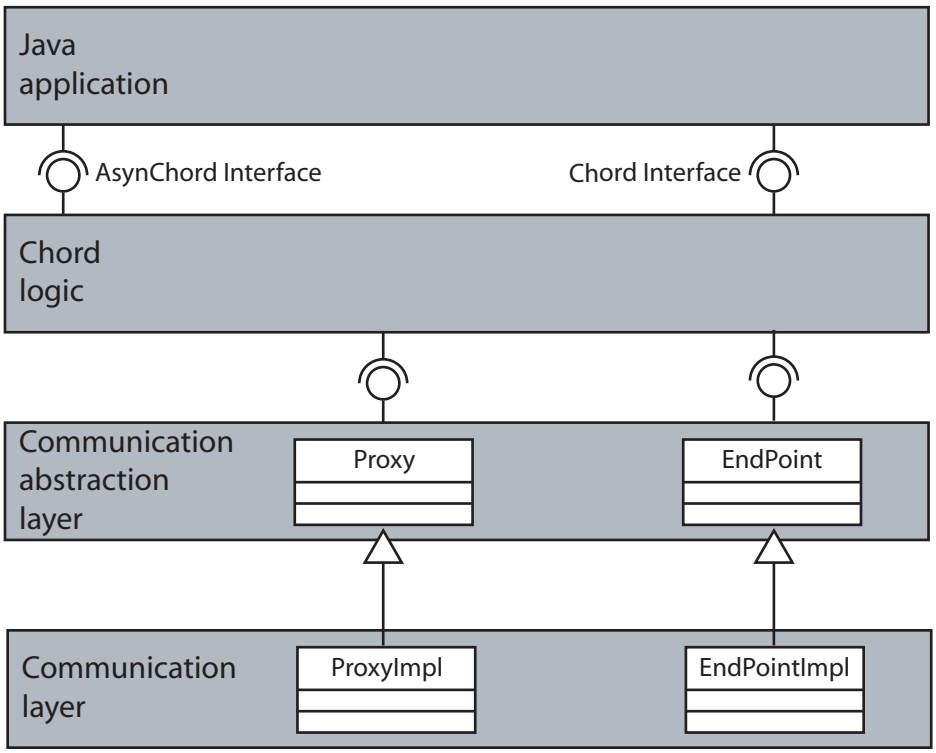



Figure 4.4: Open Chord Architecture

scheme is needed to provide a mechanism that guarantees concurrent operations independent of the termination of the process. In order to provide such logic, the upper level must rely on the interfaces provided by the Communication abstraction layer.

In order to provide asynchronous and synchronous communication the Chord logic layer provides interfaces and methods for providing both communication schemes. The Chord logic layer must rely on the Communication layer to provide such schemes. Also the Chord Logic layer is responsible for data replication and maintenance of the properties that are necessary to keep the DHT running.

The Chord/DHT layer allows the distributed storage and retrieval of metadata entries, leveraging on the efficient location of metadata provided by Chord's decentralized lookup algorithm. Given a key and a value, three basic operations were defined for this layer:

- store operation - *put(key, value)*
- lookup retrieval operation - *value = get(key)*

- remove operation - *remove(key)*

For more information concerning the implementation and design of the Open Chord library please refer to [33].

### 4.2.4 Persistence Layer

The Persistence layer allows the *RS* to persists metadata entries, and to recover from node failures. This layer takes care of saving, loading and deleting metadata entries from a persistence store, and to quickly recover metadata after an *RS* node crashes. A standard file system can be used to implement this layer, but our system uses a relational database because it provides a simple model to organize the data into tables as well as all the facilities typically associated with databases: transaction management, concurrency control, recovery, queries, etc. We choose PostgreSQL as the persistence DBMS because of its speed, ability to handle large volumes of data, and strong support for binary data formats. The metadata entries are stores in binary format in a table. Table 4.2.4 shows the structure of the table where the data is stored.

| **Field** | **Description** |
|---|---|
| id | string of 40 characters that represent the octal format of the 160-bits corresponding to the hash value of a key. Also, this field is the primary key in the table. |
| value | binary representation of the entry value. The data type used for this field was *bytea*. |
| insertion-time | timestamp of the insertion of entry into the table |

Table 4.1: Fields of entry table

### 4.2.5 Cache Manager Layer

The Cache Manager Layer (CML) enables the creation of transient copies of frequently requested metadata items. When a lookup for a metadata entry is requested, the node first verifies if the entry requested is located inside the cache to avoid the normal process of finding the metadata

through the Chord/DHT layer. This layer provides a pluggable interface for various replacement policies. Also, this layer provides an additional mechanism to deal with possible inconsistencies between cached data and actual metadata distributed across the Chord/DHT layer.

Figure 4.5 shows a simplified class diagram of the CML. The class *Frame* represents the object values stored in the cache. The main attributes for this class are: *expiration-time* which is a constant that indicate the maximum time for a *Frame* object in the cache, *time* that indicate the time-stamp when *Frame* object was inserted and *items* that represent the object value store by the cache. The interface *ReplacementPolicy* encapsulate the specific actions for a cache replacement strategy. *LRU* class is a concrete implementation of this interface. For last, the class *CacheManager* contains all necessary logic to represent a consistence cache in memory. Internally, a Hash Table is used to implement the cache storage.

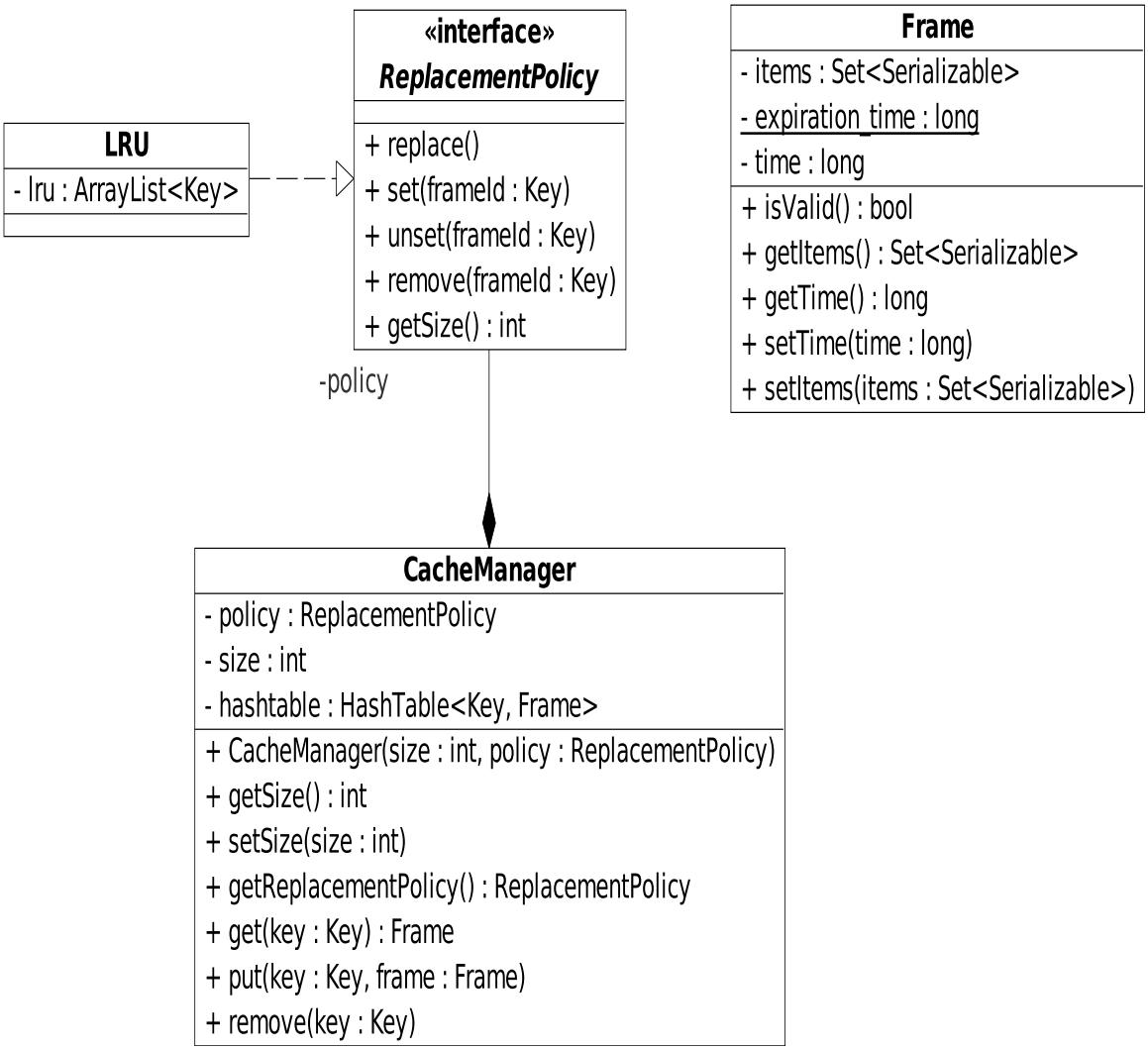



Figure 4.5: Simplified diagram class of Cache Manager Layer

# CHAPTER 5

# Experimental Results

## 5.1 Introduction

This chapter presents a performance study of an RS prototype, designed to validate our proposed ideas and help us to clarify our assumptions. The idea was to setup an RS network on several computers, to create a distributed catalog manager and establish some real clients to interact with this catalog using the *RS* component. The specific objectives of the experiments were:

1. Compare our distributed catalog approach with current conventional approaches for catalog management, examining the possible effects that arise due to random system failures. The conventional approaches used were: centralized, fully replicated and partitioned.
2. Measure the effect of cache size on the response time of our system.
3. Measure and compare the throughput of our approach with current conventional approaches.

To test the first objective, a fixed a number of failures were generated in each catalog manager organization for a preset time window. These failures were independent, non-simulataneous and randomly distributed during this period. Additionally, for each configuration we repeated the experiment changing the number of clients. More details about this experiment can be found on Section 5.2. The second objective was tested by performing changes in the cache size of our approach and measuring the response time for each cache size. Likewise, for each configuration we

repeated the experiment with a different number of clients. This experiment is detailed in Section 5.3. The third objective was tested by the submittal of metadata lookups to the system by an increasing number of clients. This experiment is detailed in Section 5.4.

The current *RS* implementation was developed using the Java 1.5 programming language, Apache Tomcat [34] as the application server, Apache Axis [35] tools for web service deployment, and PostgreSQL 8.0 as the database management system. Our experimental environment is composed by:

- Homogeneous IBM xSeries Linux cluster, with 64 dual proccessor nodes at 1.2GHz, 1GB of RAM per node memory and 40GB of storage. These machines were used to run the client applications.
- Eight (8) IBM servers with dual Xeon processor at 3.6 GHZ, 4GB of memory and 140GB of storage. Each machine ran an RS component.

## 5.2 Effect of Failures in the System

In this section we present the performance of our system for the first scenario explained above. We wanted to study the impact of failures on the percentage of queries correctly answered by the various catalog management organizations. Table 5.2 summarizes the setup used for each catalog configuration.

For each of theses cases, we ran experiments in which clients were constantly submitting metadata requests to the system. We ran several trials using various numbers of clients: 5, 10, 20, 30 and 40, and each client ran by itself on a cluster node. Each client would ask for a sequence of metadata objects in a random fashion. Each trial would last ten minutes, and in this period the catalog system would experience three random failures (independent from one another). Three experimental runs were performed for each architecture and client combination to calculate the percentage of queries answered correctly by the system.

| Catalog type | Setup description |
|---|---|
| Centralized | For each one of the eight servers, a centralized catalog was built. In each server, we ran the same experiment in which client sent request for metadata. We averaged the results from each server. . |
| Fully Replicated | We used the eight servers and replicated in each one all catalog records. |
| Partitioned | Each one of the servers contained a partition of roughly equal size of the catalog, with the total eight forming the whole catalog. |
| RS Approach | We used eight servers to built a catalog manager using the RS. The catalog records were distributed based in the hashing scheme that we adapted from the Chord algorithm. |

Table 5.1: Catalog setup for the first set of experiments

Figure 5.1 depicts the results for the centralized approach. As can be see in the figure the percentage of queries answered for a different number of clients is very similar, nearly to 60%. Comparing these results with the other catalog approach the centralized catalog has the worst performance among all. The reason for this is due to a failure in the central site which causes all of the catalog entries to be temporarily unavailable until the central site is restored.

The results obtained with fully replicated approach are presented in Figure 5.2. This figure confirm that the ideal performance, i.e. 100% of queries answered, can be achieved with fully replicated approach. That is thanks to the replication concept, whenever a failure occurs at least a copy containing all catalog entries will be available. However, a fully replicated catalog is very difficult to implement and maintain in practice because of the complexity and cost incurred in the update operations.

Figure 5.3 shows the results for the partitioned catalog approach. The partitioned catalog approach has a better performance than the central repository approach when the number of clients is low; nevertheless as the number of clients increase, it can be seen that both approaches are very similar. This is due to the fact that when a catalog request is processed by this approach, the system internally contacts each server, one-by-one, until the catalog record is found. This method produce a overhead in communication that is far more evident when the number of clients are increasing.

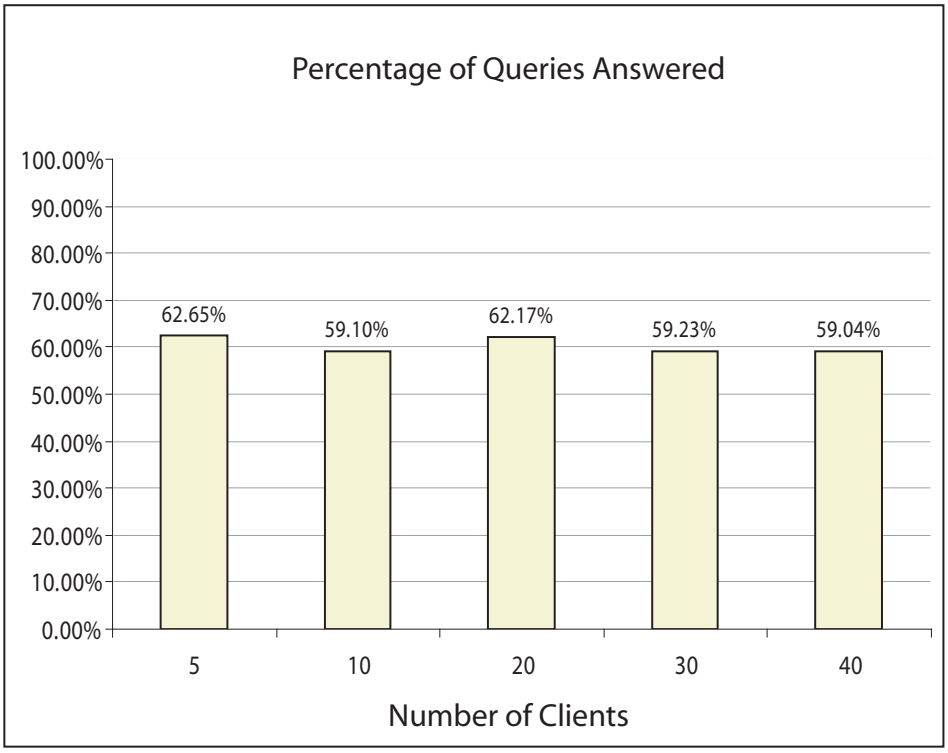



Figure 5.1: Percentage of queries answered in centralized catalog approach

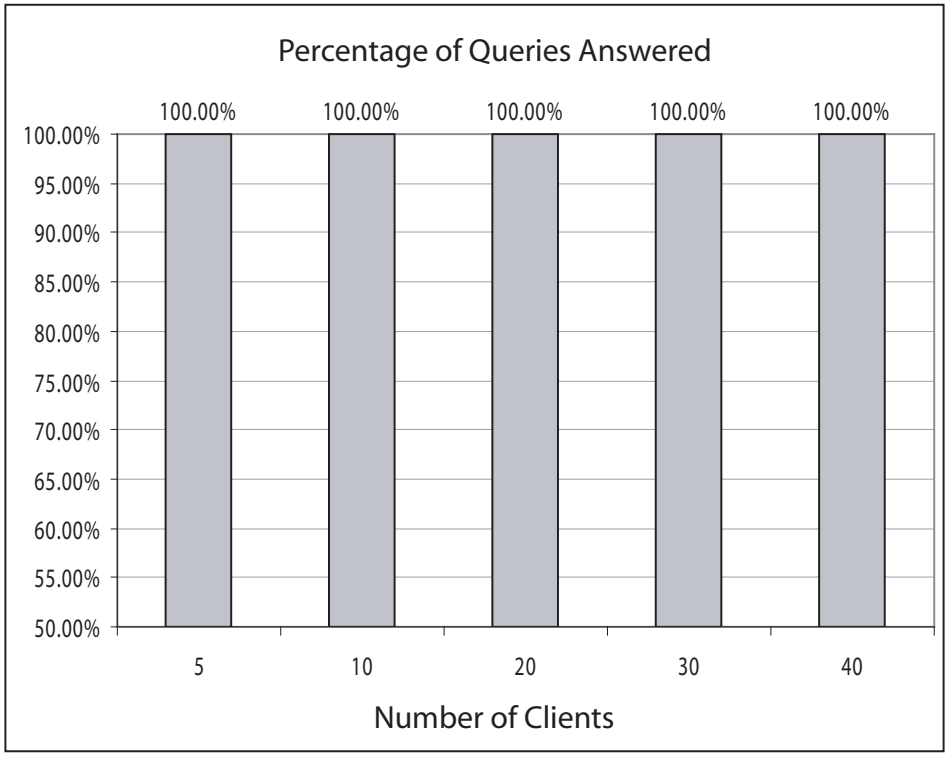



Figure 5.2: Percentage of queries answered in fully replicated approach

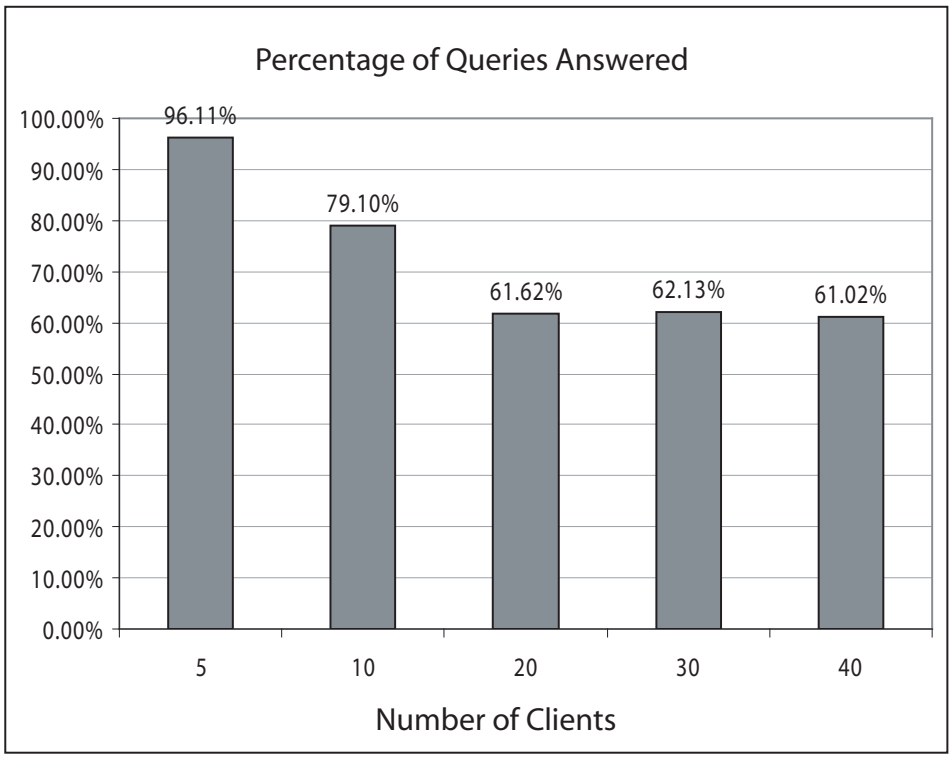



Figure 5.3: Percentage of queries answered in fully replicated approach

The results obtained with our RS approach (derived from Chord) are presented in Figure 5.4. This figure depicts that our approach approximate the performance of the fully replicated catalog system, but without the need replicate all catalog records. Notice that our approach reaches very similar performance (99.9% answered queries) compared with the the fully replicated version (100 %). This slight difference is due to the stabilization process necessary to handle the departure of node. During that period, the responsibilities for the metadata items are being reassigned by updating the finger tables, successor nodes, and predecessor nodes. Nonetheless, our approach compared favorably with the ideal fully replicated catalog, but is an option than can be implemented in practice.

## 5.3 Effect of Cache Size in Lookup Response Time

The second set of experiments was designed to measure the impact that the node cache size has on response time. This experiment consists of a comparison our RS approach in four

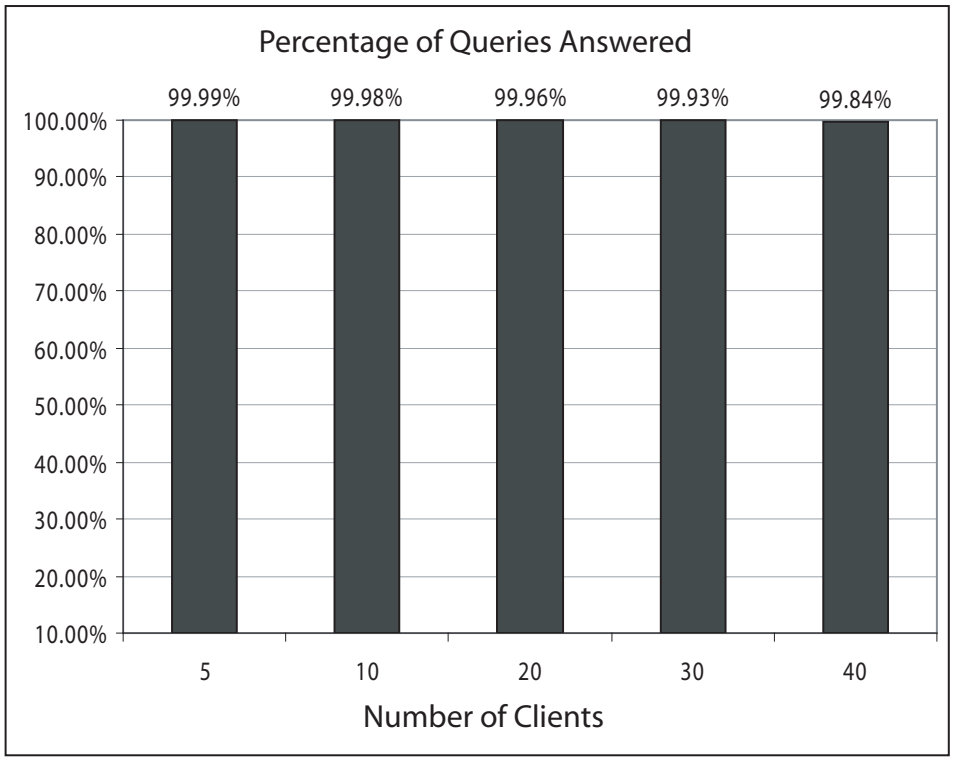



Figure 5.4: Percentage of queries answered in Chord based approach

configurations:

- No cache
- Cache holding up to 5% of the total metadata
- Cache holding up to 10% of the total metadata
- Cache holding up to 13% of the total metadata

We set up 8 servers running our *RS*s in order to deploy our distributed catalog manager, and we had a variable number of clients: 5, 10, 20, 30 and 40 that were constantly submitting request to the system. For the purpose of this set of experiments we impose our clients to only submit lookup request to the system. Each client would ask for a sequence of metadata objects in a random fashion. Three experimental runs were performed for each configuration to calculate an average response time.

Figure 5.5 depicts the results when the RS is without a cache. As can be seen from the figure the increase in the average response has a exponential behavior with the increasing of number of clients in the system. This is due to the overhead produced by the high traffic communication between the nodes when the number of clients is high (i.e., more than 20 clients). As can be seen for lower number of clients (less than 20) the system behavior is nearly stable, with a difference of approximately 30 *m*sec just for an increment of twice the numbers of clients. This behavior motivated us to use a cache, with the goal of improving the response time of the RS. As can be seen for the subsequent results with the cache implemented our hypothesis tested correct, thus the response time is highly affected by the cache, that is, if more cache, is added the system tends to respond faster to client queries.

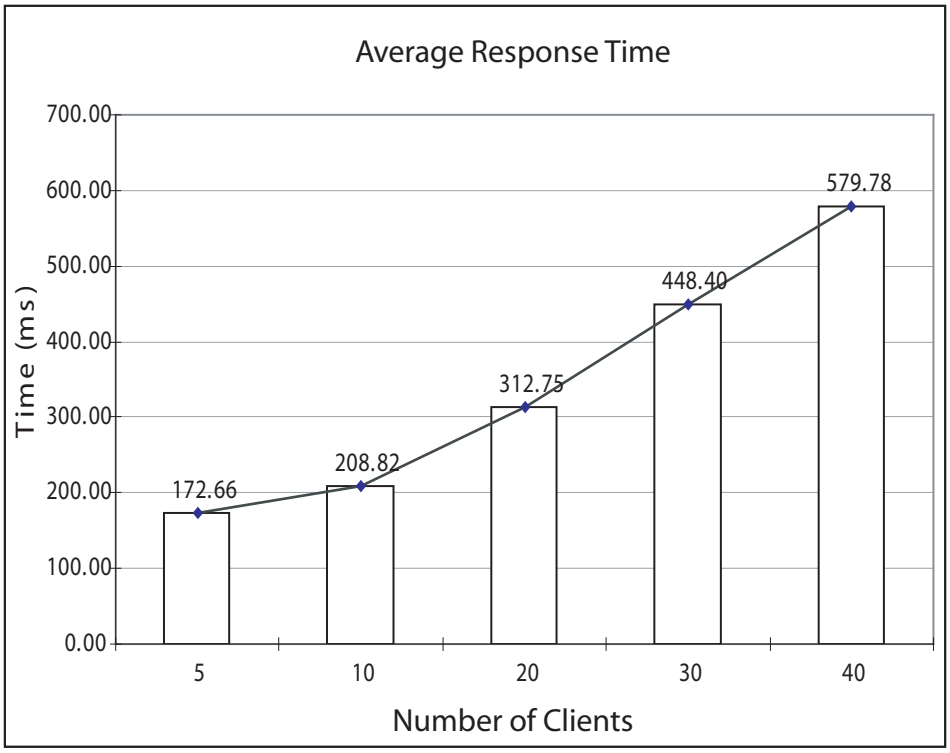



Figure 5.5: Average response time without cache

The results obtained with cache a holding up to 5% of the records in an RS are presented in Figure 5.6. This figure confirm the previously stated hypothesis about the cache size. It is clearly shown that the improvement is not satisfactory since we are using a cache relatively small (recall

that it is only of 5 percent of the total records). Nonetheless, an increase of only 5 percent reflects in an approximately 30 percent reduction in the response time.

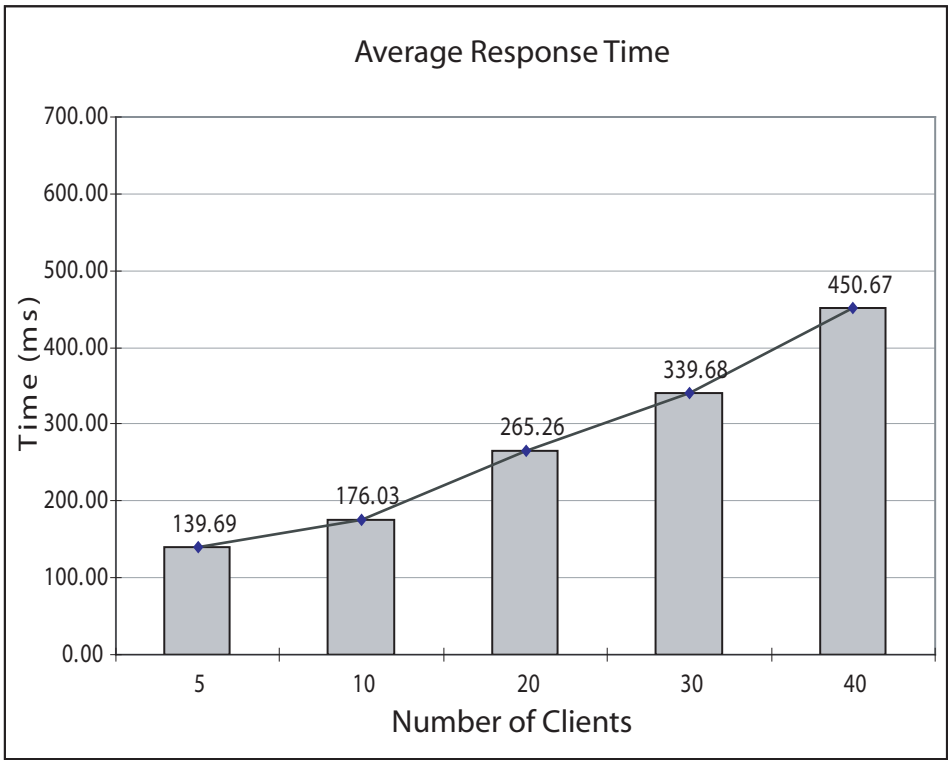



Figure 5.6: Average response time with cache holding up to 5% of the total records

Figure 5.7 depicts the results by doubling the cache size to 10% of the records.. Besides a noticeable reduction of nearly 50 percent in response time versus the original system without cache, the most important finding is that the exponential behavior of the systems now tends to a quasi-linear one. This in fact presents an outstanding accomplishment, since if the system behavior is sustained then, one can be able to estimate the system behavior by a simple extrapolation without incurring in significant errors when dealing with large amount of clients (real life scenarios).

Notice that for comparison purposes we are emphasizing on the higher number of clients, since ultimately we want to build a scalable practical system and the lower number of clients does not account for that.

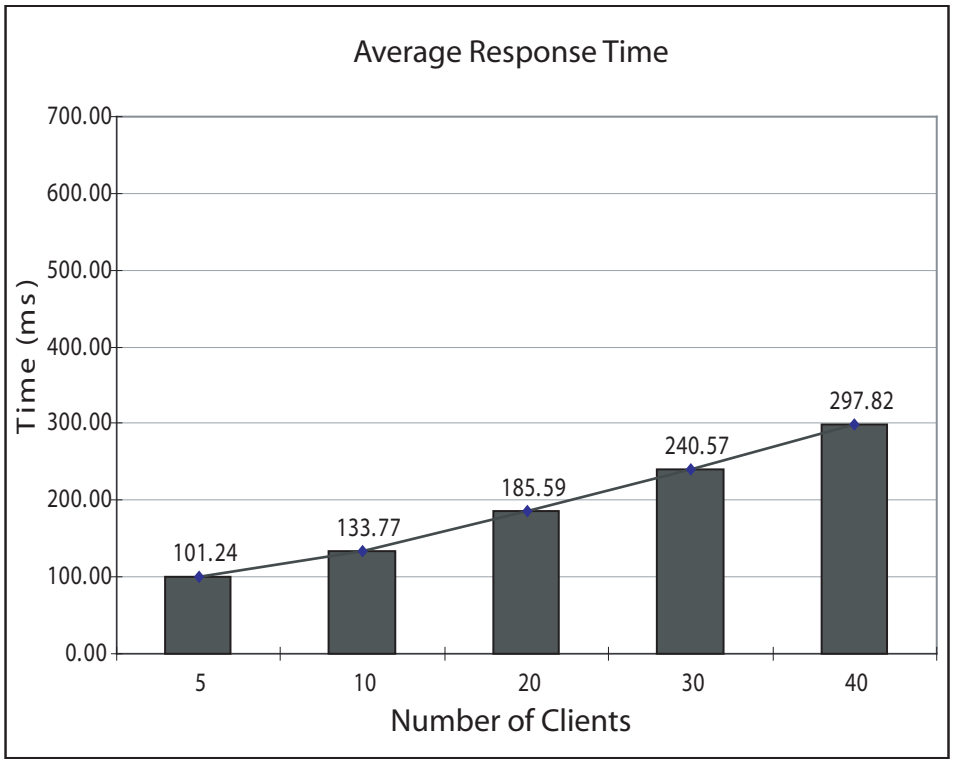



Figure 5.7: Average response time with cache holding up to 10% of the total records

The results obtained with a cache holding up to 13% are presented in Figure 5.8. For this test we did not want to double the size of the cache again, i.e. to 20%, since 20% of large data-sets can translate into lot of resources misused, instead we find 13% cache increase to be a reasonable balance between resources usage and the resulting improvement in response time. We do not plan on estimate the system behavior for this test, since in order to obtain a correct curve fitting it is mandatory to replicate the test with more clients, but it appears that the curve tends to saturate at some point. Nevertheless, when compared to the performance obtained on figure 5.7 it is shown that the behavior outperforms the quasi-linear one, or we can be conservatives and say that the system is also quasi-linear but with a lower slope.

We now present the reader with a plot of all experiments results on the coordinate system, showing only the curves fitted, with the purpose of further clarifying the main findings of this work. See figure 5.9.

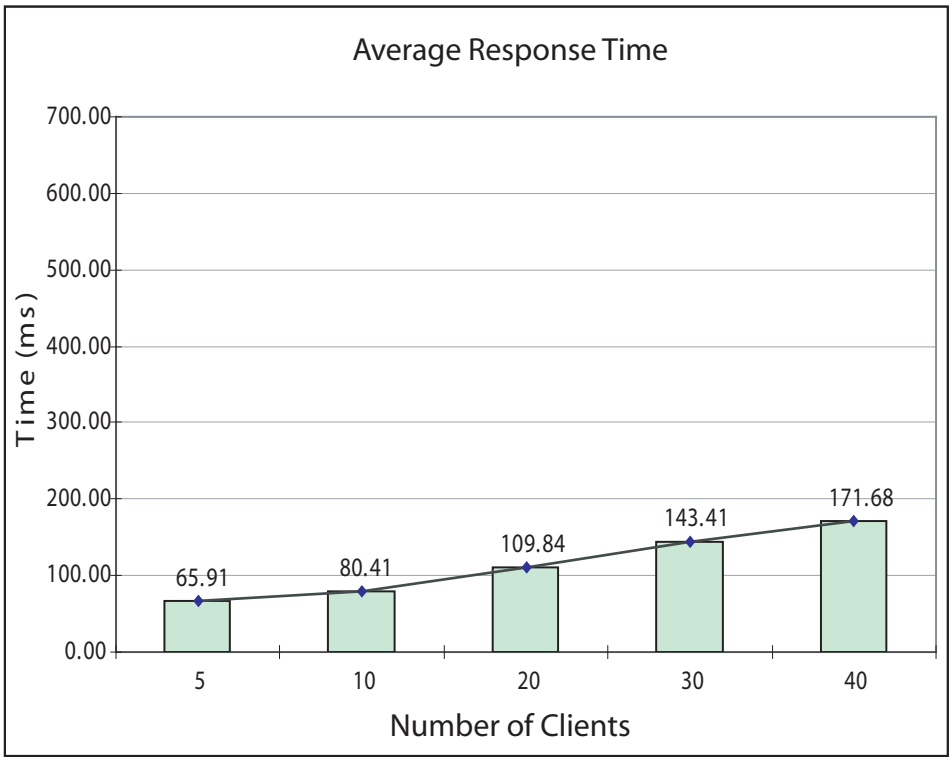



Figure 5.8: Average response time with cache holding up to 13% of the total records

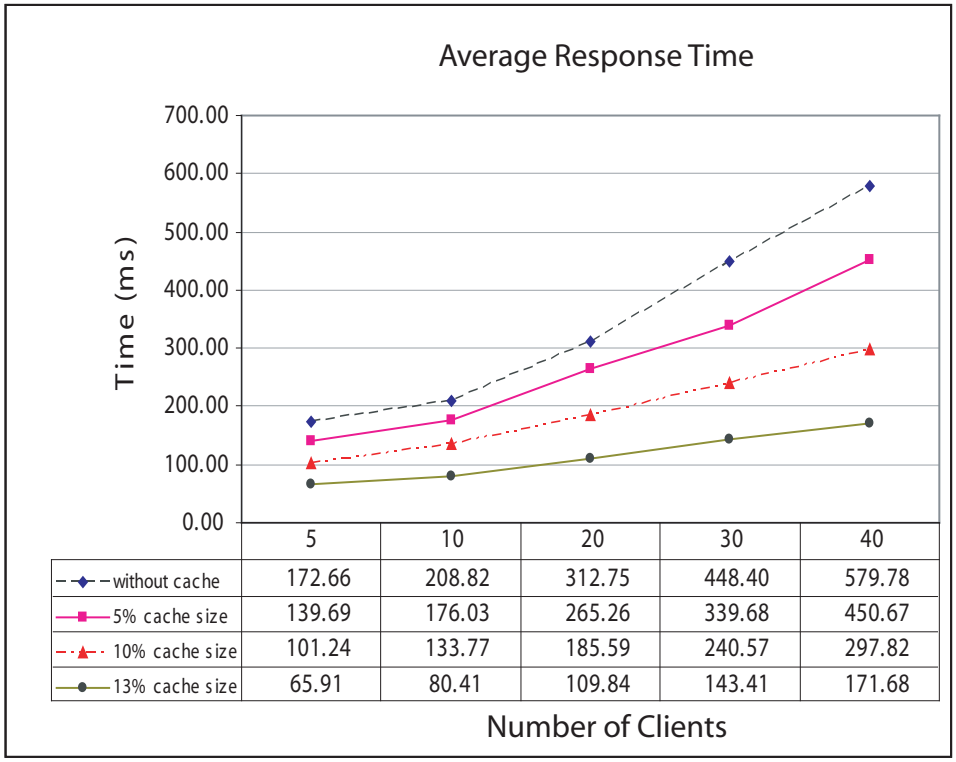



Figure 5.9: An ensemble of all cache varying test perfomed

From this experiment we can conclude that our system shows a hard dependency to the cache size; regardless of the configuration used, an increase in the cache size improves the average response time. This effect is more noticeable as the number of clients increases. Thus, this approach is recommended whenever the architecture under design has the capability of dealing with cached entries.

## 5.4 Measuring the Throughput in the System

The first set of experiments in this section was developed to measure the availability of the system using of our approach. The purpose of this set of experiments is to compare the throughput of our approach (i.e how fast metadata queries are processed) with other conventional approaches: centralized, fully replicated and partitioned. The setup configuration used by each catalog was the same as the one presented in section 5.2.

For each setup configuration, we ran experiments in which clients were constantly submitting metadata lookup requests in a random fashion to the system. We ran several trials using a varying number of clients: 5, 10, 20, 30 and 40, and each client ran by itself on a cluster node. Each trial would last 20 minutes and three experimental runs were performed for each architecture and client combination to calculate the throughput. We defined the throughput of the system as the number of metadata queries processed by time unit.

Figure 5.10 shows the resulting throughput for the configurations with five clients. This figure confirms the ideal performance of the fully replicated approach. This approach can process approximately six times more metadata queries per minute than the centralized approach, twenty times more queries per minute than the partitioned approach and thirty one times more queries per minute that Chord based approach. The fully replicated approach has the best performance due to each site having replicates all data and can therefore answer metadata queries independently of each other. The distributed approaches: Partitioned and Chord based approach have the worst throughput due to the inherent distributed nature that produces an overhead in communication.

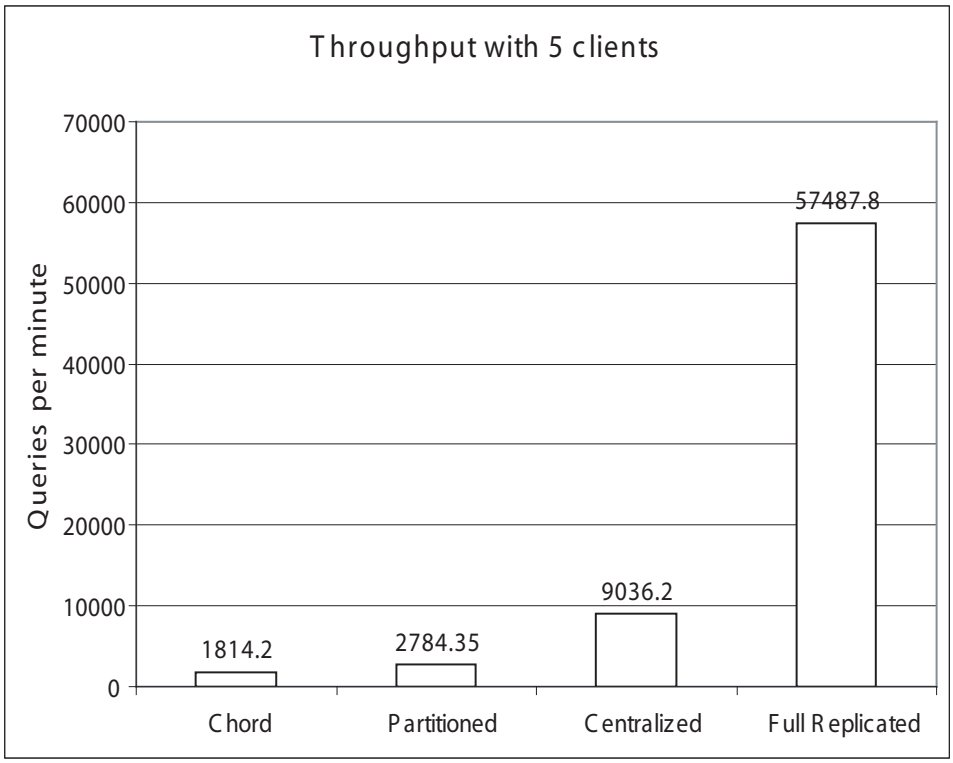



Figure 5.10: Throughput of catalog approaches with 5 clients

The same behavior is also present in the figures 5.11, 5.12, 5.13 and 5.14 when the number of clients are increased to 10, 20, 30 and 40, respectively. Although, the partitioned catalog approach has a better performance than the Chord based approach when the number of clients increase, it can be seen that both throughputs are very similar. This is due to when the number of clients are low the overhead of communication in Chord is relative high compared to the Partitioned approach. However, in the Chord approach if the number of clients is increased the impact of communication cost on the throughput is comparable to the Partitioned approach.

The results obtained from this set of experiments show that our Chord based approach presents the worst throughput regardless of the number of clients. In spite of these findings, we decided to use Chord as the catalog engine because the benefits provided by that approach, such as data location, data replication and fault tolerance out-weight its throughput limitations.

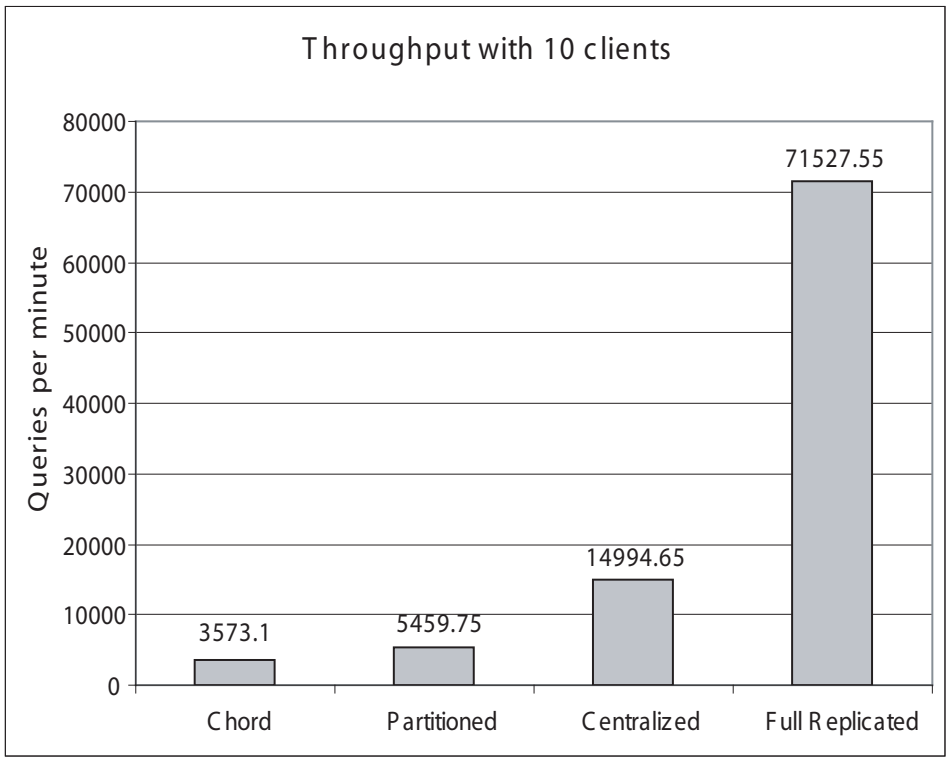



Figure 5.11: Throughput of catalog approaches with 10 clients

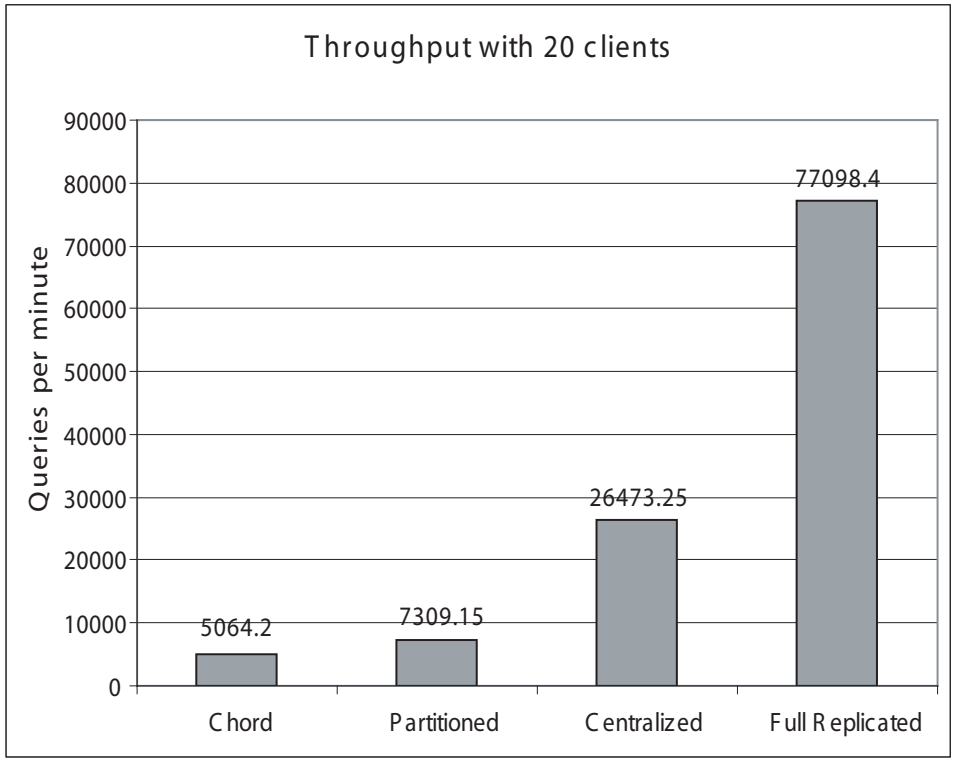



Figure 5.12: Throughput of catalog approaches with 20 clients

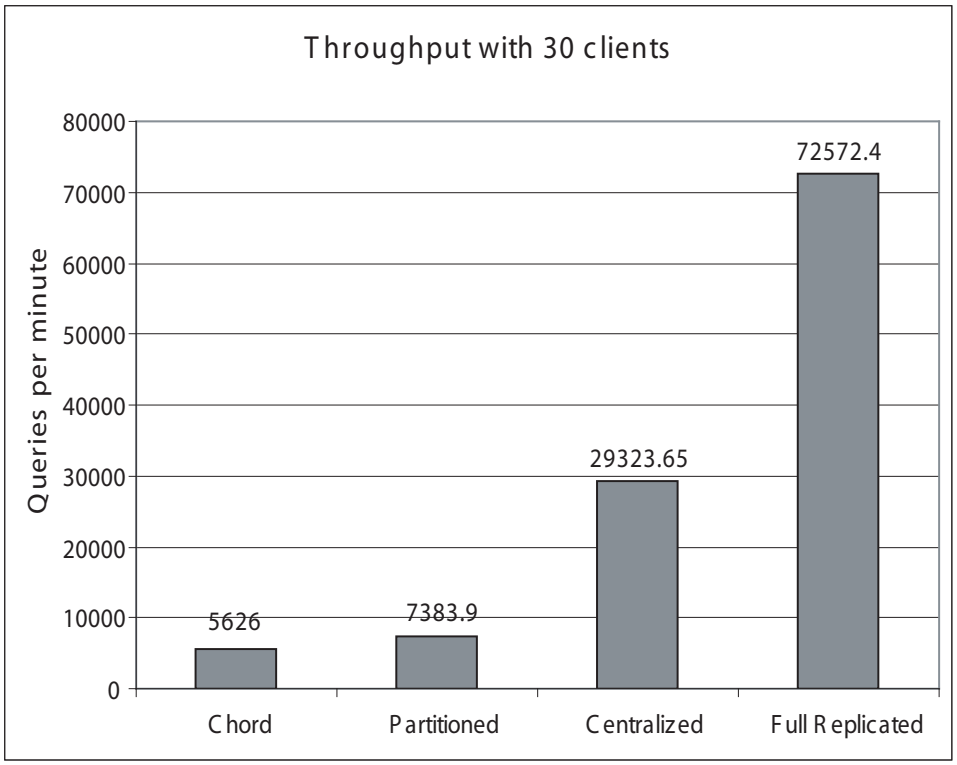



Figure 5.13: Throughput of catalog approaches with 30 clients

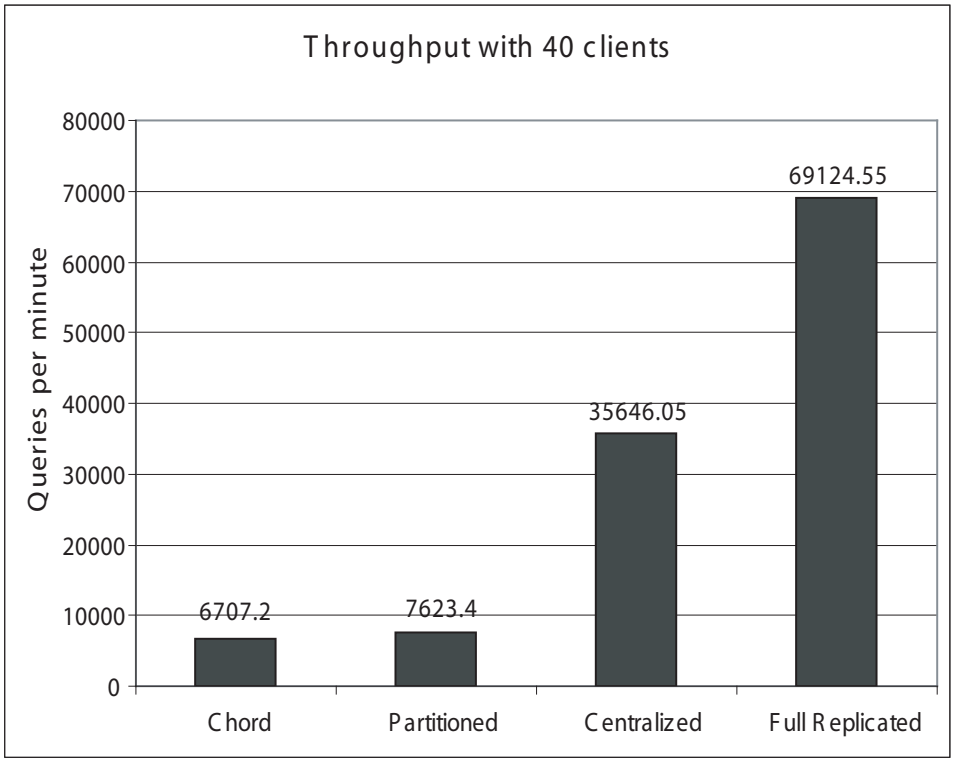



Figure 5.14: Throughput of catalog approaches with 40 clients

# CHAPTER 6

# Conclusion and Future Work

In this thesis we presented a *Peer-to-Peer Catalog Manager Scheme* for the NetTraveler middleware system designed to cope with the management of metadata used by the query optimizer and query processing modules. We proposed a system which provides an efficient metadata lookup service with several advantages over the other presented catalog manager architectures. First, our lookup service operates in a de-centralized mode, where no central authority controls the contents nor becomes a performance bottleneck. The nodes do not require global knowledge of all other nodes to locate a specific data entry, they only maintain a logaritmic-sized routing table (the finger table) to route and locate the data into the system. Second, this lookup service guarantees to a given metadata query that it will find the answer if the metadata exists in the system (deterministic location) in a bounded number of network hops.

We presented that on continuos changes of data sources, our catalog operates adequately, automatically adapting to events, such as joins, or departures of nodes without affecting the lookup service. When a new node $m$ joins the system, some entries are redistributed to the $m$ node and a few nodes need to update its finger table to reflect the the addition of $m$. The system is capable of dealing with graceful and abrupt departures. When a node $n$ leaves the system gracefully, the entries that hold $n$ are automatically redistributed to the rest of the members of the system and the finger table of few nodes are updated to reflect the departure of $n$. When a node $o$ leaves

the system abruptly, we use the replication mechanism to guarantee that the entries that $o$ holds cannot be lost. Additionally, the nodes are capable of detecting the abrupt departure and repair its finger table to maintain the correct routing location.

Amongst the approaches presented here we must recall the following conclusions. The nature of the centralized approach does not permit the joining or departure of nodes. In the partitioned approach the arrival of new nodes lead to re-distribution process of all entries between all participant nodes. Departures in this approach without a replication scheme are prohibitive due to the fact that node entries will be lost with their departure. In the fully replicated approach the arrival of new nodes implies the copying of the entire catalog to each new member of the system. If the catalog size is large the cost of maintaining each copy of the catalog can be prohibitive. Departures do not affect the catalog operation since the current replication in all nodes avoids the lost entries.

We decided to build a set of experiments to compare the availability of our approach with other traditional catalog approaches (centralized, partitioned and fully replicated) in terms of percentage of queries served in presence of node failures. The results of these experiments indicated that our approach has a better availability that other traditional approaches since the percentage of queries served in the presence of failures is 30 percent better than the centralized approach, and 20 percent better (on average) than the partitioned approach. Also these experiments indicated that our approach has a performance comparable to the ideal scenario of a fully replicated catalog approach.

We analyzed the behavior of the response time in presence of a variable number of clients. We detected an exponential growth of the response time when the number of clients is increased. This aspect motivated us to implement a data cache technique to reduce this behavior. We use several cache sizes to measure its impact on the response time. We decided to use a 13 percent of all data as our cache size due to in the presence of a large data set this cache size is a reasonable balance between resources usage and the resulting improvement in response time. With this cache technique we achieve a good behavior of the response time.

We also compared the throughput of our approach with the previously mentioned traditional approaches. We detected that distributed approach: Partitioned and Chord based approach have the worst throughput due to the inherent distributed nature that produces an overhead in communication. In spite of these findings, we decided to use Chord as the catalog engine because the benefits provided by that approach, such as data location, data replication and fault tolerance out-weight its throughput limitations.

## 6.1 Future Work

This section describe some of the aspects of the Catalog Manager system that were beyond the scope of this work, but will be important to consider in future work. The current work may be improved and extended in various ways:

- **Security**: our current system architecture assumes that the nodes involved are trusted. A malicious node can delete an arbitrary data without any control or it can generate a lot of entries and try to attempt to overload other nodes. For these situations, there is a need to implement a complete set of security architecture to deal with these problems.
- **Search facility**: Structured Peer-to-Peer network such Chord, only support exact matches of search key values (like Hash table). Our catalog manager could benefit from a search facility mechanism, that allows a user to retrieve entries with more complex predicates, such as: range predicates, and/or predicates and join predicates.
- **Other cache alternatives**: currently, our approach only supports one data caching technique. Other caching schemes can be implemented to improve response time in the system, such as *path caching* in which each node remember the path to locate a specific data instead of caching the data itself.
- **Performance**: in our work, we only consider performance aspect related to response time and availability. Additional performance test related to scalability and other partitioned data structures that would hold the information of the entries, reflecting a course granularity.

# BIBLIOGRAPHY


[1] Elliot Vargas and Manuel Rodriguez. Design and Implementation of the NetTraveler Middleware System based on Web Services. In *AICT-ICIW 2006*, Guadeloupe, French Caribbean, February 2006.

[2] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[3] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, Wilms P, and R. Yost. R*: An Overview of the Architecture. Technical Report RJ3325, IBM Almaden Research Center, San José, California, 1981.

[4] Michael Stonebraker. The Design and Implementation of Distributed INGRES. In *The INGRES Papers*. Addison-Wesley, Reading, Massachusetts, 1986.

[5] Michael Stonebraker, Paul M. Aoki, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, 1996.

[6] James B. Rothnie Jr., Philip A. Bernstein, Stephen Fox, Nathan Goodman, Michael Hammer, T. A. Landers, Christopher L. Reeve, David W. Shipman, and Eugene Wong. Introduction to a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):1–17, 1980.

[7] Laura M. Haas, Renée J. Miller, B. Niswonger, Mary Tork Roth, Peter M. Schwarz, and Edward L. Wimmers. Transforming heterogeneous data with database middleware: Beyond integration. *IEEE Data Eng. Bull.*, 22(1):31–36, 1999.

[8] R. Ahmed et.al. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*, pages 19–27, December 1991.

[9] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proc. of IPSJ Conference*, Tokyo, Japan, 1994.

[10] M.T. Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *23rd VLDB Conference Athens, Greece*, 1997.

[11] M. Rodríguez-Martínez and N. Roussopoulos. MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources. In *ACM SIGMOD*, June 2000.

[12] Oracle Corporation. Oracle Generic Connectivity and Transparent Gateways. http://www.oracle.com/technology/products/gateways/index.html, 2006.

[13] Sybase Corporation. Sybase Data Integration: Analyzing Your Options. White Paper.

[14] Mema Roussopoulos, Mary Baker, David S. H. Rosenthal, Thomas J. Giuli, Petros Maniatis, and Jeffrey C. Mogul. 2 p2p or not 2 p2p? In *IPTPS*, pages 33–43, 2004.

[15] Limewire. http://www.limewire.org/.

[16] Kazaa. http://www.kazaa.com/us/index.htm.

[17] Yoram Kulbak and Danny Bickson. The eMule Protocol Specification. Technical report, DANSS Laboratory, The Hebrew University of Jerusalem, Jerusalem, 2005.

[18] SETI@Home. http://setiathome.berkeley.edu/.

[19] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[20] Cosimo Anglano and Andrea Ferrino. Using chord for meta-data management in the n3fs distributed file system. In *HOT-P2P '04: Proceedings of the 2004 International Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P'04)*, pages 96–101, Washington, DC, USA, 2004. IEEE Computer Society.

[21] The Gnutella Protocol Specification v0.4. Document Revision 1.2.

[22] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[23] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 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*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[24] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329+, 2001.

[25] P.A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and Ilya Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. In *Fifth International Workshop on the Web and Databases (WebDB 2002)*, 2002.

[26] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. Peerdb: A p2p-based system for distributed data sharing, 2003.

[27] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. Miller, and J. Mylopoulos. The hyperion project: From data integration to data coordination, 2003.

[28] Bestpeer: A self-configurable peer-to-peer system. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 272, Washington, DC, USA, 2002. IEEE Computer Society.

[29] A. Kementsietsidis, M. Arenas, and R. Miller. Mapping data in peer-topeer systems: Semantics and algorithmic issues, 2003.

[30] N. I. of Standards and Technology. Secure hash standard. FIPS 180-1 Standard in U.S. Department of Commerce/NIST, April 1995.

[31] Panos Kalnis, Wee Siong Ng, Beng Chin Ooi, Dimitris Papadias, and Kian-Lee Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 25–36, New York, NY, USA, 2002. ACM Press.

[32] The mime multipart/related content-type. http://www.ietf.org/rfc/rfc2387.txt.

[33] Open chord library. open-chord.sourceforge.net/.

[34] Apache tomcat. http://tomcat.apache.org.

[35] Apache axis. http://ws.apache.org/axis/.