**RECOVERY OF CLIENT QUERY WORK IN THE NETTRAVELER MIDDLEWARE SYSTEM**

By

*Elliot Arturo Vargas-Figueroa*

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
2006

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 |
| Wanda Negrón, M.S.<br>Representative of Graduate Studies | Date |
| Isidoro Couvertier-Reyes, Ph.D.<br>Chairperson of the Department | Date |

# ABSTRACT

## RECOVERY OF CLIENT QUERY WORK IN THE NETTRAVELER MIDDLEWARE SYSTEM

By

*Elliot Arturo Vargas-Figueroa*

This thesis presents NetTraveler, a Database Middleware System for Wide Area Networks that is designed to efficiently run queries over sites that are either mobile clients or enterprise servers. NetTraveler is designed to cope with dynamic Wide Area Networks where data sources go off-line, change location, have limited power capabilities, and form *ad-hoc* federations of sites. This thesis presents how NetTraveler can continue running a query when the client posing the query leaves, and then delivers the query results when the client returns. This thesis presents an implementation experience of NetTraveler using Web Services. The ideas of the design and implementation choices of NetTraveler are validated with a preliminary performance study. This study shows that NetTraveler can complete more queries per unit of time than middleware systems that cannot gracefully recover from contingencies during query execution caused by a client leaving the system. This study also shows that NetTraveler can handle a larger query load when compared to solutions based on previous approaches.

# RESUMEN

# RECUPERACION DE CONSULTAS SOMETIDAS POR CLIENTES EN EL SISTEMA TIPO MIDDLEWARE NETTRAVELER

Por

*Elliot Arturo Vargas-Figueroa*

Esta tésis presenta a NetTraveler, un Sistema de Bases de Datos tipo "Middleware" para Redes de Cobertura Amplia que esta diseñado para ejecutar eficientemente consultas que accesan fuentes de datos que residen en clientes móviles o servidores de nivel empresarial. NetTraveler esta diseñado para manejar Redes de Cobertura Amplia que son dinámicas, y en donde las fuentes de datos se desconectan, cambian de localización, tienen limitaciones de potencia, y forman federaciones de servidores de manera *ad-hoc*. Esta tésis presenta como NetTraveler puede continuar la ejecución de una consulta cuando el cliente que somete la misma pierde su conexión de red, y como los resultados son entregados al cliente cuando éste regresa. En ésta tésis también se detalla la experiencia obtenida al implementar NetTraveler utilizando tecnología de Servicios Web. Las ideas del diseño y las decisiones de implementación sobre NetTraveler fueron validadas con un estudio preliminar de rendimiento. Este estudio demuestra que NetTraveler puede resolver más consultas por unidad de tiempo que un sistema de tipo "middleware" que no puede recuperarse de incontingencias ocurridas durante la ejecución de una consulta causadas por un cliente que pierde conexión al sistema. El estudio realizado también demuestra que NetTraveler puede manejar una carga de consultas mayor en comparacion con soluciones basadas en otras métodologias.

To parents Elida and Juan, brothers and sisters Enrique, Elida, Edgardo, Enid, Elvia, and to fiancé Maryliz.

# 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 the Virgin Mary for her support and comfort when I most needed it. To my fiancé Maryliz Gonzalez, thank you for your support, patience and love during all these years. Thanks to my thesis committee, specially my advisor Manuel Rodríguez for sharing with me his knowledge, time and friendship during all these years. I deeply thanks my friends and laboratory partners, specially Rene Badia and Juan Correa, for their support, motivation and help during the most difficult times of this research. I give special thanks to Miguel Pérez and Wallace López for being such wonderful house-mates, but above all for being my best friends for years. Finally, I thanks all my family, my parents Elida and Juan, my brothers and sisters Enrique, Elida, Edgardo, Enid and Elvia and my beautiful nieces Valeria, Ariana, Alaina, Elena and Eliana. You have all been an incredible motivation for me during these years and for that I dedicate this work to you.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# Introduction

## 1.1 Overview

With the broad acceptance of mobile devices in our modern society, companies have been forced to put more effort on the development of ever more powerful and accessible mobile devices. Figure 1.1 depicts some of the top-notch mobile devices on the market. This new breed of devices, ranging from laptops and Tablets PC to PDAs and smart cellphones, are designed with Internet connectivity in mind and are capable of running all types of popular applications such as web browsers, e-mail applications, word processors, spreadsheets and video games. However, the truly revolutionary impact of these devices will come when they enable their users to have ubiquitous access to the Internet from virtually anywhere a person can go. We are seeing now that the ever increasing affordability of broadband and wireless technologies is making this once utopian dream a tangible reality.

We are seeing that mobile computing devices have a dual role for their users as client of modern networked environments. First, they continue to act as the client sites that run the applications used to obtain content from the Internet. Second, they have become valuable sources of information that users need to carry out with them as they move across geographic locations. We can find data sources containing very diverse data sets such as MP3 files, lists of appointments,

Figure 1.1: Modern mobile devices

phone directories, sensor readings, relational data, and XML files, among others. Therefore, new distributed systems that support seamless data access to these sources are needed. In particular, we will need data integration systems to organize mobile sites into a coherent wide-area information system, which also incorporates more traditional data sources (e.g. relational databases) that are residing on enterprise servers.

Unfortunately, current data integration solutions are inadequate to support efficient data integration of mobile clients over dynamic Wide-Area Networks (WANs), such as the one depicted in Figure 1.2. On one side, most of the work done to support data processing for mobile clients view clients as "couch potatoes", whose only role is to help the users digest data periodically produced by various data sources [1]. Work on mobile database systems, for example [2], has only dealt with the problem of replicated data synchronization, and supporting transaction in the presence of databases that might be off-line at the moment that a transaction related to them is to be committed. The issue of efficient and reliable query processing for mobile databases has been just barely scratched. On the other side, existing database middleware solutions for data integration [3, 4, 5] are designed for rather static systems, based on enterprise servers that are always on-line, on a fixed-location, with continuous power sources, configured and administered by teams of professionals, and organized into slow-changing federations of cooperative sites.

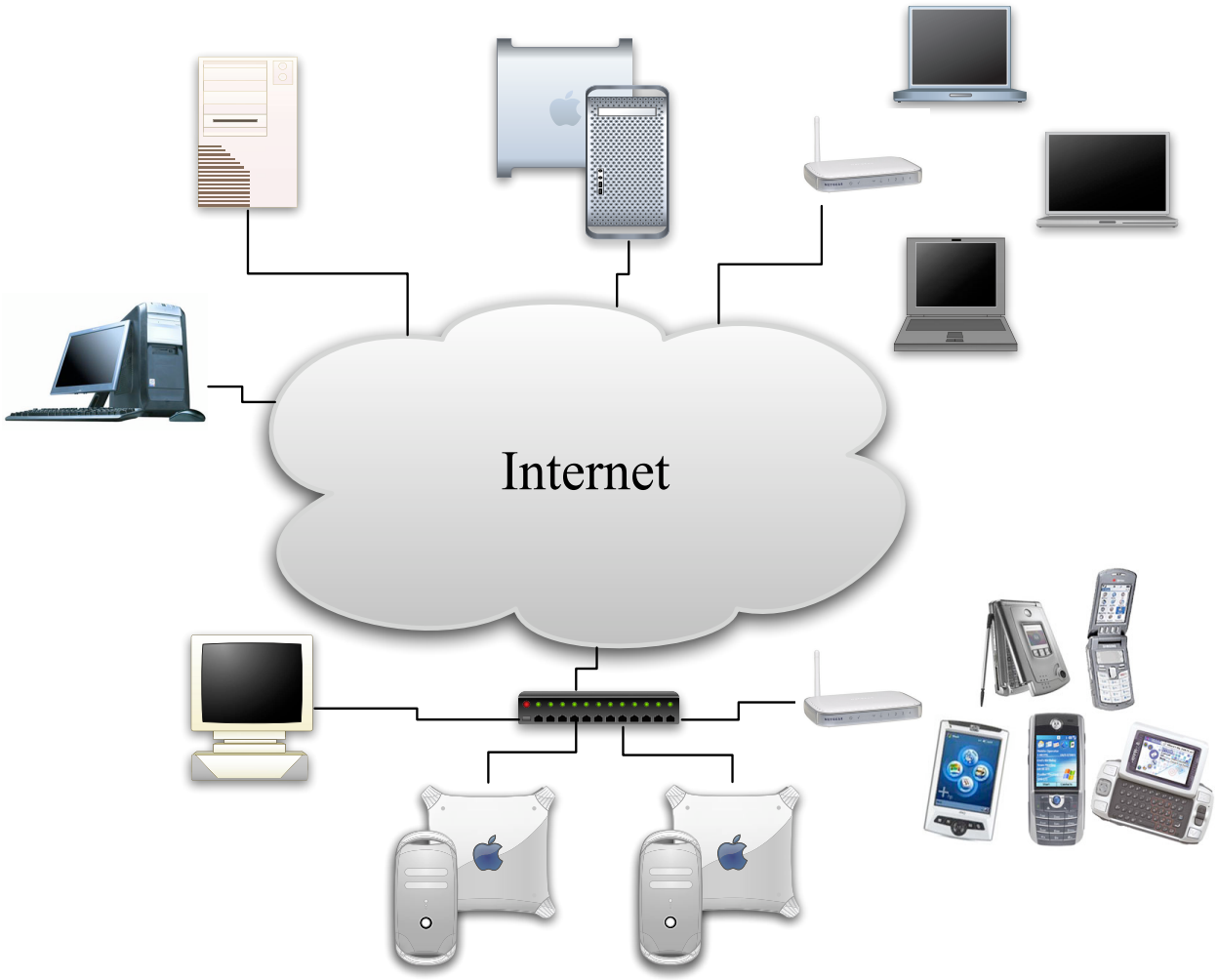



Figure 1.2: Next generation WAN

This thesis presents NetTraveler, a Database Middleware System for WANs that is designed to efficiently run queries over sites that are either mobile clients or enterprise servers, taking into consideration the nature of clients and hosts for query processing purposes. NetTraveler is designed to cope with dynamic WANs where clients and data sources go off-line, change location, have limited power capabilities, and form ad-hoc federations of sites that work together to complete a given task and then go about their business independently. NetTraveler treats mobile devices as bonafide data source sites and copes with the realities of their environments.

## 1.2 Problem Statement

Mobile devices have special characteristics that make them different from typical (e. g. Desktops) middleware client devices. They have power limitations, intermittent connectivity, can move across locations and may get connected through pricey Internet links. Therefore, if mobile devices are to be fully supported in Database Middleware Systems, there is a need for middleware that can take into consideration the limitations of these devices and that can find ways of attending their request in an appropriate fashion. Most of the existing Database Middleware Systems treat all query requests with the same policies, regardless of the nature of the computing device that is holding the client application. In this sense, the middleware has little awareness of the specific requirements that clients have due to the nature of their software, hardware or network link.

Execution engines in current middleware solutions have no knowledge of the capabilities of the client posing a query and will abandon all efforts in solving a query at the first sign of error. This leads to slow response times as queries need to be re-started from scratch. This results in wasted resources, a decrease in system throughput (defined as the amount of queries solved per unit of time) and an overall unsatisfactory user experience. Moreover, if by any chance the client device was using a pricey internet link, like a 3G connection, the user will also end up paying extra money due to the amount of extra time needed to complete the query. In an environment where a considerable number of clients are expected to reside in mobile devices, disconnection failures are expected to occur often due to clients losing battery power, going temporarily offline or changing

network location. That is, moving from one Local Area Network (LAN) to another LAN in the same WAN.

The work developed so far that concerns query execution in Database Middleware Systems has been geared towards finding efficient ways of executing queries given the distributed nature of the system. However, none of the existing middleware systems have dealt with the problem of also finding efficient ways to recover the query work performed before a failure in the system. Because the execution engines of middleware systems have resembled query execution engines of classical Database Management Systems (DBMS), they have not been designed with the idea of mobile clients in mind. This traditional middleware query execution scheme is not fitted for a system that expects to have a large amount of clients residing in mobile, power-limited devices.

## 1.3 Proposed Solution

The NetTraveler Database Middleware System is currently being developed at the University of Puerto Rico, Mayagüez Campus. It is being designed as a vehicle for the research of three specific issues within the realm of middleware applications: 1) **Context-aware query processing**, 2) **Server-side and Client-side query recovery** and 3) **System self-configuration**. Context-aware query processing deals with the issue of finding the most efficient plan for solving a query, taking into consideration the characteristics of the client device that is submitting the query. The work done so far related to context-aware query processing has dealt with the issue of solving a query based on geographical position of the client. NetTraveler intends to go beyond this paradigm and take into consideration other parameters, such as the physical properties of a device or specific rules set by the client, at the moment of query execution. For example, a query in NetTraveler might look like this: *Find the nearest restaurant to my current position taking into consideration that I am a PDA with 25% of battery left and that I am using a pricey Internet link.* In this case, the system will try to solve the query as soon as possible, or will generate the results for the client while the client is offline and will deliver the results when the client returns.

Server-side and client-side query recovery deal with the problem of completing the computation of a query when a server or a client goes offline in the middle of query execution. Current middleware systems will abandon the execution of a query at the first sign of error, causing the lost of the system resources that were invested in the computation. If the query has to be completed, its computation must be re-started from scratch. NetTraveler is intended to detect errors that occur in the middle of query execution, recover from those errors and complete the computation of the query. Recovery of the query work will allow NetTraveler to complete a query without the need of re-starting it.

System self-configuration deals with the issue of forming networks and execution environments of mobile devices, in combination with full-fledged servers, in automatic fashion. Automatic and efficient mechanisms are needed for the formation of reliable ad-hoc networks for areas lacking a network infrastructure, such as a disaster site. Relying on site administrators to configure and manage such ad-hoc execution environments will be in most cases impractical, if not folly.

By attacking these three specific areas, NetTraveler is intended to deal with the arrival of mobile devices as important players in wide-area environments. NetTraveler is being built on top of the Soap communication protocol using Web Services technology, and is the first middleware system implemented with a connectionless oriented architecture. A connectionless oriented architecture can help in improving the evaluation of queries where parallelism can be exploited in distributed systems. More importantly, a connectionless oriented architecture has benefits over its counterpart in an environment where many of the clients posing queries reside in mobile devices, since a disconnection failure will not necessarily imply the abortion of a query. By having a connectionless architecture, NetTraveler is able to brake the paradigm of connection oriented query engines in middleware architectures and be the first middleware system with the notion of "persistent queries". That is, queries that are active (persist) beyond disconnection failures.

This thesis focuses on the issues related to **client-side query recovery** and "persistent queries". The issues of context-aware query processing, server-side query recovery and system self-configuration are being, or will be, researched apart from this work. Therefore, they have being

left out of the scope of this thesis. In this thesis we shall see the components needed to allow the recovery of the work of a client that suffered a disconnection in the middle of query execution. NetTraveler is composed by 5 server applications:

1. *Registration Server* (RS) - This server application advertises metadata that describes resources such as: query operators, local tables, global tables, data types, CPU cycles, data sources, network bandwidth, and so on. The RS is responsible of coordinating automatic configuration of clients and servers that enter and leave a NetTraveler federation. Also, two or more RSs work as peers to exchange metadata.

2. *Query Service Broker* (QSB) - This server application takes on the responsibility of finding the computational resources to extract data from the data sources to answer the queries posed by the clients. QSBs perform query related tasks and exhibit Peer-to-Peer functionality.

3. *Information Gateway* (IG) - This server application has the role of providing access to the wealth of information contained in the data sources.

4. *Data Synchronization Server* (DSS) - This server application helps clients in caching query results, obtaining extra disk space, and keeping synchronized copies of data natively stored by a client which also happens to behave as a data source from time to time. A DSS can become a *proxy* for a client, gathering the results intended for the client if the client goes offline or experiences some type of failure.

5. *Data Processing Server* (DPS) - This server application provides a commodity service for computational tasks during query processing. These tasks include query execution, sorting, or any other type of specific computational operation required.

In NetTraveler, the issue of client-side query recovery is addressed by leveraging on the system architecture to enable the *rerouting* of the client-side execution of a query. The goal of this thesis is to build the necessary infrastructure to investigate if *a middleware system capable of re-starting queries that have been interrupted due to a disconnection failure in the client will allow for a greater throughput, defined as the amount of queries solved per unit of time, than a system that lacks this capability.*

NetTraveler is currently a read-only system, only queries that retrieve data from the data sources are supported. Updates queries are not supported since distributed updates are complex operations that are still being researched. The topic of distributed updates is out of the scope of this thesis. For the purpose of this work, it is assumed that each data source is administered independently and that administrators take on the responsibility of updating the data sources.

## 1.4 Possible NetTraveler Applications

In this section, we shall see two sample scenarios that show possible applications that could be built using the NetTraveler Middleware System. The scenarios are purely conceptual, and a lot of technical implementation details are omitted for simplicity. The main purpose of these examples is to help visualize the usefulness of NetTraveler.

### 1.4.1 City Guide Application

In this first scenario, we have a city that is covered by a broadband network managed by the city administration. The administration has decided to implement an interactive city guide application based on NetTraveler. The administration will provide the Registration Servers (RSs), the Query Service Brokers (QSBs) and some Data Synchronization Servers (DSSs). Other DSSs will be provided by private entities that might impose fees for extra services, like differentiated disk space sizes, and bandwidths greater than those provided by the city administration. Information Gateways (IGs) will be run by businesses, like restaurants, hotels and coffee shops, to expose their services and products in the city guide. Examples of possible services or products to advertise are dinner plate specials or rebates for electronic devices.

NetTraveler enabled client devices will automatically register themselves with an RS in the city guide application when they enter the city federation. Once a client is configure, an interface will be used to submit queries to the system and search for available services and products. Since many of the clients of the city guide application are expected to be handheld devices like PDAs

and cellphones, context-aware query processing can be used to enhance these clients experience. For example, a user could search for services that are close to its geographical location, or could specify that results should be returned under 1 minute or else the query should be aborted.

Let us assume that one of the services provided by the city guide application searches for dinner plates at restaurants, makes a comparison between them based on parameters specified by clients, and returns results sorted by relevance. Assuming that a client asks for a comparison between the available dinner plates of all the american food restaurants in the city, and that the city is medium- to large-sized, such query is likely to access an amount of data sources ranging in the hundreds, which could make this query to take a considerable amount of time to complete. It is unlikely that a user with a cellphone will wait for more than a minute for this query to complete. In part because the user is most likely being charged extra money for accessing the Internet with the phone, but mainly because waiting for a long period of time for this type of query is not worth the hassle. After all, there are restaurant in almost every corner of a city, and appetite preferences are easily suppressed.

Nonetheless, let us suppose that a female tourist has planned a day trip that ends with dinner at an american food restaurant. With NetTraveler, she can submit the query in the morning, and specify that she wishes the results to be handled by her DSS. The user will submit the query, go about her day trip and, when dinner time is at hand, communicate with the DSS and retrieve the results from it. The user could narrow the search further once the results are cached in the client device. For example, the user can ask the client application to show only restaurants that are close to her current location.

Finding dinner plates is just but an example of all the services that can be provided by a NetTraveler-based city guide. Another service could be to have super markets to expose their products and be able to compare the prices of meat, vegetables and other goods. A user could submit the query before going to work in the morning, and then retrieve the results from her/his DSS in the afternoon and decide the best market to buy from after work.

### 1.4.2 Mail Delivery Tracking System

In this second scenario, we have a mail delivery company that has decided to track their vehicles during the coverage of their delivery routes. The company is interested in monitoring the current position of vehicles (and therefore of the packages they carry) not only when they arrive at a delivery/pick-up station, but also when the vehicles are on the road. By implementing this system, the company expects to measure, with good accuracy, the efficiency of their delivery system and also expects to have a more accurate feedback system for itself and for its clients.

The company decided that each driver will carry a NetTraveler enabled handheld device with satellite Internet connectivity and a *Global Positioning System* (GPS) receiver for the purpose of implementing this system. The status of each package (e.g. delivered, on-route, etc.) on each vehicle will be stored in a database on each device. Information Gateways (IGs) will be run in the devices to give access to the information in their databases. The purpose of the devices is hence twofold: (1) using the GPS in the devices, a map system will built and images of the vehicles will be superimpose to have a visual image of the current position of each vehicle and (2) the system will periodically query the devices in the vehicles to gather the status of packages. This way the company expects to have almost realtime status of their vehicles and of packages. Using the devices in the vehicles as data sources is different than the "classic" approach where devices broadcast their status periodically whenever they get network connectivity. With NetTraveler, devices can be queried as needed, avoiding the waste of resources in terms of bandwidth and battery consumption caused by broadcast systems in mobile devices.

It is worth nothing that the data sources in the vehicles are mobile, and might be unreachable when the mail delivery tracking application tries to access them. To deal with this issue, IGs running in the mobile devices will register themselves with the nearest Registration Server (RS) in the mail delivery tracking application when they enter the federation. Every time the IGs register, network routes for each device will be calculated to allow the devices in the vehicles to be queried. Also, Data Synchronization Servers (DSSs) will be used to cache the data from the devices in the

vehicles every time they enter the NetTraveler federation of the mail delivery tracking system. This way, if a data source is not available at the moment of query execution, data from a DSS can be used instead of the original data source.

The mail delivery tracking system will update the status of vehicles periodically in an automatic fashion. Maybe every hour, the system will update the current location of all vehicles using GPS and queries will be submitted to update the status of packages. If a query is submitted to find the status of the packages being carried by a specific vehicle, and the device in the vehicle is unreachable, and there is no DSS replicating the information of the device, the query can be put in a suspended state until the device can be reached. This way, queries do not need to be re-submitted for every vehicle that is unavailable. Clients of the mail delivery company could see the exact location of their packages by using a Web-based interface. Nonetheless, for security reasons, the mail delivery company could decide to show approximate locations and packages status through the Web interface and use accurate locations and package status for internal purposes.

## 1.5 Objectives of this Thesis

The main goals of this thesis are:

- The design and implementation of the first components of the NetTraveler Middleware System that will serve as the building blocks upon which the rest of the components, algorithms and proposed objectives of the NetTraveler project will be developed.
- To implement the first **connectionless query execution** for a Database Middleware System. This query engine will be used in further research issues concerning NetTraveler.
- The implementation of a client-side query recovery mechanism based on the connectionless execution engine. This mechanism will enable a client that has suffered a failure that forced it to lose network connectivity to complete the execution of a query without re-starting it.
- Investigate if a Database Middleware System capable of recovering the query work of a client is able to solve more queries per unit of time than a system without this capability.

## 1.6 Contributions

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

- The NetTraveler Architecture is presented and arguments are made about its benefits for supporting query processing in Wide Area Networks. It is shown how NetTraveler can continue running a query when the client leaves, and then delivers the query results when the client returns.
- An implementation experience of NetTraveler using Web services is presented. To the best of the author's knowledge, no other query execution engine has been implemented in this fashion.
- A preliminary performance study is carried out to start validating the design and implementation choices of NetTraveler. This study shows that NetTraveler can complete more queries per unit of time than middleware systems that cannot recover from contingencies during query execution caused by clients leaving the system. This study also shows that NetTraveler can handle a larger query load when compared to solutions based on previous approaches.

## 1.7 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 presents a survey of the more relevant work related to this thesis. In Chapter 3, the NetTraveler system is presented from an architectural point of view and each of the NetTraveler components is introduced and explained in detail. Chapter 4 discusses the implementation details of NetTraveler, emphasizing in the implementation of the *client-side query recovery*. Chapter 5 presents the experiments and the results obtained. Finally, Chapter 6 presents the final 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, Database Middleware Systems and Peer-to-Peer Systems. An extensive amount of work has been carried out in these fields, hence this chapter discuss only the works that are more important to this thesis.

## 2.1 Distributed Database Systems

A Distributed Database System is form by a collection of Database Management Systems (DBMS) that are physically apart and connected via a computer network. These Database Management Systems have agreed to form a federation of sites that share their collections of data and query processing capabilities. Several Distributed Database Systems have been prototyped in the last decades; from among the most interesting we have R* [7] by IBM, Distributed Ingres [8] developed by the Univeristy of California at Berkley, and Mariposa [9] developed by the Univeristy of California at Berkley. Figure 2.1 shows the typical architectural organization of a Distributed Database System.

In general, the following assumptions are made by the majority of the Distributed Database Systems:

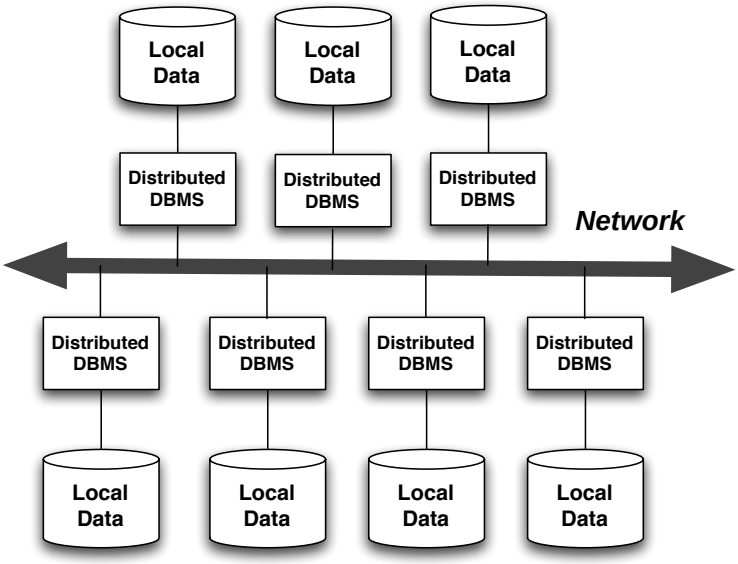



Figure 2.1: Typical Distributed Database Architecture

- All sites are independent from one another and autonomous. No central authority dictates what data should be stored at each site or how it should be accessed.
- All sites follow the same data model, typically the relational model.
- All sites run the same DBMS software or a DBMS for which there is a common communication protocol like ODBC [10].
- Users should be able to ask queries without knowing or specifying where the relations are located.
- Users should be able to perform transactions that affect data at several sites just as they would execute transactions over a local Database Management System.
- The effects of a transaction across sites should be atomic; all the changes persist if the transaction commits and none persist if it aborts.

The early Distributed Database Systems, like R*, assumed that all sites were full-fledged database servers. Latter on, an approach in which the DBMS was divided into a *client-server* architecture was introduced. In this architecture the server, or *back-end* component, is responsible for managing the data and executing transactions. Meanwhile, the client, or *front-end* component, has the role of interacting with users, requesting data from one or more servers. Servers are run in machines

with very fast disks, multiple processors and lots of memory, while clients are run on a vast array of machines ranging from workstations all the way to handheld devices. Almost all of the existing commercial DBMS (Oracle, IBM and Microsoft) follow this architecture. This *client-server* DBMS architecture became the most studied in the past two decades because of its simplicity of implementation due to the clear separation of functionality between client and server. The bulk of the work done has been related to server-side replication and client-side caching techniques ([11, 12]). The idea behind these techniques is to minimize communication costs by reducing network access.

Query optimization techniques in Distributed Database Systems have resembled that of traditional DBMSs, pioneered by the System R prototype [13] developed by IBM. This optimization strategy, described in [14], is based on dynamic programming and follows a *cost-based* model to calculate total resource consumption for a query. Each operator in the plan is given a cost and the overall cheapest plan, calculated by adding the cost of each operator, is always selected. In a typical, single-site DBMS the cost estimate is dominated by disk access time. In a distributed environment, however, other factors such as communication costs and differences in local computational costs must also be taken into consideration. The R* prototype system was one of the very first to introduced these additional factors into the cost model. The "classic" cost-model has been proven useful in optimizing the overall throughput of a system. However, this type of optimizer will not always find the plan with the lowest response time for a query in cases where the machines are lightly loaded and the communication channel is fast, since it cannot take into consideration intraquery parallelism [15]. Intraquery parallelism occurs when a query plan has several operators that can be evaluated in parallel because they can be evaluated at different sites. Figure 2.2 shows a plan that exhibits intraquery parallelism. Another model that does take into consideration intraquery parallelism is the *response-time* model [15]. This model has been successfully used for query optimization in [16].

The main issue in distributed query execution has been finding the optimum location for the execution of each query operator. The first Distributed Database Systems started using two techniques to evaluate operators: *query shipping* and *data shipping*. In *query shipping* all

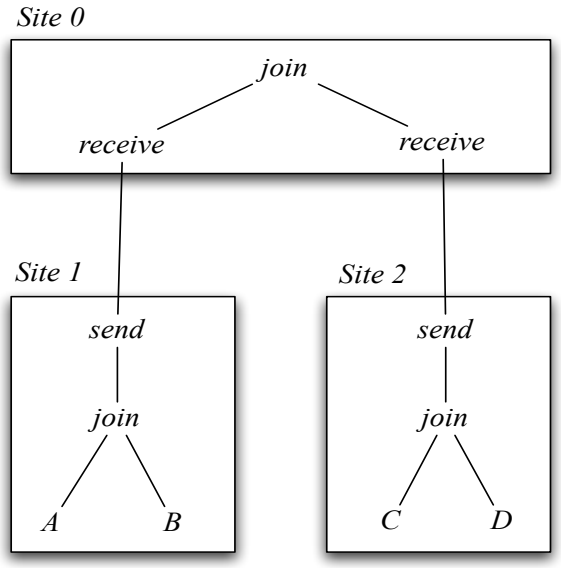



Figure 2.2: Distributed query plan that exhibits intraquery parallelism

operators are executed at the site where the data resides and join operations between relations stored at different sites must be evaluated using some distributed join algorithm like the *semi-join* [17] or the *2-way semi-join* [18] operators. Query shipping reduces the amount of data sent over the network but might cause the servers to become overloaded because the amount of data processing. The exact opposite of this strategy is *data-shipping* where all operators are evaluated at the client machine. All data is gathered from the servers and cached in the client's main memory or disk. Cached data is used to execute subsequent queries, off-loading the servers, but increasing the amount of load in the communication network because the data is being transmitted "raw". The work in [16] argues that none of this strategies is the best policy in all situations and combines both approaches into a *hybrid-shipping* architecture. Experiments showed that the performance of *hybrid-shipping* is superior, or at least equal, to that of the other two policies. Garlic [5] and MOCHA [3] are both systems that employ *hybrid-shipping*.

## 2.2 Database Middleware Systems

The goal of a Database Middleware System is to integrate *data sources* with very different characteristics in terms of data model, database server and query processing capabilities. Database

Middleware Systems follow an architecture that provides seamless access and an unified view of the data. This means that the data sources accessed by a single Database Middleware System might include relational database servers, object-relational database servers, file systems (e.g. Mac OS, Linux, Windows), Web servers, image servers, and many others. The middleware architecture is designed to allow a single query to span these data sources without requiring every source to be capable of managing multi-site execution strategies. Middleware systems have been attractive when integrating legacy systems or systems whose capabilities cannot be further extended. Figure 2.3 depicts the architecture commonly employed by Database Middleware Systems.

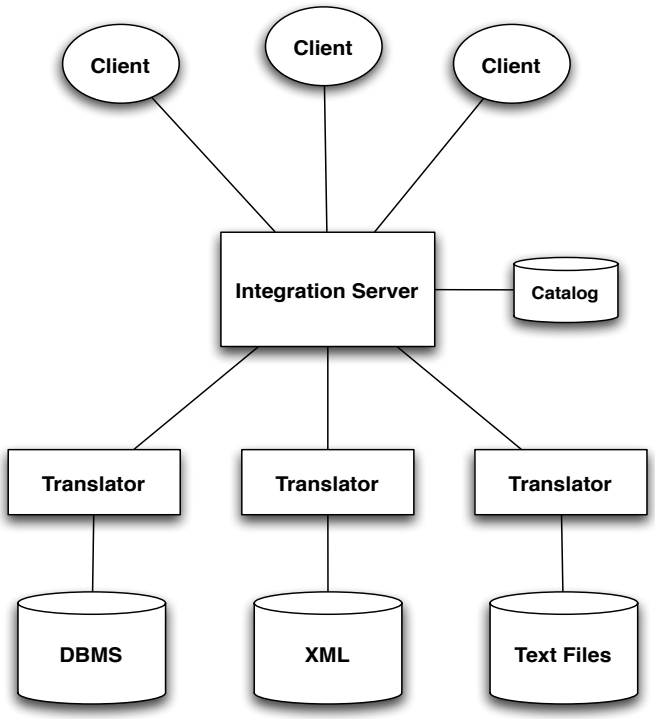



Figure 2.3: Typical Middleware Database System Architecture

The Database Middleware System is implemented as a layer of software between the clients and the data sources. This layer of software is a special server, typically called the *integration server*, that coordinates the execution of queries over several independent data sources. The integration server typically executes relational query operators (e.g. joins, selections, projections) over the data obtained from the sources. In its most basic form, the middleware can be made by software

modules known as *database gateways*. A database gateway allows a single-site DBMS to access data managed by other database servers, typically from other vendors, and used this data as if it has come from its local tables. Thus, the DBMS acts as the integration server for its clients. Database gateways are provided by the major database vendors such as Oracle [19] and Sybase [20].

Another type of middleware implementation is the mediator type. In this system, 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 also provides a common data model required to resolve conflicts between different schemas used by the data sources. To retrieve the data from the data sources the meditator uses another piece of software known as the *wrapper*. 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. Data retrieved from the data sources are translated from the local schema of the data source to the global schema imposed by the mediator before being transferred to the mediator for further processing. Some example of mediator systems are Pegasus [21], TSIMMIS [4], Garlic [5] and MOCHA [3].

Query optimization has not received as much emphasis as data integration in the context of middleware applications. The vast majority of the query processing typically occurs at the integration server with little or no data processing at the data sources. The most extensive work in query optimization in middleware systems has been performed by the Garlic and the MOCHA projects. In Garlic, a dynamic optimization framework in which the wrappers cooperate with the mediator to determine the capabilities of a data source is proposed. Any operator that the data source can execute is "pushed" for execution at the data source and operators that are not available at the data source are evaluated by the mediator. In MOCHA a cost-based optimization is used where each query operator is assigned a cost named the *Volume Reduction Factor*. This factor determines the amount of data to be transmitted through the network after applying a query operator. If the amount of data after applying a query operator is less that the amount of

data before, the operator is evaluated at the data source. Meanwhile, if the amount of data after applying a query operator is greater that the amount of data before, the operator is evaluated at the mediator. MOCHA employs an extension of the *hybrid-shipping* query evaluation technique called *code-shipping*. In *code-shipping*, the code needed to execute an operator is sent to the data source if the data source lacks the capability to execute the given operator.

## 2.3 Peer-to-Peer Systems

In a Peer-to-Peer (P2P) system there is no centralized authority that dictates how the resources of each computer in the network are utilized. Each computer is totally independent from the others and computers can communicate, exchange information and share resources with each other without the need of any type of centralized coordination authority. A computer in a P2P network can act as a server at a given time and as a client at another time. Furthermore, because P2P systems lack a centralized server they minimize system bottlenecks and eliminate single points of failure. Depending on how the P2P network is constructed it can be *unstructured* or *structured*. An unstructured P2P network is formed in a pure *ad hoc* fashion using a protocol like Gnutella [22]. A structured P2P network is formed by following strict rules, and the resulting topology of the network is one that is controlled. Chord [23] is a P2P system that exhibits a structured formation.

P2P systems have been mainly used in the context of *content distribution*. A P2P content distribution system "creates a storage medium that allows for publishing, searching, and retrieval of files by members of its network", as defined in [24]. Many content distribution systems are currently deployed, and by far the most famous are used for file sharing. Examples of such systems include Limewire [25], Kazaa [26] and eMule [27]. Another type of P2P system that has also gain much popularity is related to distributed computation. Distributed computation P2P systems are aimed at taking advantage of available CPU time by breaking down a computer-intensive task into small units of work which are distributed to different peers in the network. Each peer will compute a given task and return the results to the task originator. The most famous of this type of P2P systems currently deployed is the Seti@home [28] project.

In the context of query processing, P2P systems have been used to perform searches over wide area networks organized as very large P2P networks. The work in [29] proposes a P2P system for searching data over the Internet using an XML search engine. Nodes in the network exchange messages and advertise their data contents using XML messages. Peers maintain indexes (the *peer index*) of data stored in other peers to speed searches up. A peer in the network cannot know every other peer in the network for scalability reasons. Hence, the number of peers that each peer in the network knows is limited by enforcing *horizons*. The horizon of a peer is the maximum number of peers with which it will communicate. Other works related to databases and P2P networks include the work in [30]. Here, the authors proposed the *Local Relation Model*. In this model it is 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. The main goal of this data model is to support semantic interoperability in the absence of a global schema.

# CHAPTER 3

# NetTraveler Architecture

## 3.1 Overview

This chapter presents the NetTraveler Middleware System from an architectural point of view. It starts by presenting how Wide Area Networks (WANs) are modeled in NetTraveler. Then, the NetTraveler components and their corresponding roles are explain in detail. Finally, this chapter discusses the query execution framework of NetTraveler and gives an overview of query execution.

## 3.2 Architecture Overview

NetTraveler is a middleware system designed for Wide Area Networks (WANs). WANs are modeled in NetTraveler as a collection of applications $H = \{h_1, h_2, ..., h_n\}$, each having a specific role in helping a client to solve a given query. The collection of applications $H$ is running on host computers spread over a group of LANs that conform the entire WAN environment, as shown in Figure 3.1. These LANs can be made of wired or wireless technologies, such as Ethernet, DSL, IEEE 802.11b, and 3G networks.

From the set $H$ we have a subset of applications $C = \{c_1, c_2, ..., c_i\}, i < n$, which have

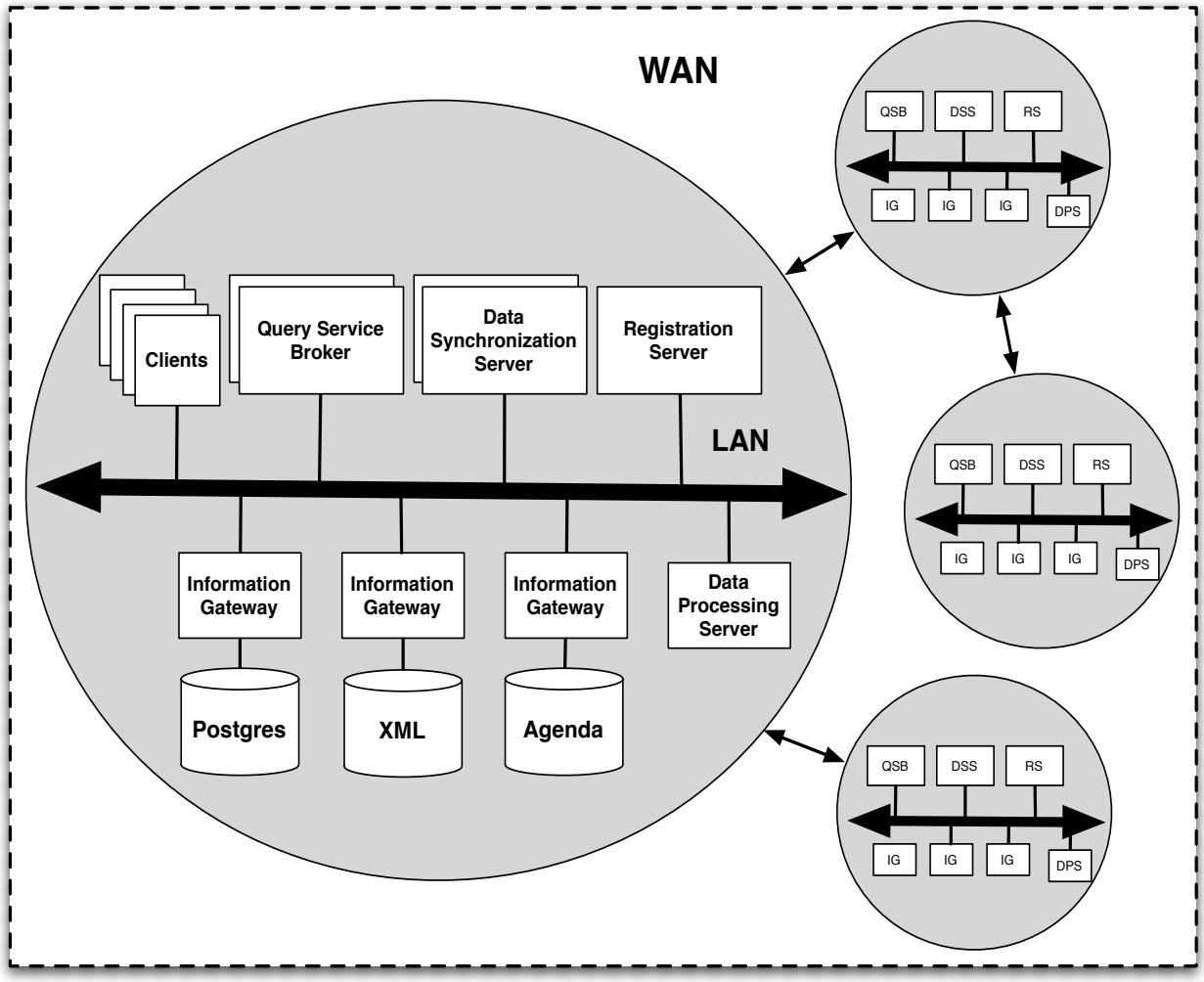



Figure 3.1: NetTraveler Architecture

client capabilities to submit queries. These capabilities come from running a given interface (most likely a GUI) that the end-user can use to pose queries to the system. The data to answer those queries comes from another subset $S \subset H$ known as the *data sources* $S = \{s_1, s_2, ..., s_j\}, j < n$. Each data source $s \in S$ is an application such as a DBMS, Web Server, XML-based data server, or some other customized server application. When a client $c \in C$ needs to pose a query to sources in $S$, it needs to contact a server application known as the *Query Service Broker*.

A collection of Query Service Brokers (QSBs) $B \subset H, B = \{b_1, b_2, ..., b_k\}, k < n$ take on the responsibility of finding the computational resources (data, disk access, CPU time, network time, etc.) required to extract data from the target data sources in $S$ to answer the queries posed by the clients in $C$. QSBs perform query related tasks such as *query parsing*, *query optimization* and *query execution*. Also, QSBs exhibit Peer-to-Peer behavior since a broker $b \in B$ might contact other brokers in $B$ to help it solve a given query. This is done to prevent a centralized operational model, in which a central broker needs to know all data sources, and becomes a focal point through which all queries must pass. This would make the system unreliable and inefficient as the central broker site becomes a single-point of failure and a performance bottleneck. The QSBs in $B$ can access the data sources in $S$ by means of a server application known as the *Information Gateway*.

A collection of Information Gateways (IGs) $G \subset H, G = \{g_1, g_2, ..., g_m\}, m < n$ have the role of providing access to the wealth of information contained in the data sources in $S$. It is at this level, of the IGs, that data extraction occurs. IGs can currently extract data from relational databases such as PostgreSQL and MySQL. In addition, IGs can execute query operators, particularly those that can filter out unwanted results, such as predicates. Clearly, metadata is needed for the brokers to be able to find the required data sources and their associated Information Gateways. These metadata must be spread throughout the system to advertise the availability of resources. The responsibility for these metadata dissemination is given to a type of server known as the *Registration Server* (RS).

A collection of Registration Servers (RSs), $R \subset H, R = \{r_1, r_2, ..., r_p\}, p < n$ deals with the problem of advertising metadata, encoded in XML, describing resources such as: query operators,

local tables, global tables, data types, CPU cycles, data sources, network bandwidth, disk space, and so on. Two or more RSs work as peers to exchange these metadata, just as network routers advertise routes to each other to enable future packet forwarding decisions. The last two elements in NetTraveler are known as the *Data Synchronization Server* and the *Data Processing Server*.

A collection of Data Synchronization Servers (DSSs) $D \subset H, D = \{d_1, d_2, ..., d_t\}, t < n$ groups server applications that help clients in caching query results, obtaining extra disk space, and keeping synchronized copies of data natively stored by a client which also happens to behave as a data source from time to time. More importantly, a DSS $d \in D$ can become a **proxy** for a client $c \in C$, gathering the results intended for client $c$ if the client goes offline or experiences some type of failure. Finally, a collection of Data Processing Severs (DPSs) $P \subset H, P = \{\rho_1, \rho_2, ..., \rho_v\}$ contains the server applications that provide a commodity service for computational tasks during query processing. These tasks include query execution, sorting, or any other type of specific computational operation required.

The elements in NetTraveler are logically organized into groups of cooperative applications known as *ad-hoc federations*. Federations are ad-hoc because they can be formed or dissolved over time, based on the decisions taken by its members. A federation can spawn more than one LAN, and a LAN can have elements that belong to more than one federation. The simplest federation is one made out of one *local group*, which consists of one QSB, one or more data sources and their associated IGs, one or more clients, one RS, one DSS, and one DPS. In some instances, having a Data Processing Server might be optional, particularly in cases where the applications only require simple queries to the data sources. DPSs will be most likely used in environments that require complex processing capabilities, or which have many low-powered devices.

Two local groups $L_1$ and $L_2$ can be combined to form a *cluster* by making the data broker from $L_1$, $b_{L_1}$, become a peer of the broker of $L_2$, $b_{L_2}$. In our framework, Peer relationship is bi-directional, hence $b_{L_2}$ becomes a peer for $b_{L_1}$. As a consequence of these events, the RS in each local group becomes the peer of its counterpart in the other local group, and they begin exchanging metadata about the resources available in each local group. A cluster of three local groups can

be made by adding a third local group $L_3$ and making its broker, $b_{L_3}$, become the peer of either $b_{L_1}$, $b_{L_2}$, or both. The same happens with the RSs in each one of these groups. Larger and more complex clusters can be constructed in this fashion and, as you can see, clusters represent complex federations with multiple brokers cooperating to share access to the data.

## 3.3 NetTraveler Query Services

The minimum set of components needed for query execution to take place in NetTraveler are: the *Query Service Broker* (QSB), the *Information Gateway* (IG), the *Data Synchronization Server* (DSS) and the *Data Processing Server* (DPS). The last two elements will participate in query processing only under special circumstances. The DSS allows a query submitted by a client that has suffered a disconnection to be completed albeit the client is unreachable. This process, known as *query rerouting*, is a subject that is further discussed in Section 4.3. The DPS is helpful in cases where extra computational resources are needed, however it is not absolutely necessary to carry out query execution.

The QSB, the IG and the DSS are the elements that must be implemented first in order to test the query execution framework of NetTraveler. The term *Query Service* is used to refer to any of these components since they participate in query execution. In the remainder of this thesis, there will be no further discussion about the functionalities of the RS and the DPS other than what has already being addressed. Additional information about both of these components can be found in [31] and in [32]. The following subsections describe the inner workings of the QSB, the IG and the DSS.

### 3.3.1 Query Services Broker (QSB)

Figure 3.2 shows a layered view of the internal components of the QSB and their organization. The *Client Service API* serves as the entry point for clients and other Query Services that need to communicate with the QSB. Requests received at the *Client Service API* are handed

to the *Execution Management* layer. This layer is in charge of coordinating the execution of every request by allocating the necessary resources and invoking the appropriate underlying components. For example, a query request received from a client has to be parsed first, by invoking the *Query Parser*, to obtain a new representation of the query known as the *parse tree*. Then, this *parse tree* is used by the *Query Optimizer*, who uses metadata information stored in a catalog, to build another representation of the query that is used by the *Execution Engine*. This representation of the query is known as the *query plan*. The *query plan* is used by the *Execution Engine* to solve the given query. All of these steps are not necessary for a query request received from a peer QSB. In this latter case, the query request will already include the operations (e.g. a *query sub-plan*) that must be carried out by the *Execution Engine*.

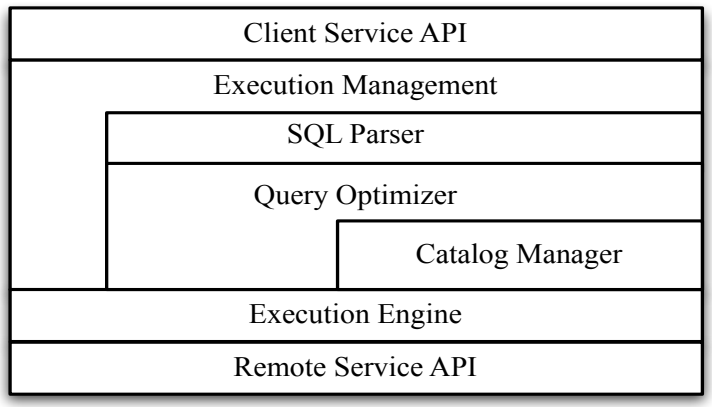



Figure 3.2: QSB internal organization

Figure 3.3 depicts the interaction between a client and a QSB and also between a QSB and one peer. Client $C$ submits the query $Q$ to *QSB1*. Meanwhile, *QSB2* receives $Q_1$ from *QSB1*. $Q_1$ is the part of the query $Q$ that *QSB2* must execute. *QSB1* in the figure is known as the coordinator of the query since it received the query, generated the plan to execute it, and will return the result tuples to the client.

As its name suggest, the *SQL Parser* in the QSB processes queries expressed in SQL syntax. Native support for other type of query languages (e.g. XML Query Language) is also intended, but has been left as future work. The *Query Optimizer* is perhaps the most complicated

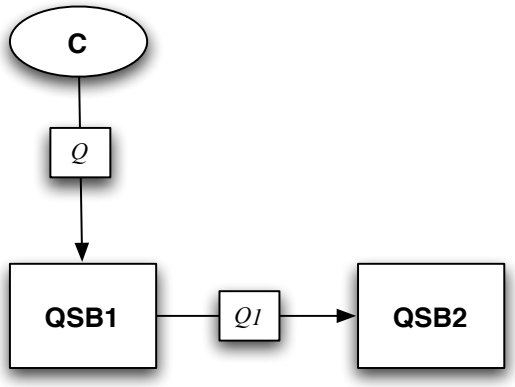



Figure 3.3: QSB interacting with a client and a peer

component in the QSB. In fact, the development of a query optimizer for a system like NetTraveler is a matter that demands a research on its own, and has been left out of the scope of this thesis. However, the generation of at least "naive" query plans was needed to test the execution of queries inside the NetTraveler framework. Therefore, the *QSB* currently uses a simplified version of the *Query Optimizer*. As we shall see in detail in Section 3.4.2, the current query optimizer generates *left-deep* query plans in which each node, or operation, in the plan has an annotation of where it must be executed (QSB or IG).

The optimizer uses metadata from a catalog to build the query plans. The interface to access this catalog is provided by the *Catalog Manager*. The current catalog implementation stores information of a NetTraveler federation that is fixed, and whose topology will not change until the federation is dissolved. Also, the catalog is replicated at each QSB in the federation. These simplifications to the catalog were needed to test the execution framework of NetTraveler without dealing with the issues of metadata dissemination and catalog updates in distributed systems, which are problems that are out of the scope of this thesis. Section 3.4.2.1 further discusses the implementation of the catalog.

The *query plan* that is generated by the optimizer is a tree where each node describes a relational operator that must be carried out to solve the query. Each node in the tree has annotations that include the site where the operator must be executed, the maximum number of tuples to return per iteration and, in the case of joins, the algorithm to be used for an operation.

Operators that filter tuples, such as predicates, are pushed to the *IG*s and peer *QSB*s. Other operators are evaluated in the *QSB* that received the query request. This approach is very similar to the one employed in [3].

To execute a query, the QSB must load the classes that execute each of the operations described by the nodes of the *query plan*. The *Execution Engine* is responsible of loading the appropriate classes. Each relational operator is implemented by a class that has an iterative functionality. The set of classes that implement the relational operators are known as the *iterators*. There are iterators for performing local selections, projections and joins as well as for fetching tuples from remote sources. The *Remote Service API* provides the interface that is used by the QSB to communicate with peer QSBs and IGs.

Each query submitted to the *QSB* is run in its own thread of execution. Also, the results of a query are produced independently of whether there is someone consuming them or not. Of course, only a limited amount of results are produced if no one consumes them in a while. The results produced are cached, and the thread of execution is put in a dormant state until a client or a peer *QSB* request them. What is important to notice here is that the execution of the query has not been stopped, the query is still being executed but it will be in a suspended state because no one is consuming the results produced.

### 3.3.2 Information Gateway (IG)

The IG runs in the same site (e.g. in the same machine) where the data source that it will be accessing is running. The IG is similar to a wrapper in a mediator system [5] in the sense that it provides the QSB with an uniform representation of a remote data source. The organization of the IG is depicted in Figure 3.4. In general, the *Client Service API*, the *Execution Management*, and the *Execution Engine* layers play the same role in the IG as they do in the QSB. The *Client Service API*, however, accepts request from QSBs only. All the requests received are also actually processed by the *Execution Management* layer.

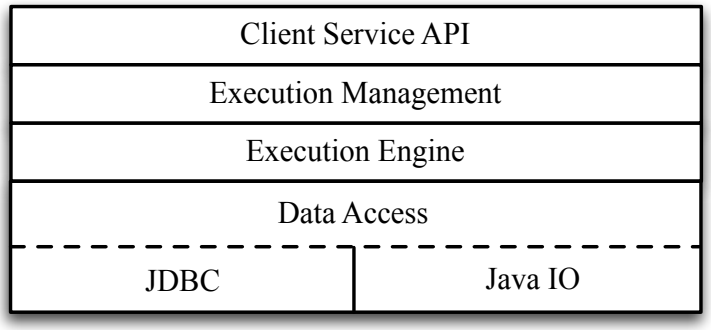



Figure 3.4: IG internal organization

The query requests that the IG receives from the QSB specify the part of the query that the IG must execute. The IG contains an iterator-based *Execution Engine* that it uses to execute the relational operators defined in the query sub-plan received from the QSB. Typically, these operations will include selections and projections that help in reducing the amount of data to be transfered over the network. The *Execution Engine* of the IG follows the same design patterns as the QSB's execution engine. Also, each request received by the IG is executed in its own execution thread. Figure 3.5 shows the interaction between the IG and the QSB. The elements in Figure 3.3 have been added to better understand how query execution takes place in a NetTraveler federation. It can be seen from the figure how the IGs receive from the QSBs the part of the query that they must execute.

The *Data Access* layer provides the interface to access different types of data sources such as text files, XML files or relational data. This layer of functionality is shared by both the Information Gateway and the Data Synchronization Server (DSS). The IG is currently limited to data sources that store data using a Relational Database Management System (RDBMS). The access mechanism for RDBMSs was build on top of JDBC [33]. The *Data Access* layer can also access data that is stored in OS files using the native Java I/O libraries, but this is not currently used in the IG. Whenever possible, the IG will leverage on the capabilities of the underlying data source when executing part of the query. The idea behind this scheme is that data sources with query processing capabilities can be exploited and not just used as mere data containers. The current implementation of the IG takes advantage of RDBMS data sources by converting part of

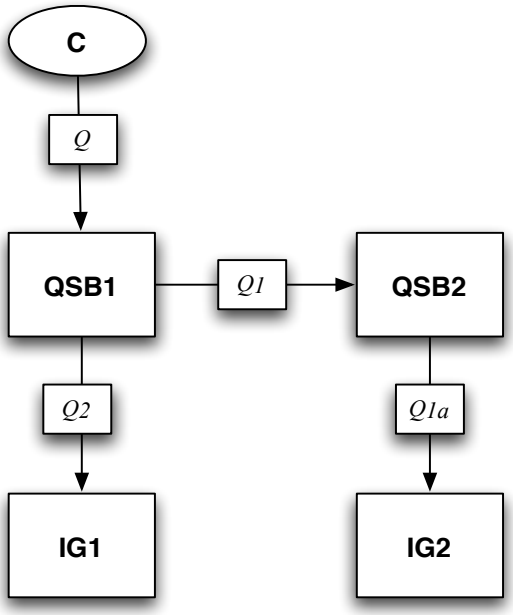



Figure 3.5: IGs interaction in a NetTraveler federation

the query sub-plan into an SQL query that is executed directly by the RDBMS.

### 3.3.3 Data Synchronization Server (DSS)

NetTraveler has been envisioned as a system that will efficiently harvest data sources and that will efficiently support query processing in Wide Area Networks were a considerable amount of clients and servers are running on mobile devices. In this sense, NetTraveler expects to cope with the problems of databases that go offline in the middle of query execution and also with the issue of continuing with the execution of queries when disconnections occur at the participating parties. The DSS was designed with these goals in mind. This thesis focuses on how to provide efficient query processing in the presence of mobile clients that experiment frequent network disconnections. This thesis does not deal with the issue of servers that go offline in the middle of query execution. Nonetheless, the implementation of the DSS provides the infrastructure upon which recovery of the server-side of a query can be built.

The DSS is the Query Service that allows NetTraveler to recover a client's query work in the case of a disconnection failure. In current middleware solutions, if a failure that produces a

disconnection at the client occurs while a query is being executed, the work performed before the failure is lost. Upon recovering from the contingency, the client has to resubmit the query and the system has to execute the whole query again. Clearly, this is a waste of system resources in terms of CPU cycles, disk access time and network access time. Furthermore, for clients accessing the system via pricey Internet links or with limited battery resources this scheme is just unacceptable.

With the aid of the DSS, NetTraveler is capable of recovering the query work of a client from the moment this client suffers a disconnection; continuing with query execution albeit the client being absent. When a client submits a query for execution to a QSB it also submits information of a DSS that is willing to work on the client's behalf. The client also submits a time value $t_i$ that is known as the *idle threshold time*. This time is defined as the maximum amount of time that a client is expected to be idle. Thus, this is the maximum amount of time during which the client will not be requesting any result tuples for a query that it submitted. A QSB will monitor the queries that it is coordinating (queries that it received from clients) to see if it finds a query that goes idle beyond $t_i$. If it finds one, the QSB will communicate with the DSS that is willing to work on behalf of the client and will send to it the information of the query. The DSS will then proceed to work on behalf of the client by gathering the results intended for it. Figure 3.6 depicts how the client $C$ suffers a disconnection and how the *DSS* gathers the results intended for the client. Figure 3.7 shows how the client $C$ gathers the results from the *DSS* after recovering from the disconnection. We shall see in detail how this recovery process is implemented in Section 4.3.

The current organization of the internal components of the DSS is depicted in Figure 3.8. Again, we have the common *Client Service API*, *Execution Management* and and *Execution Engine* layers. Through the *Client Service API*, the DSS can receive request from both, clients and QSBs. QSBs submit requests to the DSS to ask the DSS to work on behalf of a client. Clients communicate with the DSS to gather the results of a query that the DSS has executed on their behalf. The *Remote Service API* is used by the DSS to communicate with the QSB.

The *DSS* has its own *Execution Engine*. The decision of implementing an execution engine in the DSS arose from the fact that the DSS can also work on behalf of an IG. However, the

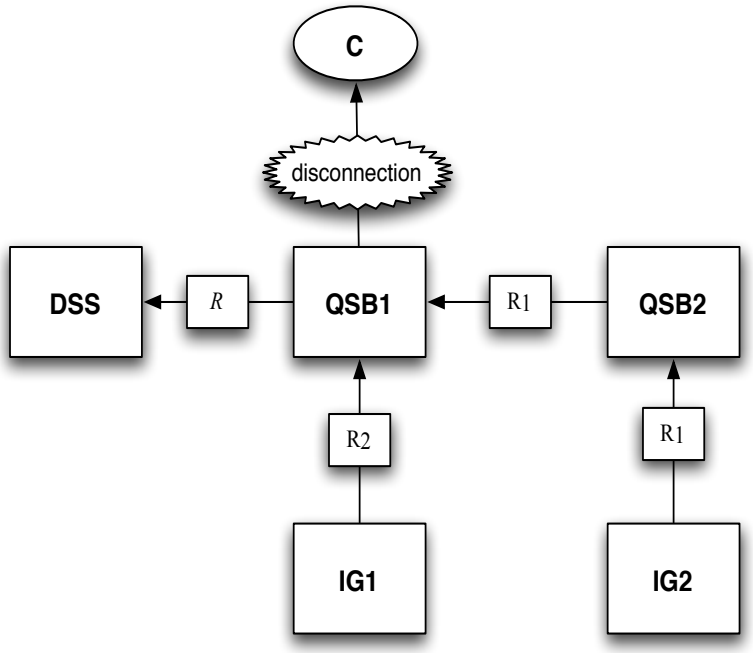



Figure 3.6: DSS gathering results on behalf of a client

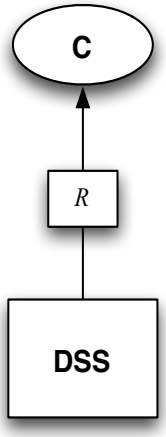



Figure 3.7: Client gathering results from a DSS

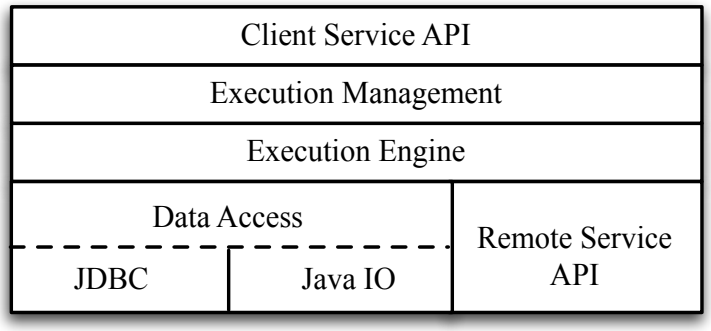



Figure 3.8: DSS internal organization

implementation of this functionality of the DSS was not needed to achieve the goals of the work proposed in this thesis and was therefore left for future work. The *Data Access* layer is implemented in exactly the same way as it was implemented in the IG. The *Data Access* layer is used by the DSS to materialize the results of a query to disk files. Currently, the DSS stores materialized relations in OS files using the Java I/O libraries. The DSS can also access Relational Database Management Systems (RDBMSs) using JDBC.

## 3.4 Query Processing Framework

Figure 3.9 shows a simplified representation of the architecture of a Database Management System (DBMS) query processor. This architecture is typically used in both centralized and distributed database systems. The following subsections briefly describe each of these components, emphasizing on the *Execution Engine*.

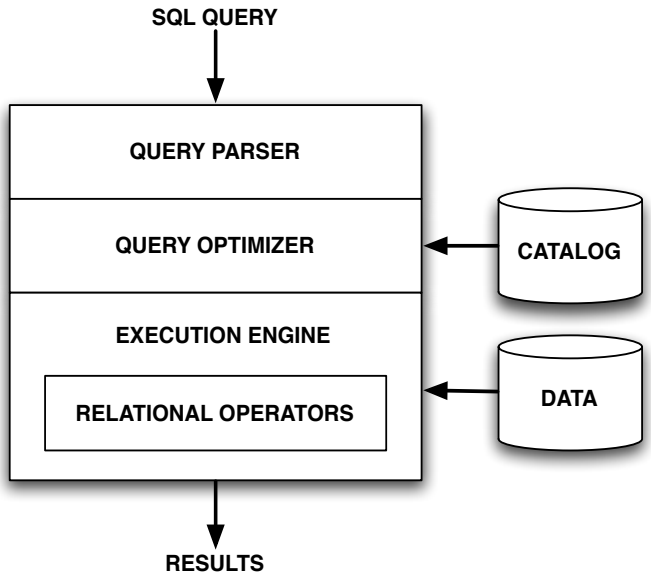



Figure 3.9: Simplified Query Processor Architecture

### 3.4.1 Query Parser

The query parser translates a submitted SQL query into an internal representation that can be processed by the later phases. This internal representation is commonly known as the *query graph*. Parser construction is a well understood matter [34] and many existing tools (e.g. Bison, JavaCC, JLex, Javacup) can be used to construct them. In NetTraveler the parser resides inside the *Query Service Broker* (*QSB*).

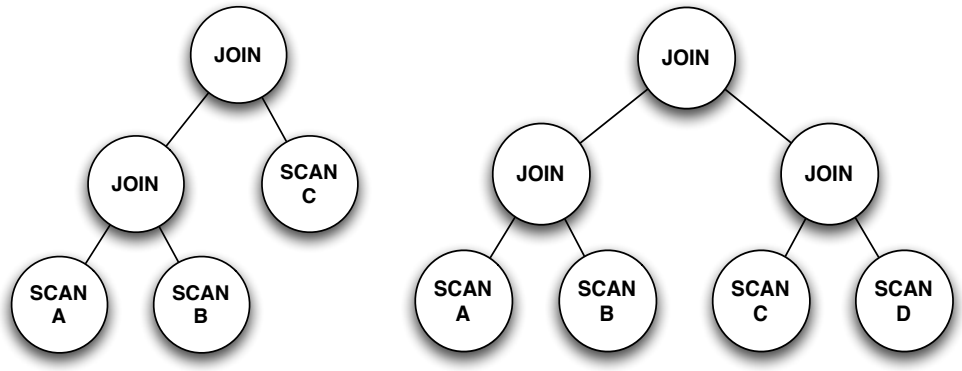
### 3.4.2 Query Optimizer

The query optimizer is responsible of finding the best strategy to solve each query that is posed to the system. The optimizer must decide which access method (e.g. *scan*, *index scan*) will be used for each relation involved in the query and the order in which joins, selections, aggregates and projections must be executed. Optimizers will typically use a cost estimation model that optimizes for a specific metric (e.g. response time, system throughput). Using statistics stored in the catalog, the optimizer will build the best plan specific to that cost model. The output of the optimizer is a *query plan*, represented as a tree, that tells exactly how the query will be executed. Nodes in the plan represent operators that carry out relational operations (e.g. *joins*, *scans*, *projections*) and each node is annotated with information regarding particular algorithms for operations (e.g. *nested-loops join*, *hash join*, etc). If the system is distributed, nodes will also be annotated with the site where each operator is to be executed.

Figure 3.10 shows two types of query plans represented as trees. As can be seen, a *left-deep* plan is one in which the *inner relation* of a join operator is always a base relation.[1] *Bushy* plans are more general, there are no restrictions in terms of the input relations for joins. Hence, in *bushy* plans joins could be performed between two base relations or between the results of two previous joins. During the development of the optimizer for the System R [13] project it was found that enumerating only left-deep plans greatly reduces the search space of possible plans for a query,

[1] The left side of the join operation is commonly known as the *outer relation* while the right side as the *inner relation*.

which in turn simplifies the implementation of the optimizer. Following this lead, most of the commercially available DBMSs have optimizers that enumerates only left-deep plans.




(a) Left-deep plan (b) Bushy plan

Figure 3.10: Tree representation of query plans

The development of an optimizer is not an easy task. In fact, many dissertations have been written on that subject alone. The focus of the work during the course of this thesis was the development of NetTraveler's query execution engine (Section 3.4.3). However, in order to test the execution engine, at least a simple optimizer capable of generating sub-optimal query plans was needed.

#### 3.4.2.1 Simplified System Catalog

The catalog of a typical DBMS contains information about every table and index that the system contains. This information includes, but is not limited to, the names of the relations, the names of the columns in each relation and the type of file implementation used for each relation. Besides this information, the catalog also stores statistical data, such as the cardinality (number of tuples) of each relation. All this information is used by the query optimizer to generate the plans to solve each query submitted to the system.

A distributed system must deal with the issue of where shall the catalog be stored. If the catalog is stored in a single site, it is easier to disseminate metadata and to perform lookups

and catalog updates. However, this scheme is susceptible to failure of the site that holds the catalog. Another approach is to distribute the catalog among the participating sites of the system. However, this makes the task of keeping the catalog up-to-date more difficult, since multiple updates operations are necessary. Finding the best way to deal with these issues is out of the scope of this thesis and further research is needed to establish the final organization of the catalog in NetTraveler.

The catalog used for the purpose of completing the work proposed in this thesis is a considerable simplification of a catalog for a distributed system. The current catalog stores information concerning a specific federation, and it is assumed that the federation does not change until it is dissolved. Hence, a catalog is generated for each federation that is formed. The catalog is replicated at all QBSs in the federation and it is stored by each QSB in tables that are part of a database. A relational DBMS engine is used for this purpose. The current catalog stores the following information:

- The *ip address*, *port*, *service type* (e.g. *IG* or *QSB*) and *implementation type* of each *IG* and *QSB* in the federation. *Implementation type* tells how an *IG* or *QSB* service is implemented. [2]
- The *peers* for each *QSB*.
- The *IGs* that each *QSB* has access to.
- The name of each relation (table) in the federation.
- The name and type of each attribute of every relation in the federation.
- The relations that are stored at each *IG*.

#### 3.4.2.2 Simplified Query Optimizer

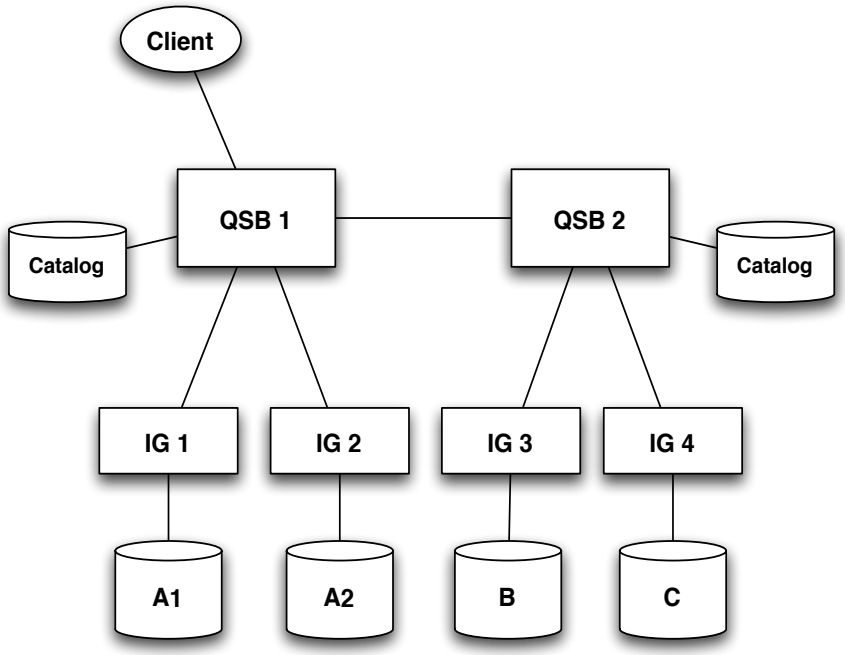
The current NetTraveler query optimizer is provisional, and uses a simplification of the dynamic programming algorithm for query optimization. The algorithm is multi-pass and builds only left-deep plans with projections and selections applied as early as possible. All optimizations are done inside the *Query Service Broker QSB*. The following example shows how the algorithm

[2] All services are currently implemented as Java Web Services using the Apache Axis Toolkit [35].

works. Suppose that the query in Figure 3.11(a) is posed to the NetTraveler federation in Figure 3.11(b). This federation is formed by two *QSBs* and four *IGs* (the rest of the components have been removed for simplicity) and has three relations: $A$, $B$ and $C$. Relation $A$ is partitioned in two ($A_1$ and $A_2$) and each partition is stored at a different *IG*.

```
SELECT A.a, B.a, C.a
FROM A, B, C
WHERE A.b = B.b
AND B.c = C.b
AND A.c > 15
```

(a) Example query




(b) Example federation

Figure 3.11: Federation and query for optimization example

The algorithm will proceed as follows:

- **Pass 1:** The algorithm starts with the first relation (from left to right) in the **FROM** clause ($A$ in this example). First, those conditions in the **WHERE** clause with attributes that apply only to $A$ are identified. These conditions are the *selection* operations that will be applied to $A$ before any join. Then, all attributes of $A$ that are mentioned in the **SELECT** clause

or in conditions of the **WHERE** clause involving attributes of other relations are identified. These attributes are *projected* before any join takes place. A *scan* operation is selected as the access method for $A$. All these operations must be scheduled to occur at the $IG$ near the data source. Therefore, the route to the destination $IG$ must be found (see Section 3.6 for details of this procedure). After the route is identified, a special *fetch* operation is introduced for every network hop needed. Figure 3.12 shows the query plan with its corresponding annotations and *fetch* operations. The *fetch* operation is added after the last operator applied to $A$, which in this case is the *projection* operator. If more than one *fetch* operation is needed it is added over the previous *fetch* operation. If the relation is partitioned, the same steps are repeated for each partition and the relation is reconstructed with an *union* plan near the site where the *join* will take place.

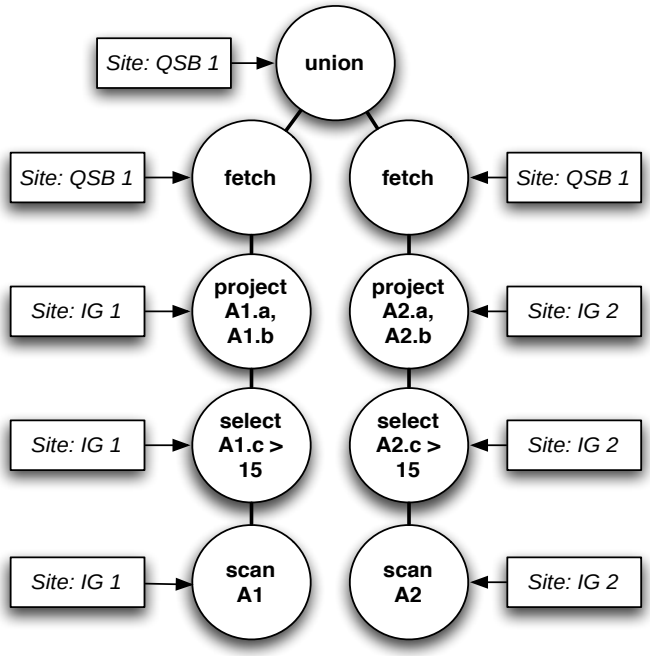



Figure 3.12: Plan after first pass of the optimization algorithm

- **Pass 2:** A two-relation plan (join) is build by considering the single-relation plan obtained after **Pass 1** as the outer relation and the next relation in the **WHERE** clause ($B$ in our example) as the inner relation. First, an access plan for $B$ is found exactly as was done for $A$ in **Pass 1**. Then, all *selection* conditions in the **WHERE** clause involving only $A$ and $B$ are

identified in order to build the join between them. The join method used is a *page-at-a-time* implementation of the *nested loops join*. This join implementation materializes the inner relation to a file after it is read for the first time. Subsequent reads of the inner relation are done by scanning the file. Figure 3.13 shows the query plan after the second pass.

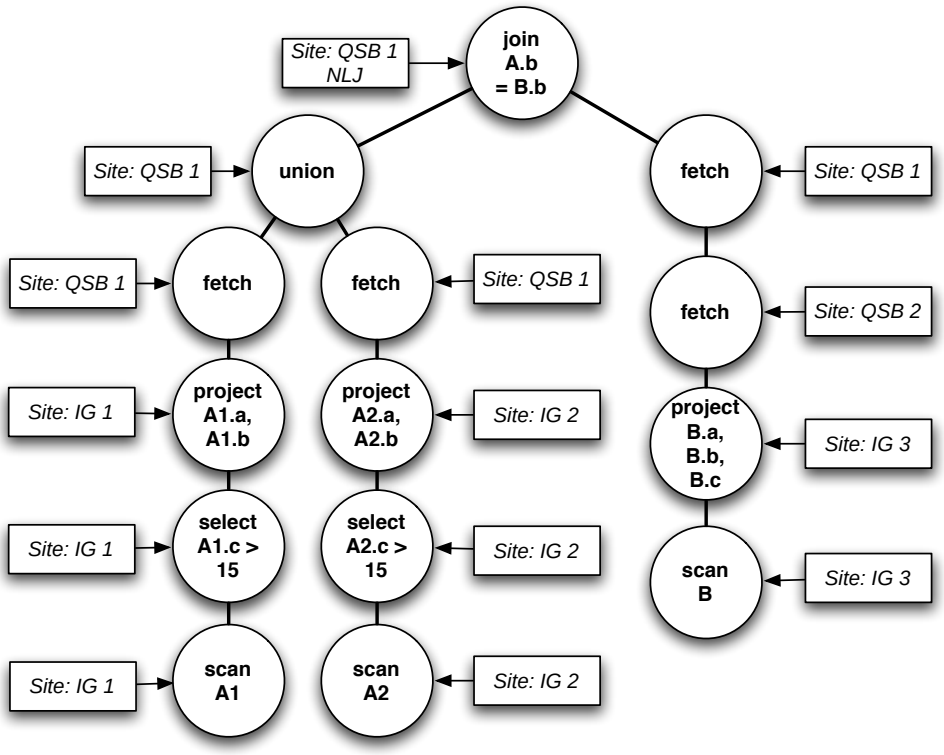



Figure 3.13: Plan after second pass of the optimization algorithm

- **Pass 3:** A three-relation plan is now generated using the plan obtained after **Pass 2** as the outer relation and the next relation ($C$ in our example) as the inner relation. The plan is generated proceeding exactly as in **Pass 2** of the algorithm. If all relations in the query have been used, an additional *projection* operation is added at the end if necessary. Figure 3.14 shows the query plan after the third pass.
- **Additional Passes:** The process is repeated with additional passes until a plan that includes all the relations found in the query is produced. The output of the algorithm is a query plan that is ready for execution.

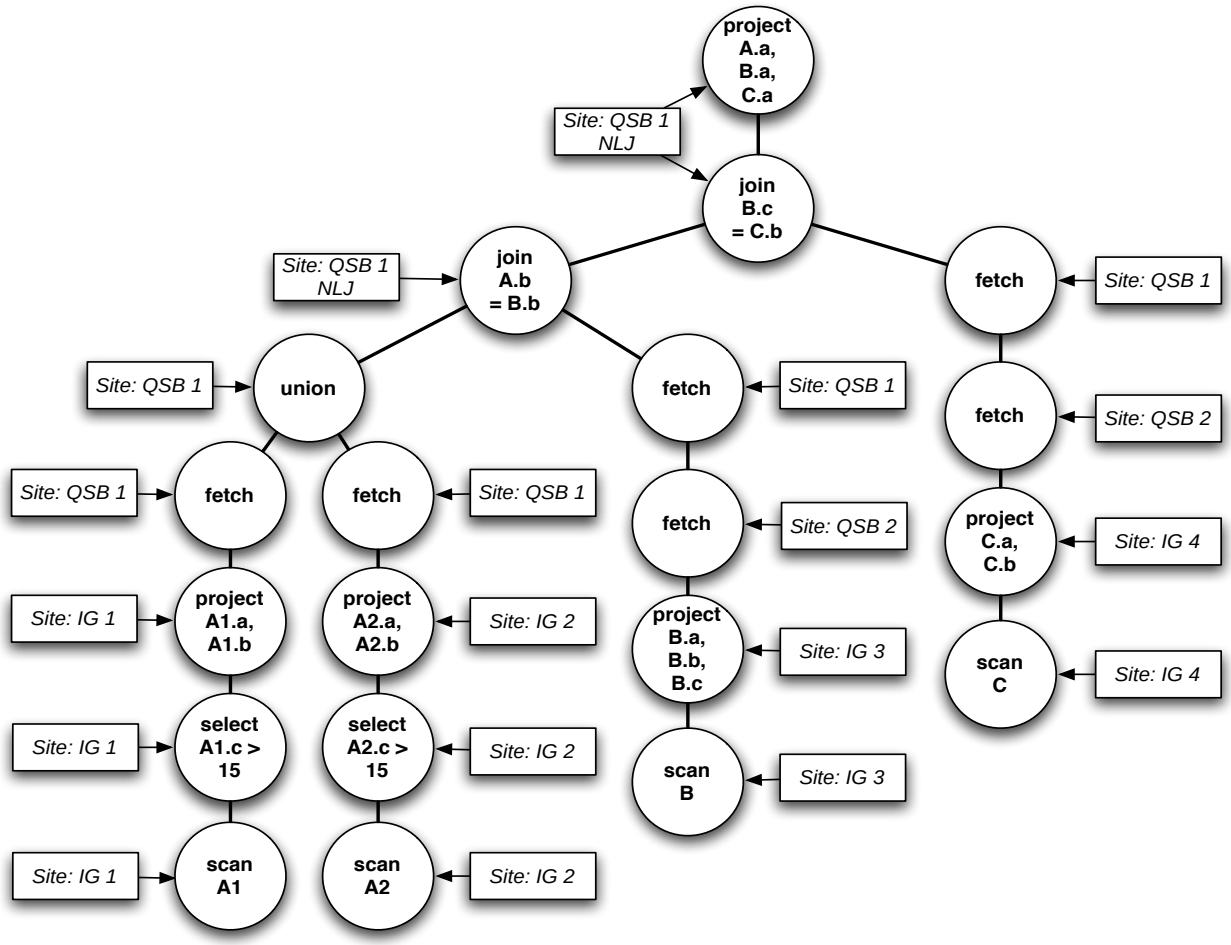


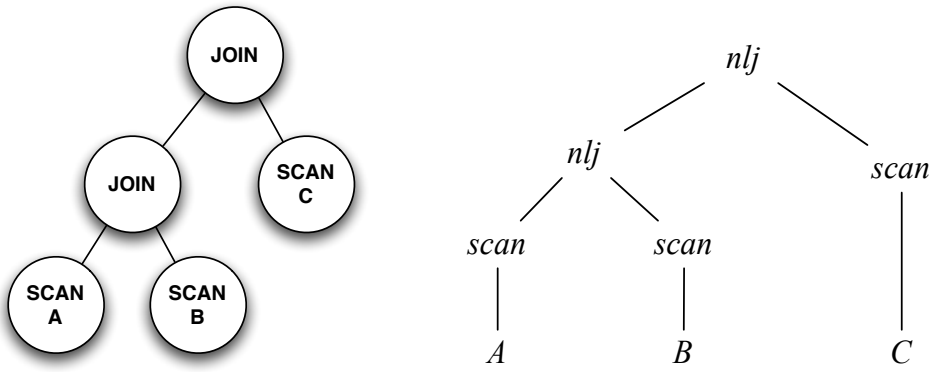

Figure 3.14: Plan ready for execution

### 3.4.3 Query Execution Engine

The plan that is generated by the optimizer is commonly called the *physical* query plan because it defines precisely how the query is to be executed. However, to actually execute the plan, it must be converted into code that implements the relational operators in the plan and that has the mechanisms for coordination and cooperation among multiple such operators. The *query execution engine* is the implementation of the relational operators needed to execute queries in such a way that they can be scheduled to run inside the DBMS. The vast majority of database middleware systems developed so far have query execution engines that closely resemble the execution engine of a single-site DBMS. This type of query execution engine is based on the *iterator* model [36], where each relational query operator is implemented with an iterative functionality.

The operation of the iterators is separated into three steps: (a) preparation of the iterator to produce results, (b) production of items, and (c) release of resources used. These functions have been named *open()*, *next()* and *close()*, respectively, since they emulate the operations performed by an operating system to scan a file. Because all iterators share these functions, they are implemented following a common programming interface that defines these operations. This allows for pipelining of any two iterators together to form complex expressions, which in turn allows for combinations of multiple iterators to execute any type of query plan. Figure 3.15(b) shows how the plan in Figure 3.10(a) (conveniently repeated in Figure 3.15(a)) is converted into iterators that are plugged into one another to execute the plan. As can be seen in Figure 3.15(b), the results produced by an iterator can be *pipelined* into another iterator without the need of writing temporary results in a file in disk. This improves execution performance considerably.

The iterator model is the state-of-the-art implementation for query execution engines in most of the prototyped and commercially available DBMSs. Hence, NetTraveler's execution engine implementation follows this model. Of course, for distributed environments where communication between remote computers occurs, there is a need for special iterators to handle the passing of tuples through the network. In NetTraveler, an iterator named the *fetch* iterator is used to get tuples




(a) Tree representation of a left-deep plan

(b) Iterator representation of a left deep plan

Figure 3.15: Plan represented as a tree and its corresponding iterator representation

from remote data sources. Figure 3.16 shows a distributed query plan that has been converted into iterators and that includes *fetch* iterators that get tuples from remote sources.

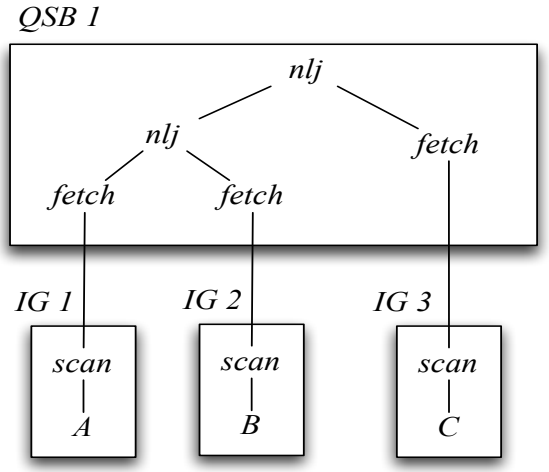



Figure 3.16: Iterator representation of a distributed query plan

## 3.5 Overview of Query Execution

We shall now see how query execution has been carried out traditionally in Middleware Database Systems and how NetTraveler proposes a new model for query execution. The model of query execution that has been used in middleware systems is the *connection oriented* model. In

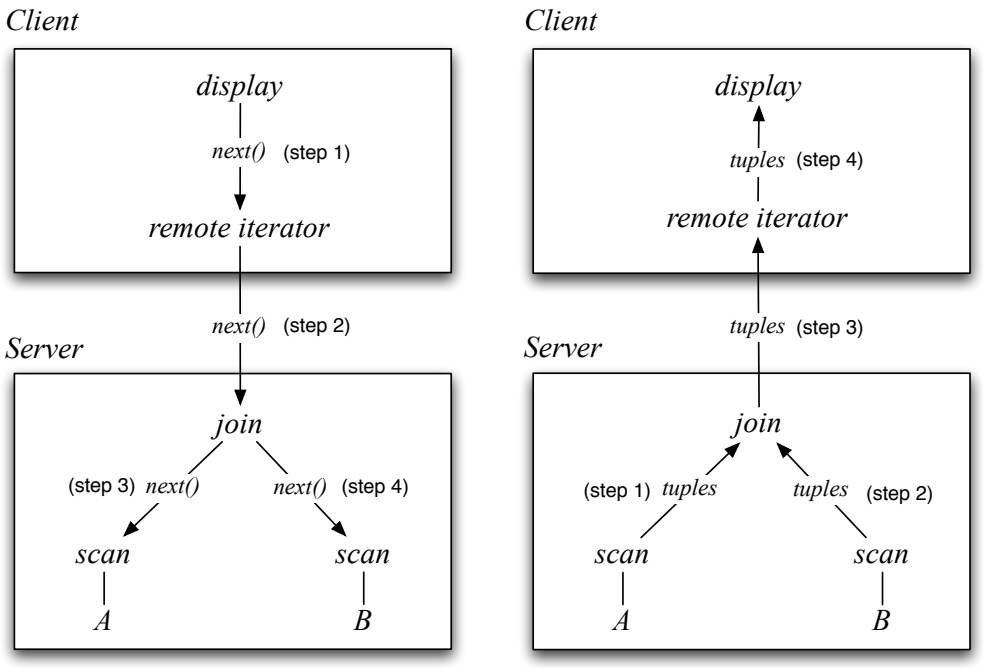
NetTraveler, a *connectionless oriented* query execution model is used.

### 3.5.1 Connection Oriented Query Execution

Query execution engines in Middleware Database Systems implemented so far have followed a connection oriented approach. To execute a query in these type of systems, a connection must be first established between the client and the server using a connection oriented protocol (e.g. TCP). Then, typically using an API that provides an iterator-like interface to communicate with the server, the client will perform the following steps:

1. Submit a query for execution by invoking an operation in the API that is analogous to the *open()* operation of an iterator.
2. Request the results by repeatedly calling *next()* until all tuples that satisfy the query are obtained.
3. Submit another query to the server, repeating steps 1 and 2, or release the connection by calling the *close()* operation.

Query execution takes place in a request-driven fashion. The server will not produce results until the client explicitly ask for them by issuing a *next()* request. Figure 3.17 shows an example of query execution in a connection oriented system. In this example, the client API provides iterators for submitting queries to the server and for displaying the resulting tuples. A *next()* request issued by a client is propagated to the server, which in turn propagates the *next()* call to the top-most iterator in the *query plan*. This iterator further propagates the call until the deepest iterator in the plan is reached. The propagation of the *next()* request can be seen in Figure 3.17(a). The numbers in the figure denote the order in which each event occurs and the arrows denote the direction of the propagation of the call. Results are generated in the deepest iterator and then forwarded to the upper iterators. Iterators continue processing the results and forwarding them up until the results reach the client. Figure 3.17(b) shows how the results are produced and returned to the client. The numbers also denote the order in which the events occur and the arrows indicate the direction of the results, which are exactly the opposite as how the *next()* request propagates.




(a) Propagation of *next* request (b) Returning of results after *next* request

Figure 3.17: Connection oriented query execution

One drawback of such connection oriented implementation is that the execution of a query will be aborted if any unexpected failure (e.g lost of power) or condition that causes the interruption of network connectivity occurs at the client. In the presence of any failure, the query will be dropped by the system and it will have to be submitted again. This leads to excessive waste of system resources such as CPU cycles, disk time, network bandwidth, etc. Also, the request-driven functionality of this type of query engine does not scale well to Wide Area Environments. If a plan must access several data sources that are located at different sites, each *next()* request that is issued will cause calls to many computers in the network, increasing the query execution time considerably.

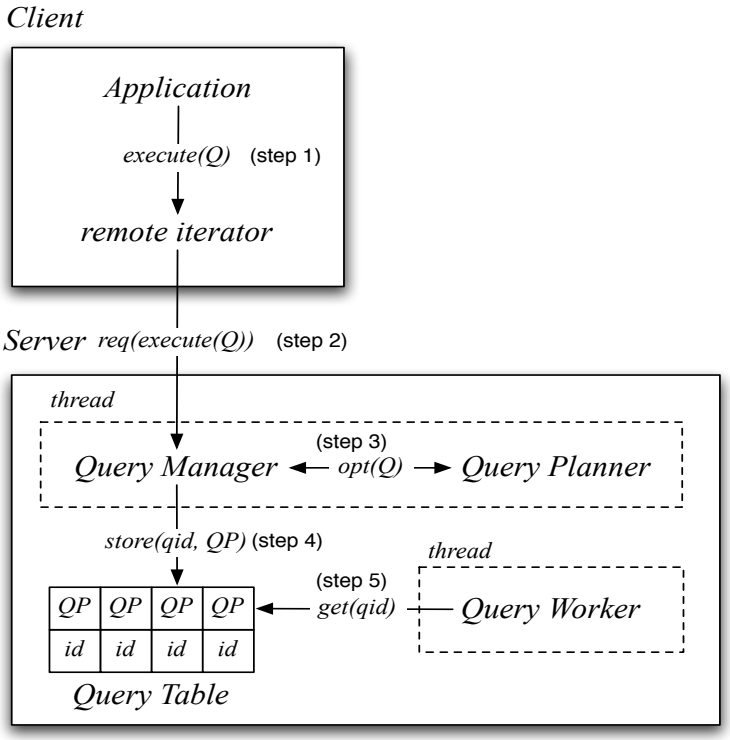
### 3.5.2 Connectionless Query Execution

From the discussion of the previous section we can see that a connection oriented query execution engine is not the best option for an environment were the majority of clients will be mobile devices. Hence, NetTraveler's query execution engine was implemented following a *connectionless*

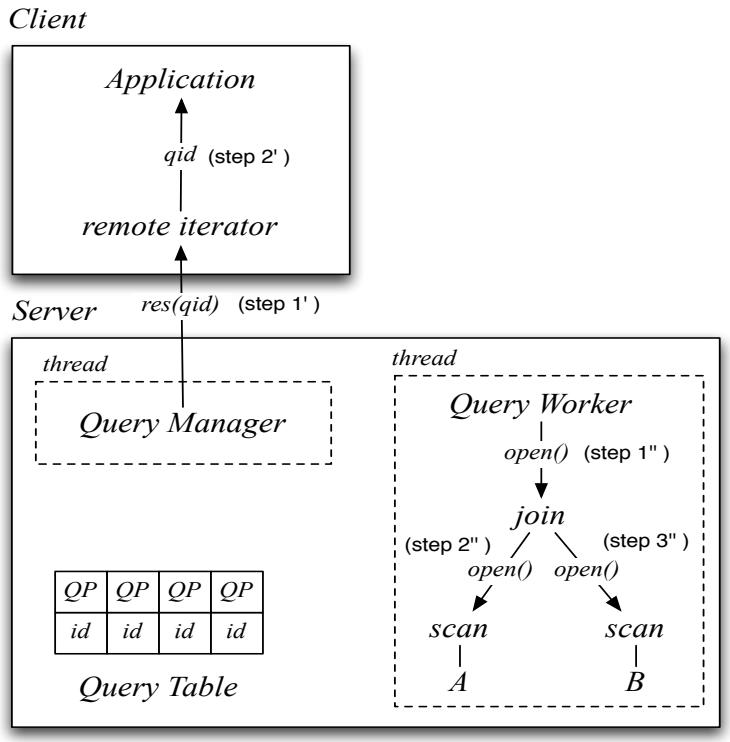
oriented approach. In a connectionless query execution engine there is no need to maintain an open connection during the entire execution of a query. Rather, a connectionless communication protocol based on requests and responses, like HTTP, is used. Clients submit queries for execution using an API that provides a set of methods that are equivalent to those provided by a connection oriented system. Each method that is invoked causes a request/response pair of messages to be transmitted between the client and the server. Hence, an execute request by a client is followed by an execute response from a server. The query that the client submits to the server and the tuples that the server returns to the client are all contained in a pair of request and response messages. We shall now see how a query is executed in a connectionless query execution engine.

Figure 3.18(a) shows the chain of events that occur when a client submits a query to a server in a connectionless oriented system. *Step 1* in the figure shows how the *Application* running in the client submits a query for execution using a special *iterator* provided by the client API. The query is sent to the server inside a request message, as denoted by *step 2* in the figure. As can be seen, the server receives the request and starts processing it in a new thread of execution. This thread, the one that processes the request, have been named the *Query Manager* thread. The *Query Manager* thread will first parse and optimize the query to obtain the *query plan* that defines the operations that are needed to be carried out in order to solve the query. This is shown in *step 3* of the figure. Once the plan is built, the server will store information regarding the state of the query inside a data structure, as denoted by *step 4*. This data structure have been named the *Query Table* for this example.

Execution of the query in the server will take place in a new thread of execution. This new thread has been named the *Query Worker* thread for this example. *Step 5* of Figure 3.18(a) denotes how the *Query Worker* thread gets the query plan to be executed from the *Query Table*. The *Query Table* data structure serves as a shared object were both the *Query Manager* and the *Query Worker* threads will synchronize during the execution of the query. The *Query Worker* will be writing result tuples to the *Query Table* and the *Query Manager* will be reading tuples from the *Query Table*. Once the query is ready to be executed, the server will return an acknowledgment




(a) Client submitting a query for execution




(b) Server sending execute response

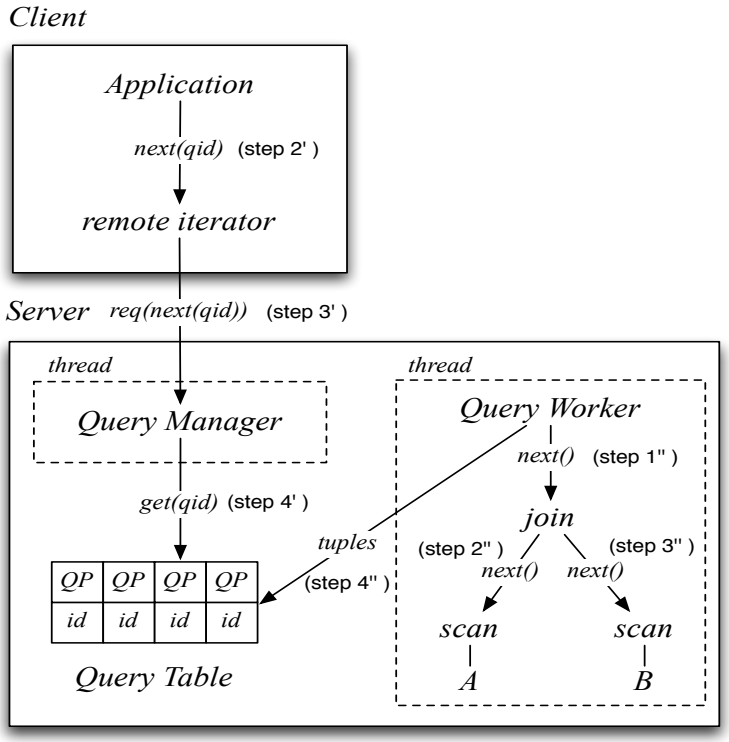
Figure 3.18: Query submitted to a connectionless system

response message to the client.

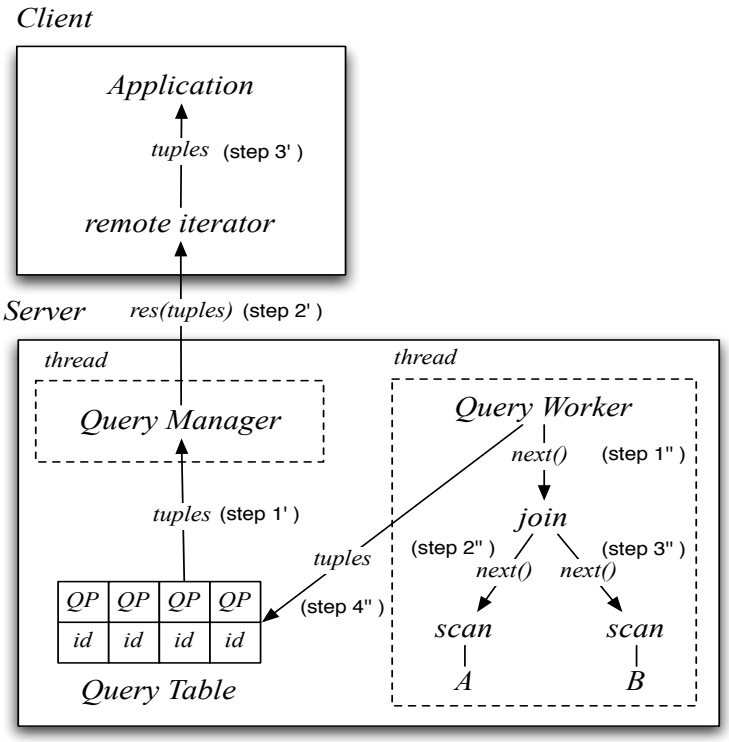
Figure 3.18(b) shows the events that occur when the server returns the acknowledgment response message for the execute request sent by the client. As denoted by *step 1′* in figure, the server will include a unique query id *qid* as part of the information contained in this response message. The id *qid* is used to identify the query inside the server for the purpose of attending future requests regarding the same query. Any information that the server finds relevant for the client might be included in the response message as well. The *Query Worker* thread will begin to execute the query at the same time that the response message is returned to the client by the *Query Manager* thread. *Step 1′* and *step 1″* of Figure 3.18(b) occur in parallel, as well as *step 2′* and *step 2″*.

The client can start gathering the results for the query once it has received the id *qid* of the query from the server. Figure 3.19(a) shows the events that occur when the client issues a *next()* request to the server. When asking for results, the client has to submit the query id *qid* to the server inside each request message, as denoted by *step 3′*. Notice that the server will perform operations in parallel and will not wait for the client to issue a *next()* request to start generating results, as in the case with a connection oriented query execution engine. Instead, by the time the client issues a *next()* request, the *Query Worker* in the server will be already executing the query by issuing *next()* calls to the iterators that implement the plan.

The *Query Worker* will store the results that are generated in the *Query Table*. The *Query Manager* thread will gather the tuples intended for the client from the *Query Table* data structure. Events that occur in parallel in Figure 3.19(a) are *step 2′* and *step 2″*, *step 3′* and *step 3″*, and *step 4′* and *step 4″*. Figure 3.19(b) shows the steps performed by the server when returning result tuples to the client. As can be seen from the figure, the *Query Worker* thread will continue to produce results for the query at the same time that the *Query Manager* thread returns the response of the *next()* request to the client. Execution will continue in this fashion until all results are gathered by the client. Once the query has been completely executed the server will discard all objects used to hold information of the query from the *Query Table* data structure.




(a) Client asking for results




(b) Server sending results to client

Figure 3.19: Query execution in a connectionless system

As we can see, for mobile clients in a Wide Area Environment, a connectionless oriented query execution engine provides several benefits over a connection oriented engine. For one part, the execution of queries that must access several data sources is greatly improved because the execution of query operators in parallel is maximized. Also, servers do not have to wait for a client (or a peer) to issue a *next()* request to produce result tuples. Rather, each server will execute the part of the query that is assigned to it independently (whenever possible). Obviously, a server cannot execute its part of the query in complete independence if the operations that it must perform depend on tuples generated by other servers. Nonetheless, this query execution scheme can help in improving the execution of queries in Wide Area Networks. For example, in environments were the network latency is considerable or where there are slow servers, this scheme is beneficial since the amount of time that must be waited for a server to produce results at the moment of issuing a *next()* request is minimized. Also, servers return result tuples from cache, which helps in improving query execution.

Another feature of a connectionless query execution is that a disconnection failure in the client or in one of the servers will not necessarily imply the abortion of a query. Parties participating in the execution of a query could reconnect after a failure and use the query id *qid* as a mechanism to attempt to re-establish the computation of a query from the moment of a failure. Having the possibility of servers that suffer disconnections in the middle of query execution lifts up the issue of *confidence in the solution*, or knowing the level of accuracy for the solution of a query. Let us suppose that each server is replicated in the system and that the system has a recovery mechanism were data from the replicas is used when a main server goes offline in the middle of query execution. If the replicas are not synchronized periodically, there is a chance that a client might end up receiving an incomplete or unasserted solution for a query. There must be a way to make the client aware that the solution might be incomplete, or even incorrect.

In the case of a *join* operation (or any other operation that requires tuples from two or more relations) the problem is easily dealt with if a server is missing and is not replicated. In this case, the operation will not get completed because one of the relations is missing, and the client

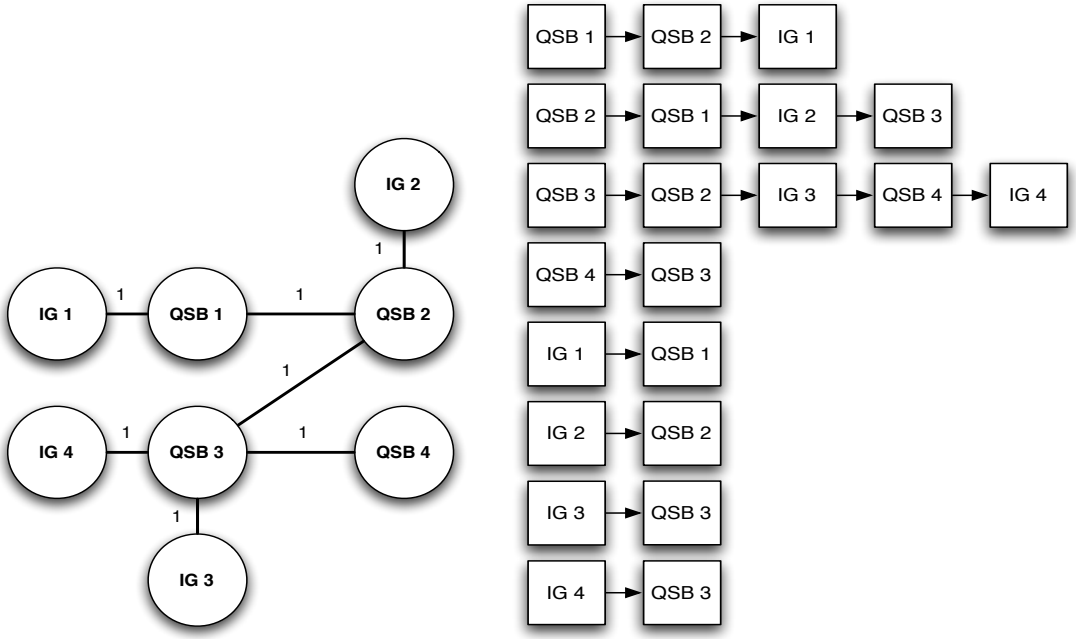
will end up without any results at all. The system could send a message to the client stating that an error has occurred, and might even encourage the client to try again later. In the case that all servers are replicated, or in the case of a *seletion* operation that requires data from a single relation that is partitioned across several sites, a missing server does not necessarily imply that the client will not receive any results.

A straight solution to this problem is to include a flag in the response message that is returned by the system to the client indicating that the solution is not complete or is unasserted. This will not necessarily comfort the user of the client application, but it will at least create awareness of the problem, and will left to the user or client application to decide whether to accept or reject the solution. A more interesting solution would be to have a metric of confidence for the solutions provided by the system. In this sense, a client could be told that the present solution is accurate with a 85% of confidence. Client applications could be tweaked to accept only solutions that are over a certain degree of confidence. In this thesis, only ideas of possible ways to deal with the issue of confidence of the solution is given. However, this thesis does not provide a concrete solution for the problem. Dealing with this issue is out of the scope of this thesis and has been left as future work.

## 3.6 Simplified Search of Data Sources Routes

The query optimizer must find the route to each of the data sources (and corresponding *IG*) needed for each query submitted to a NetTraveler federation. To achieve this, the *QSBs* use the information in the catalog to model the federation as an *undirected weighted graph* where all edges are given a weight of one. To find the routes, each *QSB* applies the *Breadth-First Search* (*BFS*) algorithm over an *adjacency list* representation of the graph using its own position as the starting point. Notice that it might be possible to obtain several routes for each data source. However, only the cheapest route (e.g shortest) is kept for each one. An in-depth discussion of graph theory and network algorithms is out of the scope of this thesis. The interested reader is referred to [37] for a comprehensive discussion of theory of graphs and the *BFS* algorithm.

Figures 3.20(a) and 3.20(b) show a federation modeled as a graph and the graph's adjacency list representation, respectively. Some of the federation's components have been removed for simplicity. The routes to each data source are stored by each *QSB* in a hashtable after applying the search algorithm. The routes are stored as linked lists where the tail node $n$ contains the information of the *IG* holding the desired data source. Every node from the head of the list to the $n-1$ node represents a hop in the network. Each hop represents a *fetch* operation that must be scheduled at the moment of query optimization (see Section 3.4.2.2). Table 3.1 shows the table of routes of *QSB* 4 for this particular example.




(a) NetTraveler federation as a graph

(b) Adjacency list representation of graph

Figure 3.20: NetTraveler federation represented as a graph and its corresponding adjacency list

| **Data Sources Routes Table** | |
|---|---|
| **key** | **entry** |
| **IG 1** | **QSB 3 → QSB 2 → QSB 1 → IG 1** |
| **IG 2** | **QSB 3 → QSB 2 → IG 2** |
| **IG 3** | **QSB 3 → IG 3** |
| **IG 4** | **QSB 3 → IG 4** |

Table 3.1: Routes table for *QSB* 4

# CHAPTER 4

# Implementation of the NetTraveler Framework

## 4.1 Overview

This chapter discusses the implementation details of the NetTraveler Middleware System. It begins by discussing the NetTraveler Query Services. The NetTraveler server applications known as the Query Services are the *Query Service Broker* (QSB), the *Information Gateway* (IG), the *Data Synchronization Server* (DSS) and the *Data Processing Server* (DPS). In this thesis we shall only discuss the QSB, the IG and the DSS. This chapter continues by discussing the mechanisms employed to recover the client-side execution of a query when a client suffers a disconnection. Finally, this chapter discusses the communication protocol used in NetTraveler.

## 4.2 Query Services Implementation

The responsibilities of the *Query Service Broker* (QSB), the *Information Gateway* (IG) and the *Data Synchronization Server* (DSS), inside a NetTraveler federation are logically different. The QSB is responsible of coordinating the execution of queries that are submitted by clients and

provides services such as query parsing and query optimization. The QSB has also the responsibility of helping other QSBs in solving queries by allowing access to data sources that are otherwise unreachable for these other QSBs. The IG is responsible of giving access to the QSBs to data stored in different types of data sources. Besides, IGs can also participate in query execution. The DSS helps in caching results intended for a client that is unreachable for some reason, like the lost in network connectivity, and that will request those results later.

However different the QSB, IG and DSS are conceptually, they are related in two ways. First, they are server applications belonging to NetTraveler. Second, they are server applications that participate in query execution. Java interfaces [38] were used to capture the relationships that exist between the QSB, IG and DSS. Java interfaces are types in the Java language, just like Java classes. However, they can only define methods but cannot implement them. The methods that are defined in the interface are actually implemented in a Java class that agrees to comply with the interface. In this sense, a Java interface serves as a contract that classes agree to comply with by implementing the methods.

Figure 4.1 depicts how the QSB, the IG and the DSS are related. The double-headed arrows represent relations by inheritance while the single-headed arrows represent relations by implementation of interfaces. As the figure shows, in Java an interface can inherit from another interface just as a class can inherit from another class. The *NetTravelerService* interface serves as the definition for all the NetTraveler components defined in Section 3.2. This interface defines the methods that must be equally supported by all the server applications in NetTraveler. Currently, this interface defines the initialization and cleanup routines that must be carried out by all servers. However, it is expected that this interface will change in the future as more server applications are implemented and new operations that are common to all servers appear.

The QSB, IG and DSS achieve the completion of their corresponding tasks by following a common iterator-like approach of execution. All three of these server applications provide a method for submitting a query-related task to them and a method for gathering the tuples that result from completing such task. The *QueryService* interface is responsible of defining these

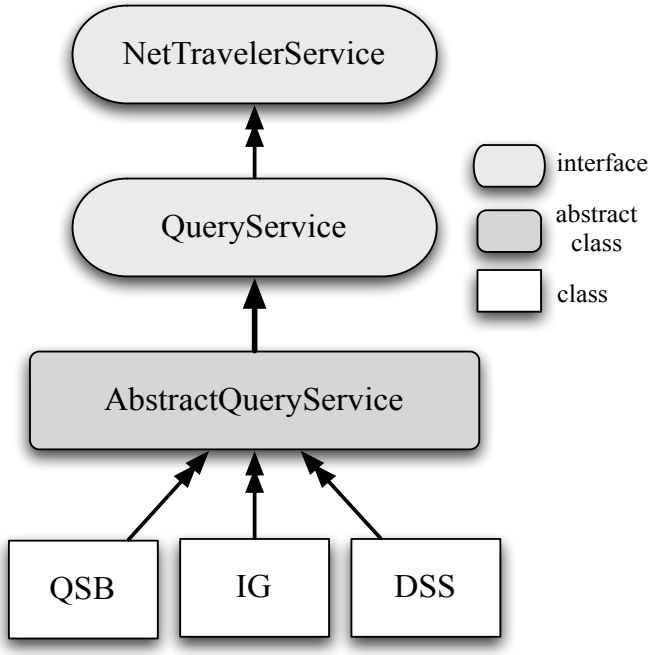



Figure 4.1: Query Services class hierarchy

methods. This interface extends the *NetTravelerService* interface and serves as the definition for all the NetTraveler server applications that participate in query execution. The methods that are defined in the *QueryService* interface are named *execute()* and *next()*. The following list details the functionality that classes implementing the *QueryService* interface must provide for each of these methods:

- *execute()* - This method is used to submit a query request to a Query Service. In the QSB this method can receive requests from clients and peer QSBs. In the DSS and the IG this method is limited to receiving requests only from the QSB.
- *next()* - This method is used to retrieve the results of a query that is being executed by a Query Service. This method must return a limited number of tuples each time it is invoked. This method must be called iteratively until all the results of a query are gathered.

The QSB server application further extends the *QueryService* interface by defining a method named *reroute()*. This method can be invoked by clients to ask the QSB to reroute the client-side execution of a query to a DSS known by the client.

The functionality that is equal for all the Query Services is implemented in the *AbstractQueryService* abstract class. The three servers, QSB, IG and DSS, inherit from this class. In fact, the differences in implementation between these three server applications strictly relies on the types of data structures they used and in the way they are initialized. Each server has a specialized initialization routine that differs from the other servers. For example, the IG has to open a pool of connections to the data source it will be accessing when initializing, which is not required by the QSB or the DSS. The QSB in turn must open a connection to a catalog that contains information regarding the federation to which the QSB currently belongs. Beside these differences in initialization routines, the Query Services attend both *execute()* and *next()* calls in the same way. They use a multithreaded scheme that relies on two entities known as the *query coordinator* and the *query worker*.

### 4.2.1 The Query Coordinator and the Query Worker

The *query coordinators* are set of specialized classes that are in charge of attending all types of requests that are received by a QSB, an IG or a DSS. The *query workers* are a set of classes that are in charge of actually executing queries inside the QSB, the IG or the DSS. Figure 4.2 depicts how the query coordinators are implemented. The *QueryCoordinator* interface defines the methods that all query coordinators must implement. The *AbstractQueryCoordinator* is an abstract class were the functionality that is shared by all query coordinators is implemented. The actual query coordinator classes for each Query Service inherit from this abstract class. The QSB, the IG and the DSS make use of query coordinators and query workers in almost the same way. Therefore, we will discuss every aspect of these components using the QSB as an example. Any differences between the QSB and the IG or the DSS will be noted when relevant.

When a QSB is initialized, a main thread of execution is generated. This thread, in turn, generates a group of query coordinators and assigns each one to a new thread of execution. The type of query coordinator generated will depend on the type of Query Service being initialized. For example, if the Query Service is an IG, a group of *IGCoordinators* is generated. For this case, the

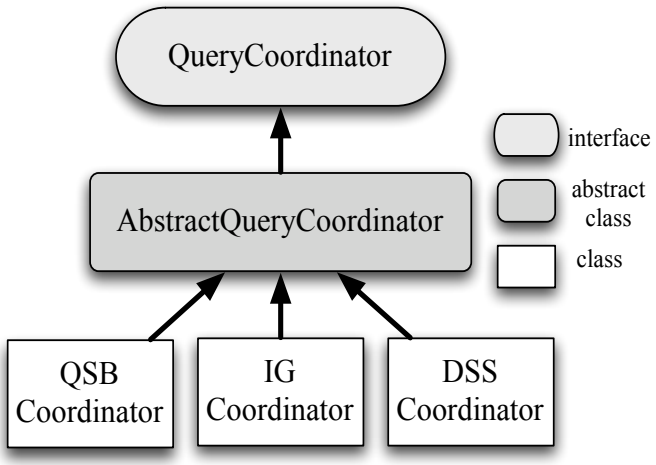



Figure 4.2: Query coordinators

query coordinators are of type *QSBCoordinators*. Figure 4.3 shows a generalized representation of a QSB immediately after initialization. As can be seen, the generated query coordinators are stored inside a queue from where they are removed by the main thread when any request needs to be processed. The threads in which the query coordinators run are kept in a dormant state while the query coordinators are in the queue.

Along with the query coordinators, a group of *query workers* is also generated when a QSB is initialized. As already stated, the responsibility of the query workers is to actually execute the part a query that corresponds to a specific Query Service. Each query worker generated will run it is own thread of execution. As Figure 4.3 shows, query workers look for queries to execute in a *Query Queue*. It is responsibility of the query coordinators to put queries that are ready for execution in the *Query Queue*. Query workers are put to sleep if they arrive at the *Query Queue* and there are no queries pending to be executed (e.g. the queue is empty). For instance, all the query workers shown in Figure 4.3 are in a dormant state.

We will now see how the query coordinators and the query workers are used by a QSB to execute a query. Figure 4.4 shows the steps performed by a QSB when it receives an *execute()* request. Coordinators sort out requests that are not allowed in a specific Query Service. For example, *execute()* requests submitted by clients are only accepted in the QSB. Meanwhile, *execute()*

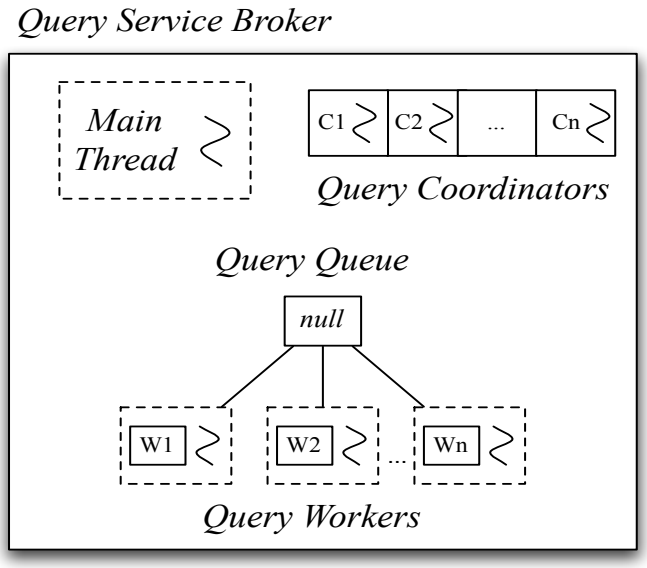



Figure 4.3: Query Service after initialization

request submitted by Query Services are accepted by QSBs and IGs. The *Client* in Figure 4.4 can represent either a QSB or a pure client application, since only these components can send *execute()* requests. The *main thread* of execution is responsible of receiving the *execute()* request, as denoted by *step 1* in the figure. Then, as denoted by *step 2*, the main thread will proceed to remove a query coordinator from the queue and will handle the received request to it. Requests are always attended in their own threads of execution since each query coordinator runs in its own thread of execution.

The query coordinator will proceed to prepare the query for execution. Its first task is to generate a unique query id *qid* for the received query. If the query was submitted by a client, the query coordinator will parse the query and will build a plan for the query by calling the query optimizer. Query optimization was discussed in Section 3.4.2. Parsing and optimization are steps that are not necessary if the query comes from a peer QSB, since the received request will already include the part of the query that the QSB must execute. After this, the query coordinator will generate an object to hold state information about the query. This state object is also used to cache tuples that result from the execution of the part of the query that corresponds to the Query Service receiving the *execute()* request.

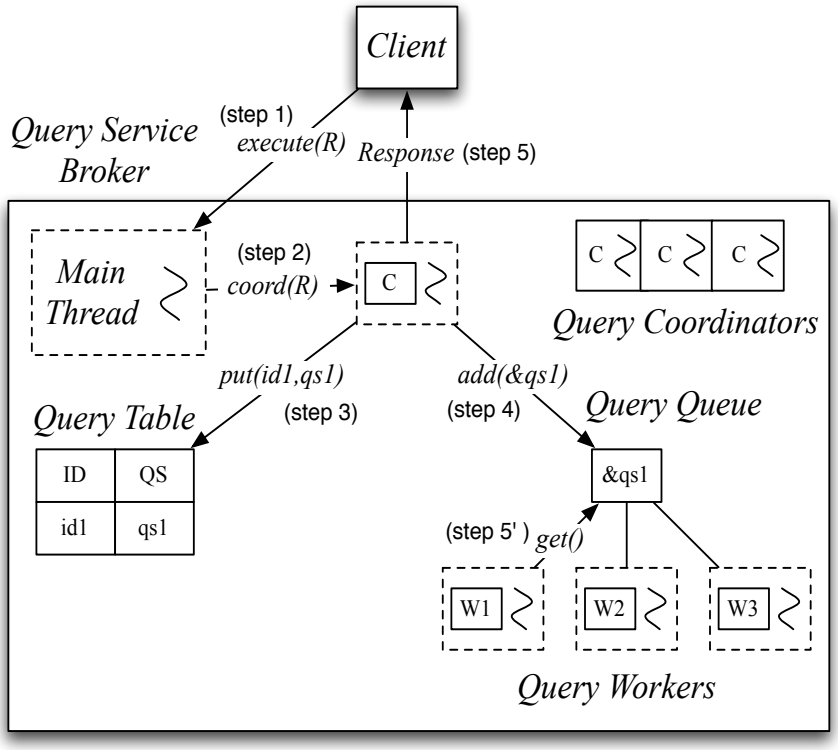



Figure 4.4: Query Service processing an *execute* request

The generated status object is put in the *Query Table* and a reference to it is inserted in the *Query Queue*. *Step 3* and *step 4* in Figure 4.4 denote these events. One of the sleeping query workers is notified when the query status object is inserted in the *Query Queue*. As denoted by *step 5′*, the query worker will wake up and will remove the query from the queue to start executing it. The query id *qid* that was generated by the query coordinator is then returned to the client or peer QSB that submitted the query. This id *qid* must be submitted to the QSB with each *next()* request that is issued to gather the result tuples of the query. It should be noticed that *step 5* and *step 5′* in Figure 4.4 occur in parallel. A query coordinator in the IG will process an *execute()* request in the same way that a query coordinator in the QSB process an *execute()* request sent by a peer QSB. A query coordinator in the DSS will also process an *execute()* request in the same manner, with the difference that it will not generate a new query id *qid* for every *execute()* request. Rather, the query coordinator in the DSS will use an id that will be handed to it by a QSB.

Figure 4.5 shows the chain of events that occur when a query is ready to be executed.

The query coordinator is returned to its corresponding queue and the thread in which it is running is put in a dormant state, as denoted by *step 1*. At the same time, the query worker will obtain the query plan that must be executed from the query status object. This plan is converted by the query worker into an executable instance of the plan (e.g. iterators). *Step 1′* in the figure denotes this event, which occurs in parallel with *step 1*. The query worker will then begin to execute the part of the query plan that corresponds to the QSB. Tuples that result from executing the plan are written to a cache in the query status object, as denoted by *step 2′* . If the Query Service is a DSS, the query worker will not actually execute a query plan but will rather just issue *next()* requests to a QSB.

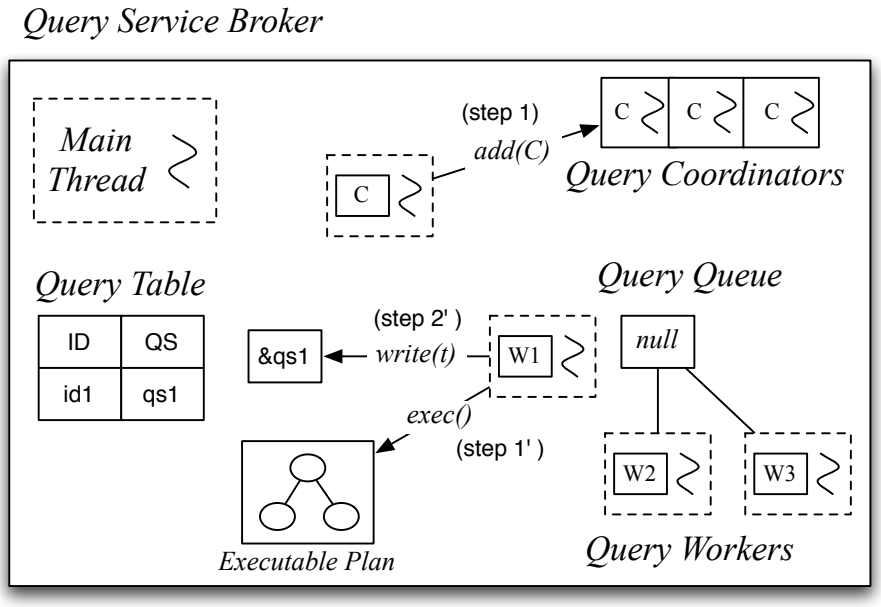



Figure 4.5: Query worker executing a query plan

The steps incurred by a QSB when a *next()* request is received are shown in Figure 4.6. The *Client* in this case can be a pure client application, a QSB or a DSS since they all can send *next()* requests. The *next()* request is received by the QSB exactly as the *execute()* request was received. The main thread of execution will get the request and will remove a query coordinator from the queue to hand the request to it. Notice how the query worker that is executing the query is generating result tuples in parallel with the arrival of the *next()* request. As with previous figures, parallel execution is denoted in the figure by two *steps* with same number but one of them having

an apostrophe. Hence, *step 1* and *step 1′* are events that occur in parallel, as well as *step 2* and *step 2′*.

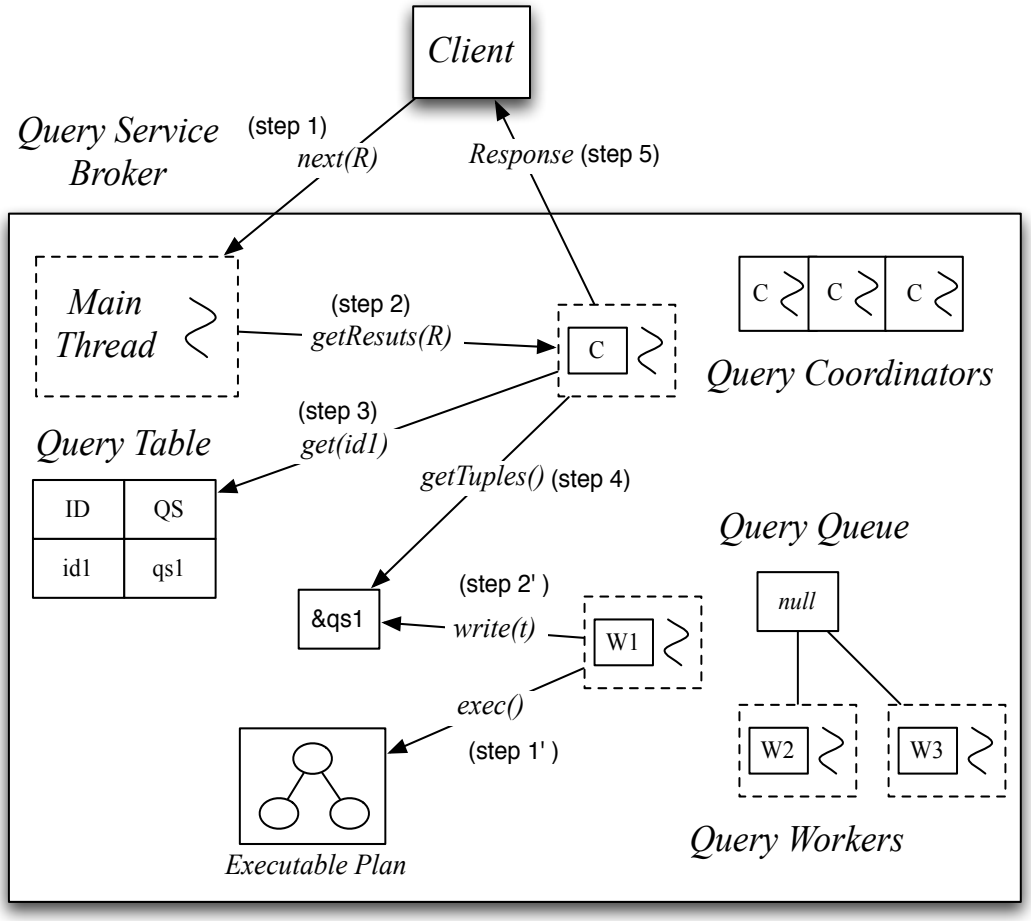



Figure 4.6: Query Service processing a *next* request

The query coordinator will get the appropriate query status object from the *Query Table* using the query id *qid* that is submitted in the *next()* request. This is denoted by *step 3* in Figure 4.6. Then, the query coordinator will read tuples from the cache in the query status object and will return result tuples inside a response message. *Steps 4* and *5* in the figure denote these operations, respectively. The query coordinator is returned to the queue of query coordinators after returning the response message to the client. The query worker will write tuples in the cache of the query status object until no more tuples are returned by the iterators that implement the query. At this point, the query worker will close the iterators and will search for more work in the *Query Queue*.

The client will continue to send *next()* request messages consecutively until all the result tuples are returned by the QSB.

## 4.3 Client-Side Query Recovery

We shall now discuss the mechanisms employed by NetTraveler to recover the client-side work of a query when the query is aborted by the client. But before, and for the sake of the following discussion, let us quickly and broadly refresh how a query is executed in NetTraveler. First, a client submits a query to a QSB by calling its *execute()* method. If the QSB receives the query, and is able to process it without errors, it will return an acknowledgment response to the client. After receiving the response, the client will repeatedly call the *next()* method of the QSB until it gathers all the result tuples of the query. The actions of calling the *execute()* and *next()* methods performed by the client when interacting with the QSB are known as the *client-side execution* of the query.

### 4.3.1 Motivation

A network failure, the lost of power, or a software crash can all cause a client device to lose network connectivity, disrupting the client-side execution of a query. In current middleware solutions, the query being processed before the disconnection is aborted, and the client needs to re-submit the query. This leads to slow response times as the query needs to be re-started from scratch. More importantly, the resources invested in processing the query are lost.

To give an example, let us assume that there is a relational schema with information about police precincts. This schema is presented in Table 4.1. The relations in this schema hold information about precincts, the police officers assigned to each precinct, the traffic tickets given by police officers to drivers, and the cars own by the drivers with traffic tickets. Let us also assume that there is one police precinct per city, and that each one has a local database that follows this schema. The database in each precinct is used to store data specific to the precinct. Hence, a

precinct in the city of Ponce will only contain information of traffic tickets given in Ponce. A middleware system is used to interconnect the data of three precincts located in three different cities: San Juan, Ponce and Mayagüez. For simplicity we will assume that this middleware is modeled exactly as NetTraveler, and that the components of this middleware have the same names as the ones in NetTraveler. Hence, a QSB receives all query requests and IGs retrieve data from the databases.

| **Relation** | **Description** |
|---|---|
| *Precinct*(pid:Integer, addr:String) | Stores information about a precinct |
| *Officer*(oid:Integer, pid:Integer, fname:String, lname:String, birthday:Date) | Stores police officers information |
| *Driver*(did:Integer, fname:String, lname:String, addr:String, birthday:Date) | Stores information about drivers |
| *Car*(cid:Integer, did:Integer, plate:String, make:String, year:Integer) | Stores cars information |
| *Ticket*(tid:Integer, cid:Integer, did:Integer, oid:Integer, date:Date, viol:String, dept:Numeric) | Stores tickets information |

Table 4.1: Schema of police precincts

Consider a user with a PDA in the city of San Juan that is interested in submitting the following query to the system: "Get the names of all drivers that have received at least one fine of over $100 in any police precinct". The client application running in the PDA will send this query encoded in SQL as follows [1]:

```
SELECT D.fname, D.lname
FROM Driver as D, Ticket as T
WHERE T.did = D.did;
```

By looking at the SQL query we quickly notice that, in order to answer this query, three geographically-apart databases must be accessed, one for the Mayagüez precinct, one for the Ponce precinct and one for the San Juan precinct. Figure 4.7 depicts this scenario. It is assumed for simplicity that the QSB in San Juan has direct access to the IGs at Ponce and Mayagüez (and of

[1]For simplicity, duplicate elimination was not taken into consideration.

course to the IG in San Juan). Another possible scenario would have been that the QSB in San Juan had to talk to a QSB in Ponce and to another QSB in Mayagüez, but this is irrelevant for the purpose of this example.

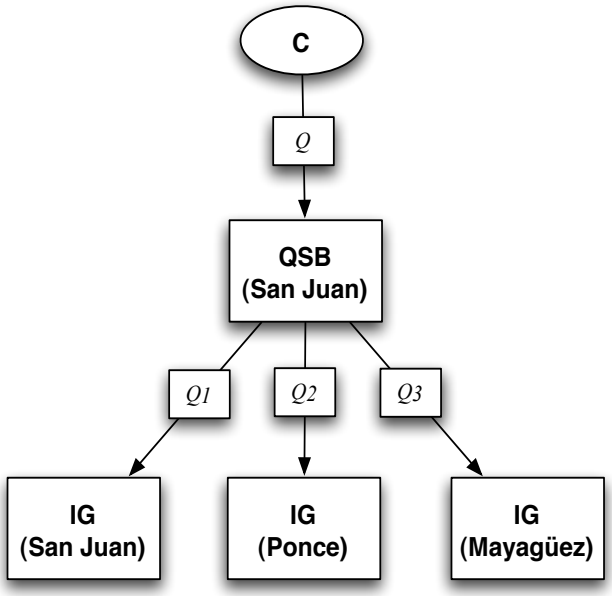



Figure 4.7: QSB that must access three data sources to solve a client's query

As can be seen in the figure, the query is received by the QSB in San Juan, who generates a plan to solve it and sends to each IG with access to the needed databases the part of the plan that they must execute. For each database, two different relations must be scanned and a *join* operation must be executed. Furthermore, the results must be shipped to the site that received the query before they can be served to the client. We must take into consideration that joins are expensive operations that require the employment of considerable amounts of resources in a machine. This is true even if we assume that joins are executed independently at each site and no distributed join [17] [18] is carried out. Also, if the result of the join is bigger than any of the two original relations, the cost of shipping the results to the site that received the query is also quite expensive.

Suppose that the expected query result is 1 MB. If a failure that causes the client to lose network connectivity occurs after the client has received some data, say 500 KB, the query will be aborted and the client will have to re-submit the query. Hence, the client will receive the results

from the beginning, downloading the first 500 KB again. The client will end up downloading 1.5 MB worth of data, since the failure made some of the work to be redone. Not only has the client downloaded more data than the necessary, but it has also wasted resources in terms of battery, CPU cycles, access to memory, and others, unnecessarily. Moreover, if by any chance the client device was using a pricey internet link, like a 3G connection, the user will also end up paying extra money due to the amount of extra time needed to complete the query thanks to the disconnection. Now, imagine this scenario repeating itself frequently. For the user of a client device with limited resources, like the PDA we have just seen, this situation is just unbearable.

The situation is not less unpleasant for the middleware system per se. Having to re-start this query several times is a clear waste of system resources. For example, a great deal of system resources are wasted in parsing, optimizing and building a plan for the same query, again and again, without ever completing the execution of the same. Not to mention the amount of resources that are employed in executing complex query operators, like joins, and in transferring large amounts of data over the network. CPU cycles, disk access, memory allocation, network sockets, are all but an example of the resources employed to execute a query of this type and that get wasted every time a disconnection in the client occurs. As if this was not enough, the system must also reclaim the resources in a timely manner to be able to continue attending new requests. This is not an easy task when you are spending a great deal of resources in executing complex queries that will not get completed.

Clearly, this traditional middleware query execution scheme is not fitted for a system that expects to have a large amount of clients residing in mobile, power-limited devices. In NetTraveler, this issue is addressed by leveraging on the system architecture to enable the rerouting of the client-side execution of a query.

### 4.3.2 Implementation of Client-Side Recovery

Recovery of the client-side execution of a query is achieved by *rerouting* the client-side execution of the query to another entity willing to work on behalf of a client. In NetTraveler, this is

achieved thanks to the Data Synchronization Server (DSS). As stated in Section 3.2, the DSS is a server application that, among other things, can become a *proxy* for a client, gathering the results intended for it. There are two possible scenarios for client-side query recovery in NetTraveler. Automatic recovery and explicit recovery. In both cases, rerouting of the client-side execution of the query is used to achieve query recovery. We shall see each of this two scenarios in detail in the following sections.

#### 4.3.2.1 Automatic Rerouting

Automatic rerouting of the client-side execution of a query occurs when a QSB finds a query that has been idle beyond an idle threshold time value $t_i$. This time value $t_i$ is defined as the maximum amount of time that a client expects to wait between successive *next()* calls to the QSB that is executing the query. The time value $t_i$ is specified by clients for each query that they submit for execution to a QSB. With each query, clients also include the information of a DSS that is willing to work on their behalf. A QSB that receives a query for execution will store the idle time $t_i$, along with the information of the DSS, as part of the state information of the query.

A QSB cannot instantly notice when a client suffers a disconnection, since a connectionless oriented protocol is used for communication. An established connection between two components in NetTraveler, for example between a client and a QSB, will be opened for the amount of time that it takes for one component to send a *request* and for the other component to return a *response*. That is why the QSB depends on the idle time $t_i$ and on receiving *next()* requests to know that a client is still connected to the network. Every time that a client issues a *next()* call, the QSB stores a time stamp of the moment the call was received. The QSB contains a *monitor thread* that periodically monitors all queries for the purpose of finding queries that have become idle pass the idle threshold time $t_i$ that was specified for the query when it was submitted. Using the time stamp of the query, the monitor thread calculates the amount of time that passed since the last *next()* call was made. If this calculated time is greater than the idle threshold time $t_i$ specified by the client, the query is said to be in an idle state, or simply idle. For each query that is found idle, the

monitor thread in the QSB is responsible of coordinating the rerouting of the client-side execution of the query.

To see automatic recovery in action let us use an example. Let us suppose that we have the same scenario that was used in Section 4.3.1 and that was depicted in Figure 4.7. Here, a PDA client device submits a query to a QSB in the city of San Juan to ask for information about drivers with tickets in the cities of Ponce, San Juan and Mayagüez. The amount of data in the result of this query is 1 MB. The user of the PDA sets the idle threshold time $t_i$ to be 15 seconds. Also, the user has access to a DSS in the city of San Juan which information it includes when submitting the query to the QSB. Now, let us assume that when 500 KB of data have already being received by the user, the PDA goes out of battery. Figure 4.8 shows the moment in which the client loses connectivity to the network. Notice that the QSB will not immediately detect this disconnection.

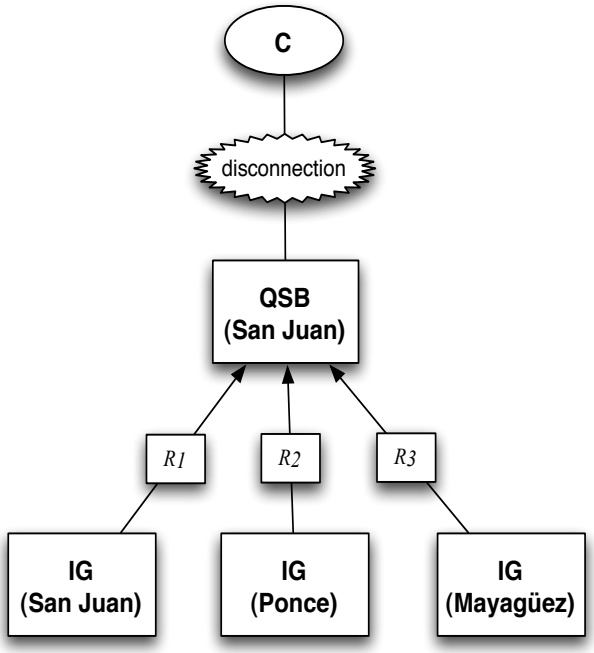



Figure 4.8: Client that suffers a disconnection in the middle of query execution

Figure 4.9 shows the steps that occur internally in the QSB when automatic rerouting is set in motion. *Step 1* depicts the moment in which the monitor thread inspect the query table and finds that the query has become idle. Remember from Section 4.2.1, that the QSB implementation of the

*Coordinator* interface has an extra method named *reroute()* that is specifically tailored to manage the rerouting of queries in the QSB. Hence, the monitor thread will remove a query coordinator from their queue and will invoke the *reroute()* method of the query coordinator, passing it a reference of the state information of the query that must be rerouted. This is denoted by *step 2* in the figure.

The query coordinator will get the information about the DSS to be contacted from the state information of the query and will communicate with the DSS, as denoted by *step 3*, to request that it continues with the client-side execution of the query. When requesting a reroute in the DSS, the query coordinator will submit the id of the client, the id of the query and the address of the QSB. If the DSS successfully receives the *reroute()* request, it will return a response to the query coordinator with an acknowledgment. After receiving the acknowledge from the DSS, the query coordinator will proceed to mark the query as *rerouted*.

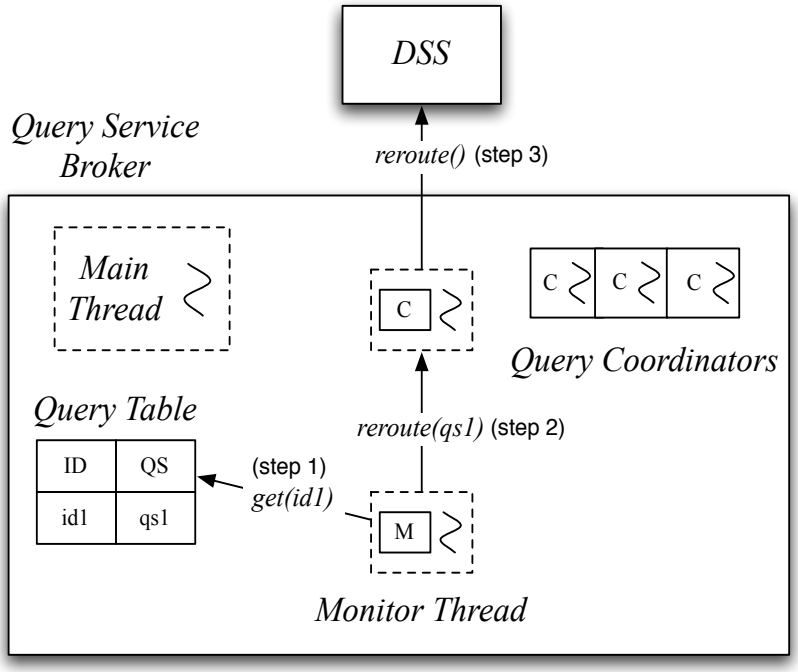



Figure 4.9: Monitor thread rerouting an idle query

When receiving a request to continue with the client-side execution of a query, a DSS will first authenticate the client using the information provided by the QSB. If the client is successfully authenticated, the DSS will begin to fetch tuples on behalf of the client by repeatedly performing

*next()* calls on the QSB. Figure 4.10 shows a DSS that is executing the client-side of a query after a reroute operation. A *next()* call issued by a DSS is seen by the QSB that is coordinating the query as a *next()* call issued by the client that originally submitted the query. The DSS is just acting as a proxy for the client. The DSS will store the remaining 500 KB of data intended for the client until the client regains connectivity.

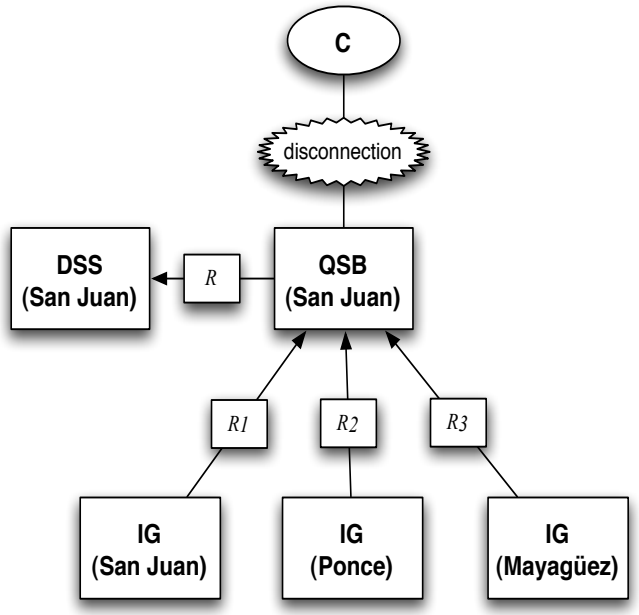



Figure 4.10: DSS executing the client-side work of a query

Figure 4.11 shows the chain of events that occur when the client recovers from the disconnection. The client will first issue a *next()* call on the QSB that was executing the query. The QSB will inform the client that the query has been rerouted to the DSS specified by the client. This is denoted by *step 1* in the figure. The client will then contact the DSS that is working on its behalf and will fetch the remaining 500 KB of data for the query, as denoted by *step 2*. This effectively completes query execution, from the point where the disconnection in the client occurred and without restarting the query.

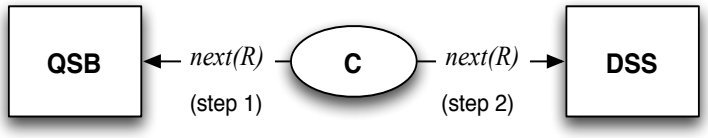



Figure 4.11: Client recovering from disconnection and gathering the result from a DSS

#### 4.3.2.2 Explicit Rerouting

The second scenario for client-side query rerouting is that the client application explicitly issues a *reroute()* request to the QSB that is coordinating the execution of the query. In this case, the client application will determine that it must abandon the execution of the current query based on some criteria specified by the user. It may be that the user specified a maximum amount of time to wait for the completion of the query or it may be that the device determines that it must shut down because it is running low on battery. In any case, the logic in the client application will trigger an event that will issue a *reroute()* call on the QSB that is coordinating the query.

Let us see how explicit rerouting occurs using the same example of the previous section, where a user with a PDA submits a query to a NetTraveler federation. Again, the result of the query is about 1 MB of data. We already saw how recovery was achieved when the PDA lost its network connection abruptly because it went out of battery. Now, suppose that the client application running on the PDA monitors the battery percentage of the PDA and that it is configure to trigger a reroute event if the battery percentage goes below 10 in the middle of a query. Suppose that the client submits a query for execution and when 500 KB of data have been downloaded the battery goes below the specified threshold. This causes the client application to issue *reroute()* call on the QSB.

Figure 4.12 shows the events that occur when a client application explicitly issues a *reroute()* request. As denoted by *step 1*, the main thread of execution in the QSB receives the *reroute()* request from the client. Then, it removes a query coordinator from their queue and hands it over the request, as denoted by *step 2*. In automatic rerouting, the monitor thread forwards the information of the query to be rerouted to the query coordinator. In explicit rerouting, the query

coordinator looks for this information in the table of queries. This operation is denoted by *step 3*. Once the query coordinator has all the information needed, it proceeds to make a *reroute* call to the DSS just as in automatic rerouting. *Step 4* denotes this operation.

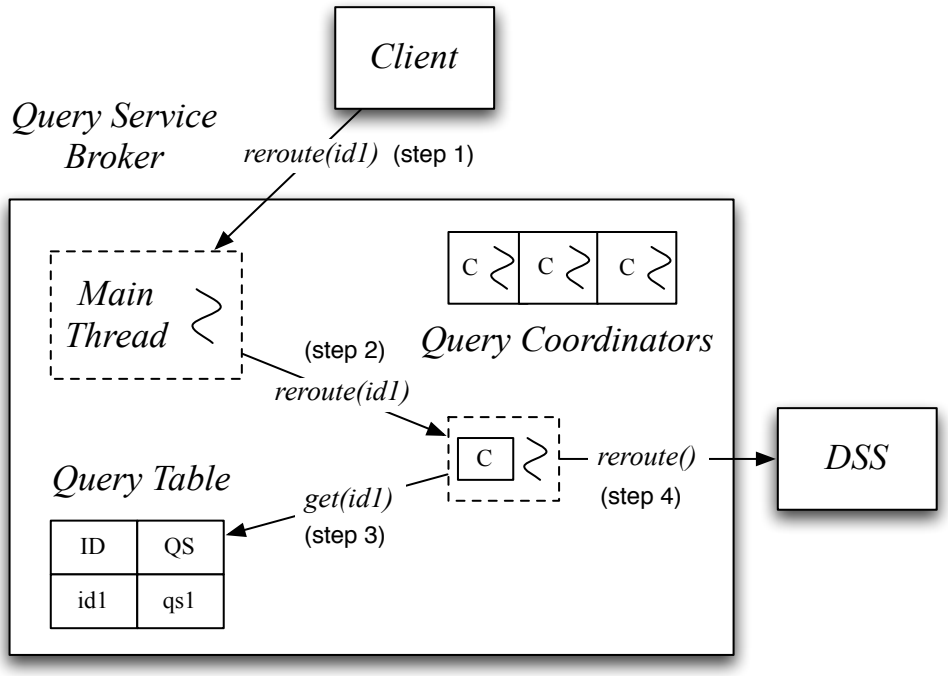



Figure 4.12: Explicit rerouting of the client-side execution of a query

If the DSS successfully receives the *reroute()* request and is able to authenticate the client, the query will continue to be executed with the DSS acting as a proxy for the client. When the client recovers from the disconnection, it will not communicate with the QSB that was coordinating the query as in the case of automatic rerouting. This is an obvious course of action since the client explicitly issued the reroute operation and knows that the remaining results of the query are in the DSS. Instead, the client application will communicate directly with the DSS that has the results and will gather the remaining 500 KB of data from it.

### 4.3.3 Summary of Client-Side Query Recovery

As we have seen, in current middleware solutions all efforts for completing a query are abandoned at the first sign of a disconnection by a client. Dropping queries in this manner wastes

both system and client computational resources every time a query is re-started. With rerouting of the client-side execution of a query system and client resources are not wasted since queries that suffer a client-side disconnection are never re-started. Also, clients do not end up downloading more data than what they really needed when a disconnection occurs.

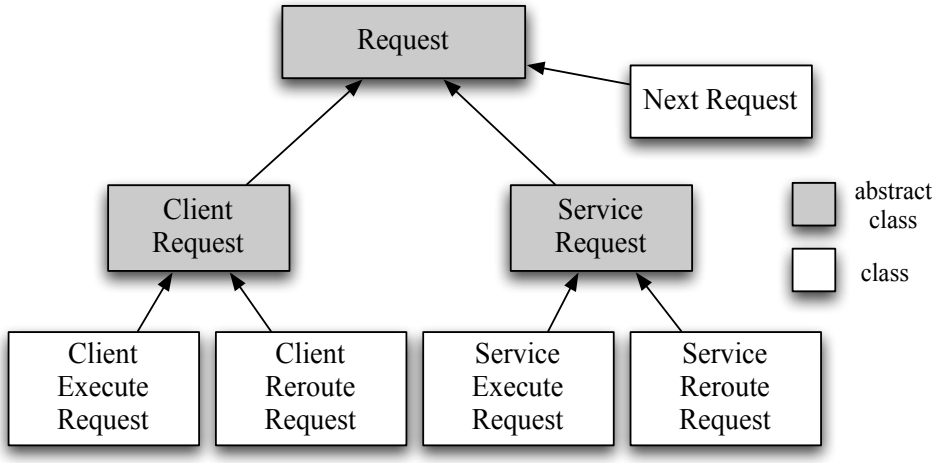
## 4.4 Query Services Communication

In this section, the details of how the Query Services communicate with each other and with clients are discussed. The section begins by describing the different request and response objects that are used. Then, the section continues by discussing how the Query Services were implemented as Web Services using the Axis Soap Toolkit. Finally, the issues found with the Axis Soap implementation, and how they were circumvented, are detailed.

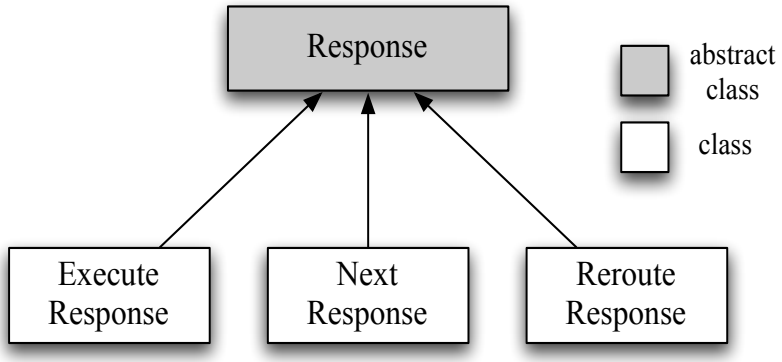
### 4.4.1 Requests and Responses

A single method of a Query Service can receive more than one type of request message. For example, the *execute()* method of the QSB can receive requests that come from a client and can also receive requests that come from another QSB. Therefore, each of the Query Services needs to differentiate between the types of request that they receive. This is accomplished thanks to a hierarchy of *request* and *response* message objects. Figure 4.13 depicts the different requests and responses objects and how they are related. Single-headed arrows in the figure indicate relations by class inheritance. As their names suggest, requests that inherit from the *ClientRequest* class are sent by clients and requests that inherit from the *ServiceRequest* class are sent by the Query Services. The *Response* message objects are strictly sent by the Query Services only.

The *ClientExecuteRequest* message object is sent by clients when invoking the *execute()* method of a QSB to submit a query for execution. As part of this request, a client will send a query $q$, encoded in SQL, a profile object $p$ and the information of a DSS $d$ that is known by the client. The profile object $p$ contains a client id *cid*, that is used to identify the client in future requests




(a) Request hierarchy




(b) Response hierarchy

Figure 4.13: Request and response hierarchy

regarding the same query, and an idle time $t_i$. The idle time $t_i$ is defined as the maximum amount of time that a client expects to wait between successive *next()* calls to the QSB that is executing the query. This idle threshold value $t_i$ is used by the QSB to determine if the client-side execution of a query must be rerouted to the DSS $d$. As we shall see in Section 4.3, the QSB contains a specialized thread of execution that constantly monitors all active queries for the purpose of finding queries that become idle pass the idle threshold time $t_i$.

The *ServiceRerouteRequest* message object is sent by a QSB to ask a DSS to continue with the client-side execution of a query on behalf of a client. The *ServiceRerouteRequest* is sent when invoking the *execute()* method of the DSS. Another way in which the client-side execution of a query can be rerouted is if the client explicitly asks for it. The *ClientRerouteRequest* message object is used by clients for this purpose and is sent when invoking the *reroute()* method of the QSB. The *ServiceExecuteRequest* message object is sent by QSBs issuing *execute()* requests to an IG or a peer QSB. As part of this request, a QSB will send the full query plan, the sub-plan that the receiving QSB or IG must execute and its own service id *sid*. The purpose of the service id *sid* is the same as of the client id *cid*; to enable IGs and peer QSBs to identify the QSB that submitted the query when it issues *next()* calls at a later time. As can be seen, the *execute()* method of the QSB receives both *ClientExecuteRequest* and *ServerExecuteRequest* message objects.

The *NextRequest* message object is sent when a *next()* call is submitted to a QSB, IG or DSS. This message object is sent by clients, QSBs and DSSs when asking for the results of a query. Information that must be submitted as part of this request includes the query id *qid* of the query for which results are desired, the id of the requester and the maximum number of results that should be returned in the corresponding response message. Recall that the query id *qid* is generated by a QSB or an IG when they receive an *execute()* request. The query ids generated by QSBs and IGs are returned inside an *ExecuteResponse* message object, which is one of the three types of response objects that exists in NetTraveler. The hierarchy of response objects used by the NetTraveler Query Services is depicted in Figure 4.13(b), conveniently repeated in Figure 4.14.

The *NextResponse* message object is returned by a Query Service after receiving a *Nex-*

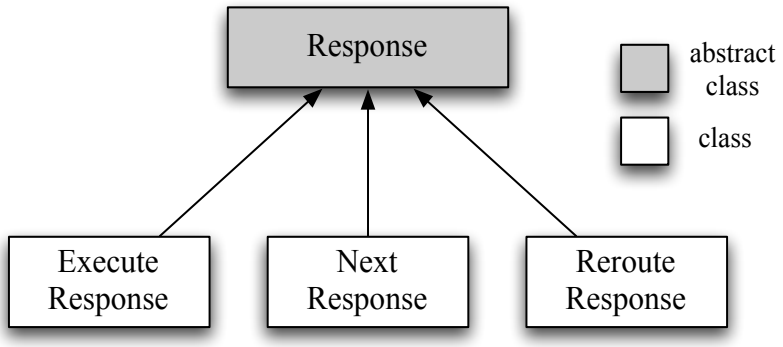



Figure 4.14: Response message hierarchy

*tRequest*. Tuples that result from the execution of the query are returned inside a *NextResponse* object. The *RerouteResponse* message object is returned by the QSB as the response for a *reroute()* call made by a client. All response objects contain a *code* that serves as information regarding the state of an operation. For example, if a *ClientExecuteRequest* message object is successfully received and processed by a QSB, then an *OK* code is returned inside the *ExecuteResponse* message object. However, if an error occurs during the evaluation of the request, an *ERROR* code is sent back in the response message. These two code objects are also returned inside the *NextResponse* and the *RerouteResponse* message objects. Other message codes used are the *DONE* code and the *REROUTE* code. The *DONE* code is returned in a *NextResponse* message object when the execution of a query is completed. The *REROUTE* message code is also sent inside a *NextResponse* object. It tells a client that the query was rerouted, and that the results of a query are being held by the DSS that the client explicitly specified in the *ClientExecuteRequest* message object.

### 4.4.2 Communication Implementation

The implementation of the Query Services is not attached to any communication protocol. In fact, the Query Services were specifically implemented as server objects independent of a specific communication mechanism. The communication protocol to be used must be provided by another layer of functionality that must be attached to each of the Query Services. The Query Services were implemented in this way to allow flexibility at the moment of deciding a communication protocol to use. As part of this thesis, only one type of communication protocol layer was implemented

for the Query Services. The communication protocol used by this layer is Soap over HTTP. The communication layer was implemented using the Apache Axis Soap Toolkit for Java Web services. Web services were chosen since they allow us to leverage on many tools for resource discovery, metadata representation, networking communications and security infrastructure.

The layered organization of the Query Services slightly variates when they are published as Web services. As an example, look at Figure 4.15 (which is the same Figure 3.2 from Chapter 3) and compare it to Figure 4.16. Figure 4.15 shows the internal organization of the QSB

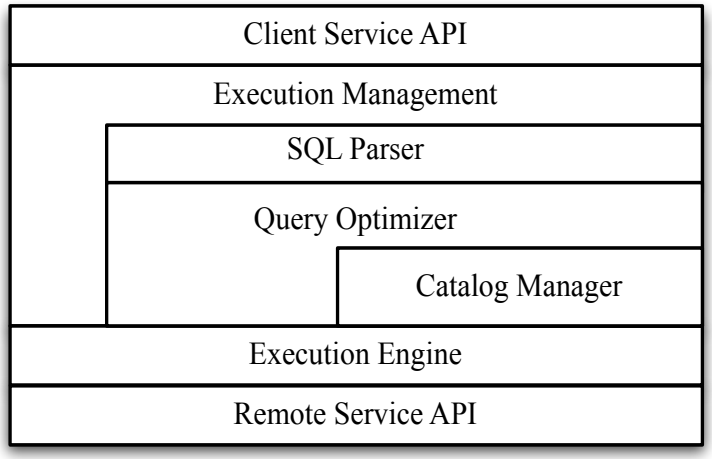



Figure 4.15: QSB internal organization

without any communication layer attached to it. Figure 4.16 shows the internal organization of the QSB with the layers for Soap communication added. The *Soap Server Engine* layer contains an instance of the Axis Soap Server Engine. The *Soap Server Engine* decodes the Soap request messages that are received by the QSB, generates the Java objects specific to each type of request (e.g. *ClientExecuteRequest*, *ServiceExecuteRequest*, etc) and hands these Java objects to the *Client Service API* layer. In the same manner, the *Client Service API* layer returns the response messages as Java objects to the *Soap Server Engine* which encodes them as Soap response messages and returns them. The *Soap Client Engine* contains an instance of the Axis Soap Client Engine. It encodes the requests objects that are sent by the QSB to other components as Soap request messages. It also decodes to Java objects the Soap response messages that the QSB receives.

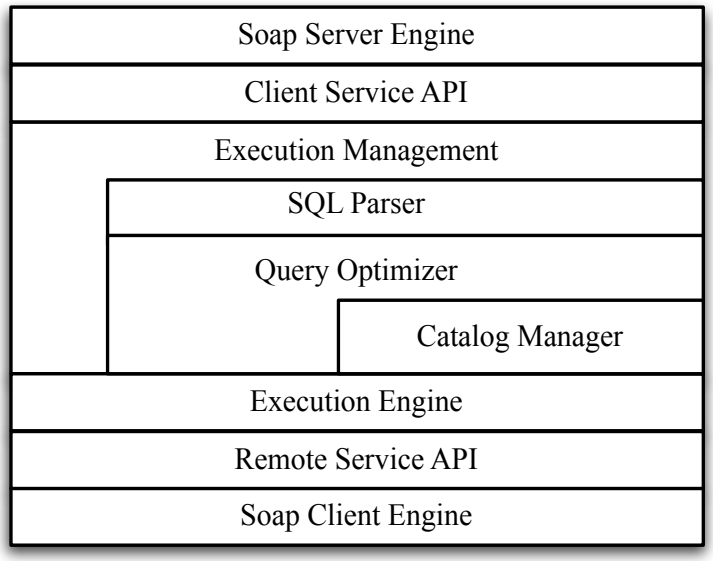



Figure 4.16: QSB internal organization with Soap layers added

These same two layers were added to the internal components of the IG and the DSS to publish them as Web services.

Figure 4.17 depicts how the QSB is implemented as a Web service. As can be seen in the figure, the Java class that is actually published as a Web service acts as a wrapper for the QSB. The same wrapper was used for the QSB, the IG and the DSS. This means that the *reroute()* method is actually present for all three Query Services when published as Web services. However, the wrapper class is aware of the type of Query Service that it is wrapping and rejects *reroute()* requests submitted to IGs and DSSs. There is a single WSDL [39] file that contains the information needed to access the QSB, the IG and the DSS when they are published as a Web services. This information includes the service namespace, the service name, the port name and the signatures of the methods that can be invoked in the Web service. The implementation of a specific Soap client object for each Query Service was not needed. Rather, a single client implementation was done using the WSDL file. This client object can be used to communicate with any Query Service.

The NetTraveler Query Services currently communicate with each other by exchanging Soap Document/Literal messages based on remote procedure calls (RPC). In this type of Web service implementation, the Soap messages exchanged are a direct mapping of the signature of the

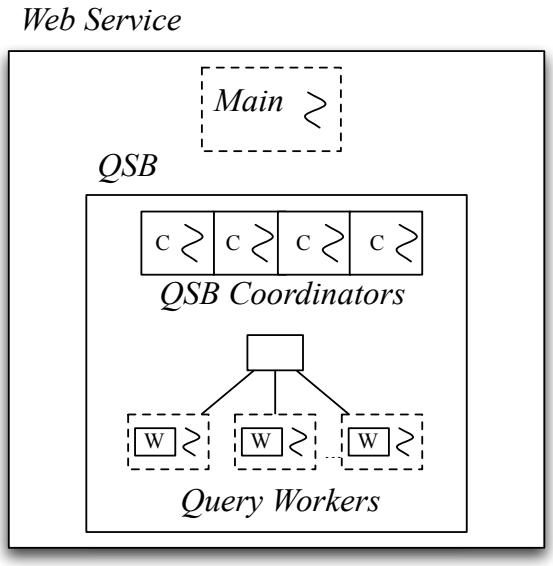



Figure 4.17: QSB as a Web service

methods defined in the Java class that is published as a Web service. This is done by using an XML struct to represent the call to the specific method and another XML struct to represent the value returned by the method. For example, if a class a class that defines the following method is published as a Web service :

```
public String reverseString(String s)
```

The Soap request message for this method will look something like this:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
        soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/1999/XMLSchema">
        <soap:Body>
                <m:reverseString xmlns:m="urn:myns">
                        <s xsi:type="xsd:string">Elliot</s>
                </m:reverseString>
        </soap:Body>
</soap:Envelope>
```

It can be seen how the method name has been translated into XML and how the method invocation is encoded as a struct. Each argument passed to the method is passed as an XML element inside the element that represents the method. The Soap response message that corresponds to the invocation

of this method will look like this:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
        soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/1999/XMLSchema">
        <soap:Body>
                <m:reverseStringResponse xmlns:m="urn:myns">
                        <rev xsi:type="xsd:string">Toille</rev>
                </m:reverseStringResponse>
        </soap:Body>
</soap:Envelope>
```

The Query Services have also support for Soap messages with attachments. We felt that support for attachments was necessary to eventually exchange not only XML messages but different types of data, such as binary files and images, as well. Nonetheless, the current performance of the Axis Web Services forced us to use attachments for a purpose different than the originally intended as we shall see in the next section.

### 4.4.3 Soap Performance Issues in Axis

The message objects discussed in Section 4.4.1 were implemented as Java Beans. Java Beans are classes that contain a pair of getter and setter methods for each attribute in the class. Another characteristic of Java Beans is that they must have an empty constructor. Using a description file, the Axis Soap Toolkit can map Java Beans to an encoded representation in XML format. This way, the Axis Soap engine can automatically serialize a Java Bean into XML and embed it in a Soap message. A Java Bean can also be deserialized and reconstructed from a Soap message. The process of serializing a Java Bean into its XML representation is known as *marshalling*. Deserializing a Java Bean from XML is known as *unmarshalling*.

Marshalling and unmarshalling of Java objects is still a very expensive operation. The reason is that Axis relies heavily on the Java Reflection mechanism. Reflection is the Java mechanism to dynamically load classes into memory at runtime. The use of this mechanism causes multiple objects to be instantiated by the Axis Soap Engine when it has to marshall and unmarshall request

and response message objects. Axis has to instantiate the objects that represents the Soap messages per se. Besides, it also has to instantiate the objects that represent the request and response message objects that travel inside the Soap messages. Not to mention that it also has to instantiate the objects that are members of these message objects. In the end, up to thousands of objects can be instantiated when multiple requests messages are received concurrently by a Query Service.

These objects end up being used only once and then reclaimed by the Java Garbage Collector. The performance of the Java Garbage Collector decreases considerably as more and more request are processed concurrently by a Query Service. Remember that for every request received there is a corresponding response returned. This decrease in performance continues until eventually the maximum amount of memory available to the Java Virtual Machine is reached. At this point, the Query Services start to behave erratically, dropping all incoming requests, and barely being able to complete the execution of queries that were already running. The situation cannot be alleviated by increasing the amount of available memory to the Java Virtual Machine, since eventually the saturation point is reached again.

To deal with this situation, all request and response message objects are sent as attachments in the Soap messages exchanged. Figure 4.18 shows an example of a QSB exchanging Soap messages with an IG. Suppose that the QSB will send a *ServiceExecuteRequest* message object to an IG. After the message object is constructed, the QSB will convert it into bytes using the Java serialization mechanisms. Then, the QSB will attach these bytes to the outgoing Soap request message. The IG, in turn, will get the bytes from the attachment in the Soap request message and will reconstruct the *ServiceExecuteRequest* message object. Once the IG finishes processing the request, it will generate a new *ExecuteResponse* message object. As with the request object, this response object is converted into bytes and attached to the Soap response message. Finally, the QSB can see the result of the operation by reconstructing the response object from the Soap response message.

Attachments in Soap messages are not processed by the Axis Soap Engine. Thanks to this fact, only the objects that are used to represents the Soap messages are instantiated. This reduces

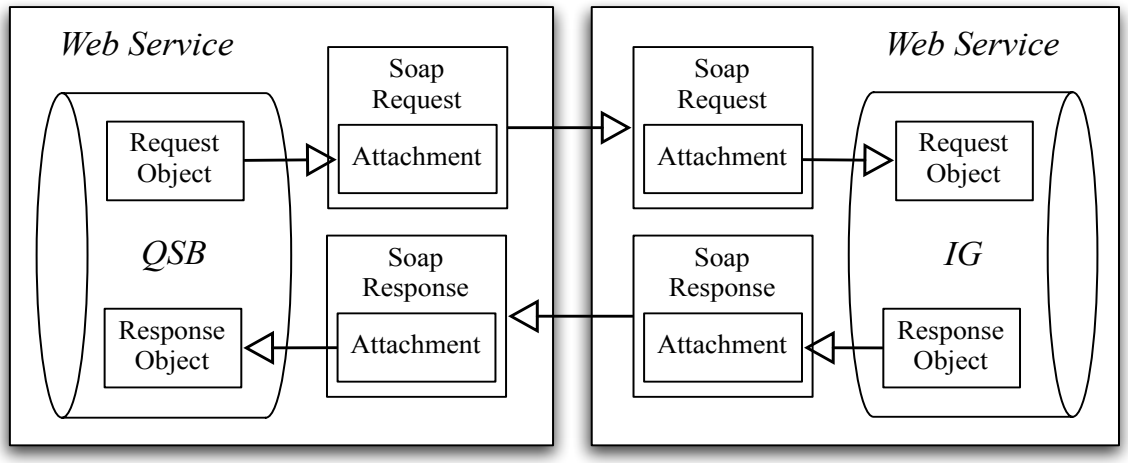



Figure 4.18: QSB and IG exchanging Soap messages with attachments

instantiation overhead considerably, since these objects are small and are handed efficiently by the Java Virtual Machine. The Query Services are able to handle a greater amount of concurrent requests using attachments than by sending request and response message objects encoded as XML. Nonetheless, these unconventional use of attachments is expected to be eliminated once the performance of Axis, or another Java Soap implementation comparable to Axis, achieves an acceptable performance level.

# CHAPTER 5

# Experimental Results

## 5.1 Introduction

This chapter presents the experiments that were carried out in order to validate the ideas presented in this thesis about a connectionless query execution engine and query recovery. The specific purpose of the experiments were to test:

1. If in the presence of disconnections by clients, NetTraveler can achieve a greater throughput, defined as the number of queries solved per unit of time, than a system that lacks the capability of recovering the query work of a client.
2. If the P2P architecture of the QSBs actually allows NetTraveler to withstand greater workloads than a typical middleware system that has a centralized architecture and that lacks any mechanism for query recovery.

An experiment was setup with a federation having a fixed number of clients and a fixed topology organization to test the first point. Using this federation, the number of disconnections per client was varied between 0, 3, 5 and 10. Also, the experiments were ran with query recovery and then without query recovery for each number of disconnections per client. This experiment is detailed in Section 5.2.1.

The second point was tested by performing two experiments. In the first one, the behavior of a NetTraveler federation was tested by organizing it using different topologies. Basically, this experiment tested if there appears to be a topology configuration for a federation that is superior to others. For each topology organization, there was a fixed number of disconnections per client and the number of clients for each topology was varied between 5 and 10. This experiment is detailed in Section 5.2.2. In the second experiment carried out to test the second point, a federation with a centralized architecture, only one QSB, was setup to compare the performance of NetTraveler against a centralized system that lacks query recovery. This experiment is detailed in Section 5.2.3.

Several assumptions and simplifications were made to carried out these experiments:

- A federation will not be dissolved or changed at any time during the experiments.
- Delays introduced by network connections in a Wide Area Network were not taken into consideration.
- The threshold idle time $t_i$, the time that a client expects to be without issuing a *next ()* request, was set to 30 seconds for all clients in all experiments.
- A connection-oriented middleware system was simulated by "turning off" recovery in NetTraveler.

To "turn off" query recovery we set the thread that monitors all queries in a QSB (Section 4.3.2) to sleep for half of the minimum amount of time that clients expect to be without issuing a *next()* request for a query, 15 seconds. The monitoring thread removed any query that it found idle beyond the threshold value $t_i$ when it waked up. Disconnection failures were distributed by each client randomly in time. The time of each disconnection was chose randomly between 30 and 90 seconds.

The number of components used for the experiments, with the exception of when a centralized topology was used, was 4 QSBs, 8 IGs and 1 DSS. Each QSB had access to 2 IGs and all clients used the same DSS. In the cases where a centralized topology was used, a single QSB knew how to access the 8 IGs. Each run in an experiment lasted 20 minutes and was repeated 3 times.

The average of the repetitions was used as the result values. Each Query Service and every client was ran in a different machine. The specifications of the machines used and what services were ran on them, one per machine, are in Table 5.1.

| **Machine Specs** | **Query Service** |
|---|---|
| Dell PowerEdge 2865, 3.0 GHz Intel Xeon, 1 or 2 GB RAM | 4 QSBs and 2 IGs |
| Dell Precision 360, 3.2 GHz Pentium 4, 1 GB RAM | 6 IGs, 1 DSS and all clients |

Table 5.1: Machines used for the experiments

### 5.1.1 Experimental Data

The data used for the experiments followed a schema based on information about police precincts. This schema is detailed in Table 5.2. The *Precinct* relation holds information about a police precinct. Each precinct has information about police *Officer*s and the *Ticket*s that they have given. For each ticket there is an associated *Car* and *Driver*. For simplicity, it was assumed that all drivers with tickets are car owners, and that they were given tickets driving their own cars. There are no drivers that were given tickets using cars that were borrowed, rented or acquired by any other means. Hence, the amount of drivers and cars in the database is the same.

We simulated a federation were each IG contained information regarding a single police precinct. To do this, relations were stored by horizontally fragmenting them inside the federation. This means that all *IGs* in a federation had a subset of the records of each relation. Table 5.3 depicts an example of how the *Officer* relation was horizontally fragmented. The attribute *pid* is the id of the precinct to where each officer is assigned. The relation was partitioned so the information in each *IG* is specific to a precinct. This way, the simulated federation represents a group of police precincts that are geographically apart, each having information specific to their geographical area, that were united by NetTraveler to obtain a unified representation of information spread over a wide area without having to move all the information to a centralized, bloated, database. The *union* operator is used to reconstruct each of the original relations. The PostgreSQL 8.1 relational database management system (RDBMS) was used as the data source for each site. It was also used

| Relation | Description | Cardinality | Size |
|---|---|---|---|
| *Precinct*(pid:Integer, addr:String) | Stores information about a precinct | 8 | 184 B |
| *Officer*(oid:Integer, pid:Integer, fname:String, lname:String, birthday:Date) | Stores police officers information | 480 | 28.1 KB |
| *Driver*(did:Integer, fname:String, lname:String, addr:String, birthday:Date) | Stores information about drivers | 2400 | 375 KB |
| *Car*(cid:Integer, did:Integer, plate:String, make:String, year:Integer) | Stores cars information | 2400 | 84.4 KB |
| *Ticket*(tid:Integer, cid:Integer, did:Integer, oid:Integer, date:Date, viol:String, dept:Numeric) | Stores tickets information | 6151 | 504.6 KB |

Table 5.2: Experiments database schema

to stored the catalogs used by the QSBs.

| **Officer** | | | | | |
|---|---|---|---|---|---|
| **IG** | **oid** | **pid** | **fname** | **lname** | **birthday** |
| **IG 1** | 45033 | 1 | Dennise | Medalla | 1975-10-28 |
| | 45154 | 1 | Bart | Aniston | 1959-03-21 |
| **IG 2** | 45103 | 2 | Alf | Iverson | 1979-09-03 |
| | 45148 | 2 | Nestor | Vargas | 1978-04-27 |
| **IG 3** | 45074 | 3 | Peter | Parker | 1980-09-10 |
| | 45156 | 3 | Wilson | Ginobili | 1981-09-12 |

Table 5.3: Officer relation showing horizontal fragmentation

### 5.1.2 Experiment Queries

Table 5.4 shows the queries that were used for the experiments. The table also shows the number of results and the amount of data returned by each query. *Query 1* gets the information of all the police officers of all precincts. This query returns 28.1 KB of data and in average takes 5.56 seconds to complete. *Query 2* gets information of all the tickets that have been given in all the precincts. This query returns 504.6 KB of data and on average takes 65.04 seconds to

complete. *Query 3* is the most complex query and gets the information of all drivers that own cars. The complexity of this query relies on the fact that a *join* operation must be carried out. Join operations are always performed in the QSB that is coordinating the query. This query returns 121.9 KB of data. Nonetheless, because a join operation must be carried out, this query takes 64.08 seconds to complete, almost the same time as *Query 2*, which returns almost 5 times more data.

| Name | Query | Cardinality | Size | Time |
|---|---|---|---|---|
| *Query 1* | *SELECT * FROM Officer* | 480 | 28.1 KB | 5.56 s |
| *Query 2* | *SELECT cid, date, viol, debt FROM Ticket* | 6151 | 504.6 KB | 65.04 s |
| *Query 3* | *SELECT D.fname, D.lname, D.did FROM driver as D, car as C WHERE D.did = C.did* | 2400 | 121.9 KB | 64.08 s |

Table 5.4: Queries used for the experiments

## 5.2 Experiments

We shall now turn our attention to the discussion of the experiments that were carried out to validate the ideas presented in this thesis.

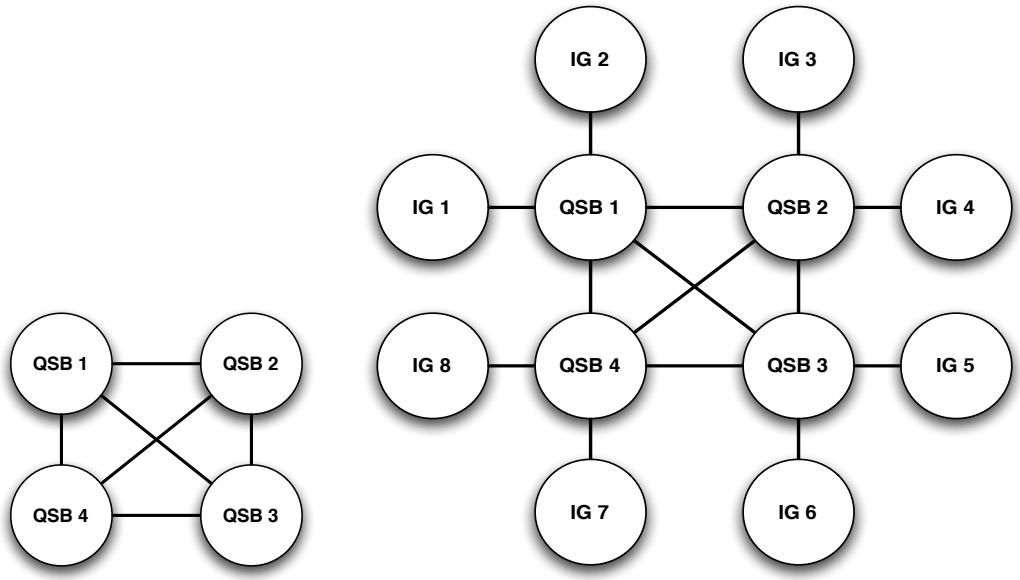
### 5.2.1 Throughput Experiment

The goal of this experiment was to test the following hypothesis: "In the presence of disconnections suffered by clients NetTraveler can achieve a greater throughput, defined as the number of queries solved per unit of time, than a system that lacks the capability to recover the query work of a client".

#### 5.2.1.1 Methodology

For this experiment we set up a NetTraveler federation with 4 QSBs, 8 IGs and 1 DSS organized in a mesh topology. Figure 5.1 depicts the organization of the federation for this exper-

iment. The topology of the federation is denoted by how the QSBs are organized. Figure 5.1(a) shows a view of the federation for this experiment with the QSBs only. Each QSB had access to 2 data sources through an IG, as depicted by Figure 5.1(b). The DSS is not shown in the figure for simplicity. Five clients were used for this experiment and clients choose a QSB randomly each time they submitted a query. All of the queries in Table 5.4 were used for this experiment. For each query we ran the experiment with 0, 3, 5 and 10 disconnections per client. For each of the disconnection values we ran the experiment with query recovery and then without query recovery.

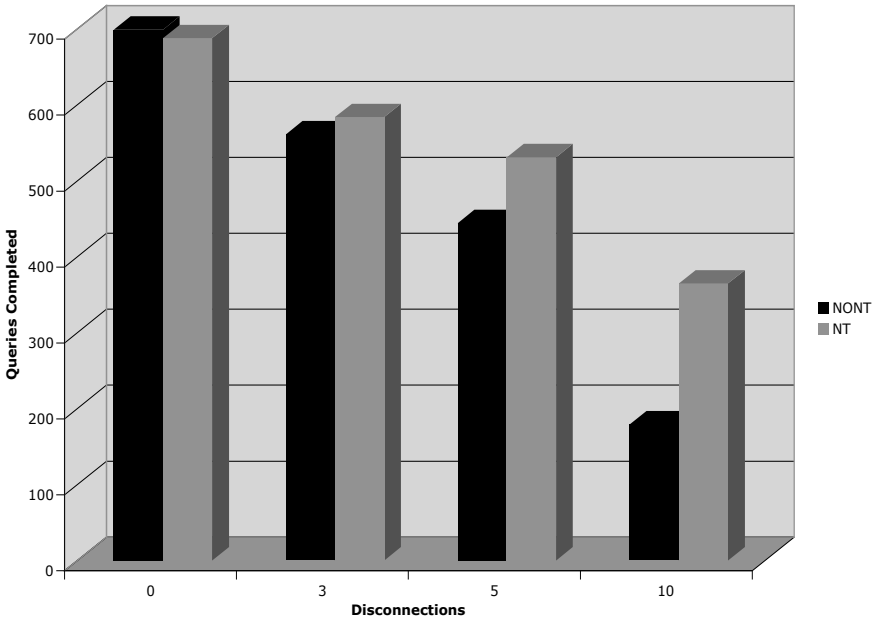



(a) Mesh topology without IGs (b) Mesh topology with IGs

Figure 5.1: NetTraveler federation with a mesh organization

#### 5.2.1.2 Results

Figure 5.2 shows the results for *Query 1* in both graphical chart and table representations. As can be seen in the chart of Figure 5.2(a), the throughput of the system with and without recovery, denoted by the grey (NT) and the black (NONT) bars respectively, is almost the same when the clients did not suffered any disconnection. However, as disconnections per client increased, the throughput of the system decreased. The decrease in throughput is more significant without recovery than with recovery. Without recovery, the throughput falls from an average of 34.95




(a) Throughput

| Disconnections | Scheme | Submitted | Completed | %Completed | Throughput |
|---|---|---|---|---|---|
| 0 | NONT | 705 | 699 | 99 | 34.95 |
| | NT | 694 | 688 | 99 | 34.4 |
| 3 | NONT | 575 | 561 | 98 | 28.05 |
| | NT | 590 | 584 | 99 | 29.2 |
| 5 | NONT | 523 | 444 | 85 | 22.2 |
| | NT | 536 | 531 | 99 | 26.55 |
| 10 | NONT | 511 | 179 | 35 | 8.95 |
| | NT | 370 | 365 | 98 | 18.25 |

(b) Comparison

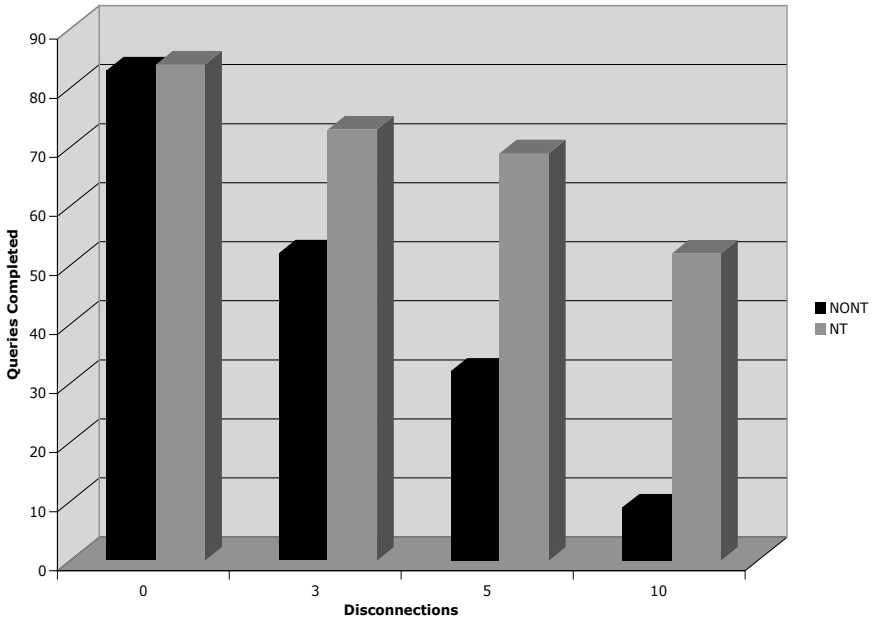
Figure 5.2: Throughput results for *Q1*

queries per minute with 0 disconnections to 8.95 queries per minute with 10 disconnections per client. With recovery, the throughput decreased from 34.4 queries per minute to 18.25 queries per minute. By using query recovery in the presence of 10 disconnections per client NetTraveler can complete 49 percent more queries than without query recovery for this first query.

Table 5.2(b) has the detailed results obtained for *Query 1*. The column named *%Completed* is the percentage of queries completed from the total of queries submitted for each of the different possible number of disconnections per client. As we can see, in the presence of disconnections the percentage of completed queries is always greater with query recovery, staying almost always above 90 percent. This percentage decreases from 99 percent with 0 disconnections to 35 percent with 10 disconnections without query recovery.

The results obtained when using *Query 2* in this experiment are presented in Figure 5.3. Basically, the results followed the same pattern as for *Query 1*. For 0 disconnections the system achieved a similar throughput with and without query recovery. However, the throughput of the system when disconnections existed was always greater with query recovery. For example, for 10 disconnections and without query recovery the system achieved a throughput of .45 queries per minute. With query recovery and the same number of disconnections the system achieved a throughput of 2.6 queries per minute. Also, the percentage of completed queries with query recovery was always greater, or at least equal, than the percentage of completed queries without query recovery.

For *Query 3* we also see a similar patter in the results as we did for *Query 1* and *Query 2*. The details of the results can be seen in Figure 5.4. We can see that, when disconnections were present, the throughput of the system for this query was greater when query recovery was used than without query recovery. Also, in the presence of disconnections, the percentage of completed queries was greater when query recovery was used. As expected, the amount of submitted and completed queries for all cases of this query is similar to the average found for *Query 2* since both queries take almost the same time to complete.

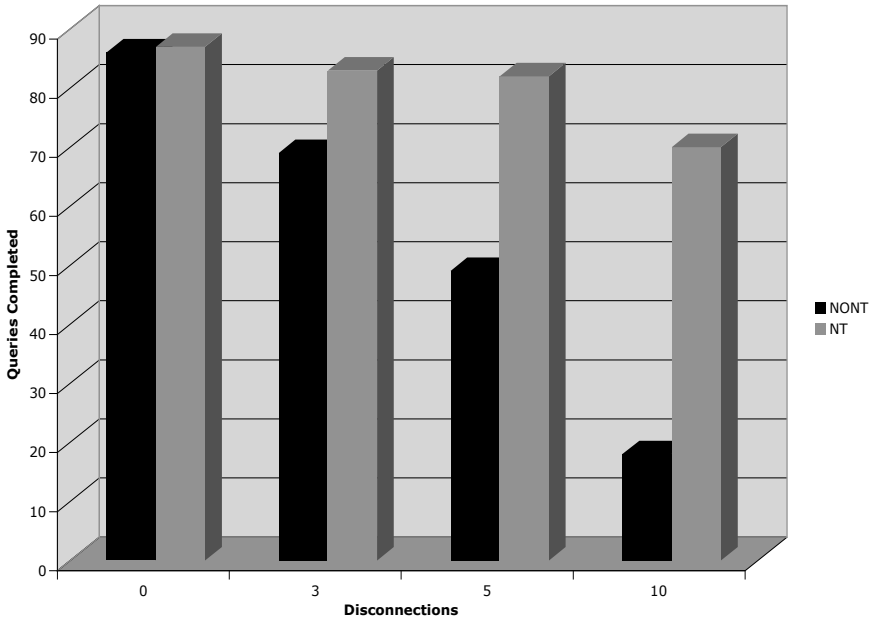



(a) Throughput

| Disconnections | Scheme | Submitted | Completed | %Completed | Throughput |
|---|---|---|---|---|---|
| 0 | NONT | 88 | 83 | 94 | 4.15 |
| | NT | 89 | 84 | 94 | 4.2 |
| 3 | NONT | 68 | 52 | 76 | 2.6 |
| | NT | 78 | 73 | 94 | 3.65 |
| 5 | NONT | 179 | 32 | 18 | 1.6 |
| | NT | 74 | 69 | 93 | 3.45 |
| 10 | NONT | 352 | 9 | 03 | .45 |
| | NT | 58 | 52 | 90 | 2.6 |

(b) Comparison

Figure 5.3: Throughput results for *Q2*




(a) Throughput

| Disconnections | Scheme | Submitted | Completed | %Completed | Throughput |
|---|---|---|---|---|---|
| 0 | NONT | 93 | 86 | 92 | 4.3 |
| | NT | 92 | 87 | 94 | 4.35 |
| 3 | NONT | 85 | 69 | 81 | 3.45 |
| | NT | 89 | 83 | 94 | 4.15 |
| 5 | NONT | 184 | 49 | 27 | 2.45 |
| | NT | 87 | 82 | 94 | 4.1 |
| 10 | NONT | 332 | 18 | 5 | .9 |
| | NT | 75 | 70 | 93 | 3.5 |

(b) Comparison

Figure 5.4: Throughput results for *Q3*

The results of this experiment show that NetTraveler is able to solve more queries per unit of time than a system that lacks a query recovery mechanism. The percentage of completed queries for all queries was always greater than 90 percent when recovery was "turned on". It was also noticed that in cases where clients suffered 5 or more disconnections, the system behaved erratically without query recovery. The issue was more noticeable for queries that took longer to complete, such as *Query 2* and *Query 3*. For *Query 1*, the issue appeared when each client suffered 10 disconnections, in which case the percentage of completed queries was 35. In both, *Query 2* and *Query 3*, the issue appeared when clients suffered more than 5 disconnections. The percentage of completed queries for *Query 2* and *Query 3* when more than 5 disconnections per client existed was always lower than 27 percent and in one case it was as low as 5 percent.

The reason for this erratical behavior was caused by the presence of multiple disconnections that overlapped each other when the system was already executing multiple queries. The system was not able to release the resources employed to execute queries that suffered disconnections in a timely manner to attend new queries. We can say that this is the saturation point of the system. From the saturation point on, queries that are submitted by clients are either rejected or are dropped in the middle of execution because of the lack of resources. The result is an extreme decrease in the percentage of completed queries as the result tables for all three queries in this experiment have shown. Nonetheless, in all cases that disconnections were present, and it was noticeable that the system did not reached a saturation point, the system was able to complete a greater percentage of queries using query recovery. Results obtained for *Query 1* with 5 disconnections, *Query 2* with 3 disconnections and *Query 3* with 3 disconnections are a prove of this.

### 5.2.2 Topology Experiment 1

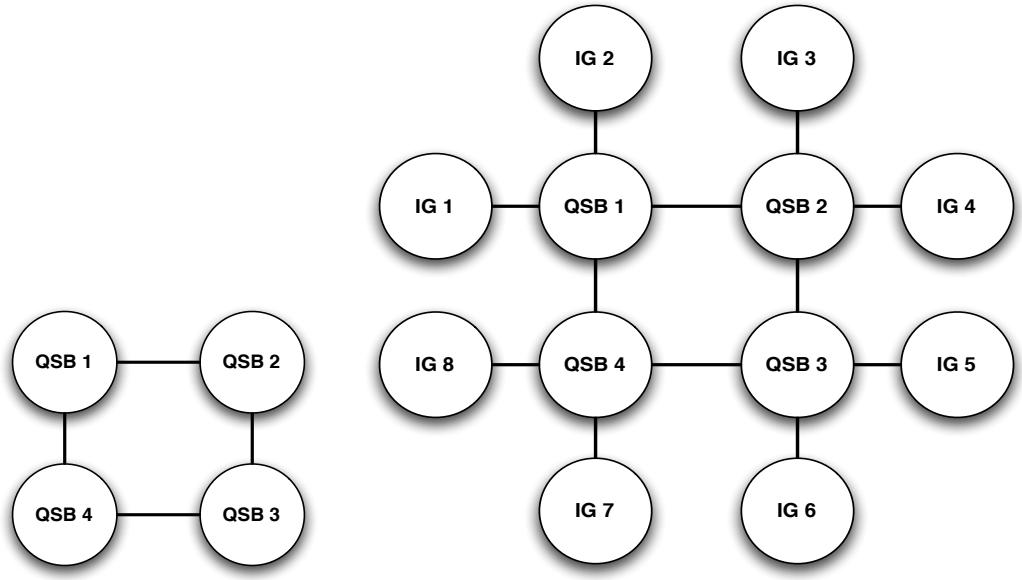
The purpose of this experiment was to test the performance of NetTraveler's P2P architecture under different topology organizations. The experiment was also used to compare the performance of a P2P topology organization using query recovery against a centralized topology organization also using query recovery.

#### 5.2.2.1 Methodology

For this experiment we organized the federation for the tests using 4 different topologies. These topologies were:

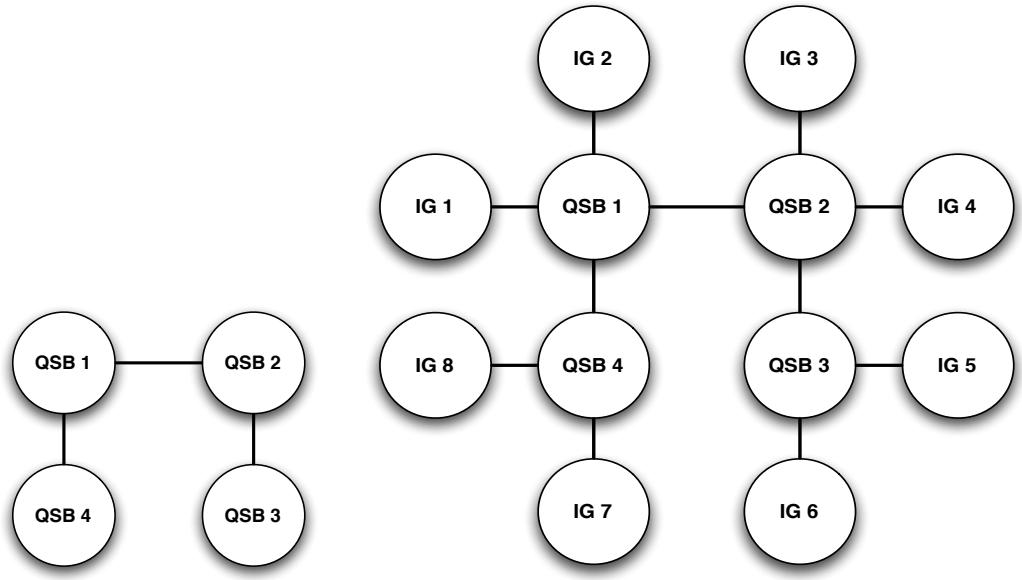
- Mesh - This is the same topology organization used for the throughput experiment presented in the previous section and is depicted in Figure 5.1.
- Ring - Figure 5.5 shows how the federation was organized for this topology. The actual ring is form by how the QSBs are organized. Figure 5.5(a) shows the QSBs forming a ring. The complete federation for this topology organization is shown in Figure 5.5(b). The DSS is not shown for simplicity.
- Tree - The federation organized as a tree is depicted in Figure 5.6. As before, the actual tree is formed by how the QSBs are organized. Figure 5.6(a) shows the QSBs in a tree topology. The complete federation, organized as a tree and not showing the DSS for simplicity, is depicted in Figure 5.6(b).
- Centralized - To organize the federation as a centralized system we removed 3 of the QSBs. A single QSB is responsible of receiving all the query requests and of accessing all the IGs. Figure 5.7 shows the federation organized in a centralized topology. The DSS is not shown for simplicity.

All the runs in this experiment were done with query recovery "turned on". The number of disconnections was fixed at 5 per client. It was decided that only 1 query was going to be used for this test and that the same query was going to be used across all topologies. *Query 2* was arbitrarily selected as the query to be used for this experiment. For each topology selected we repeated each run 3 times with 5 clients and then 3 times with 10 clients. Clients choose a QSB randomly each time they submitted a query.




(a) Ring topology without IGs (b) Ring topology with IGs

Figure 5.5: NetTraveler federation organized in a ring topology




(a) Tree topology without IGs (b) Tree topology with IGs

Figure 5.6: NetTraveler federation organized in a tree topology

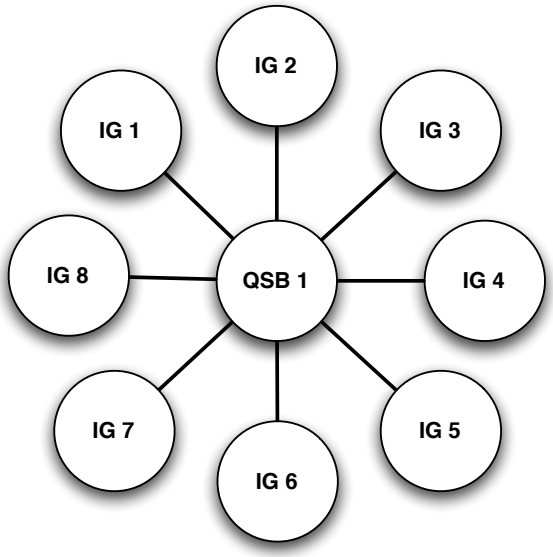


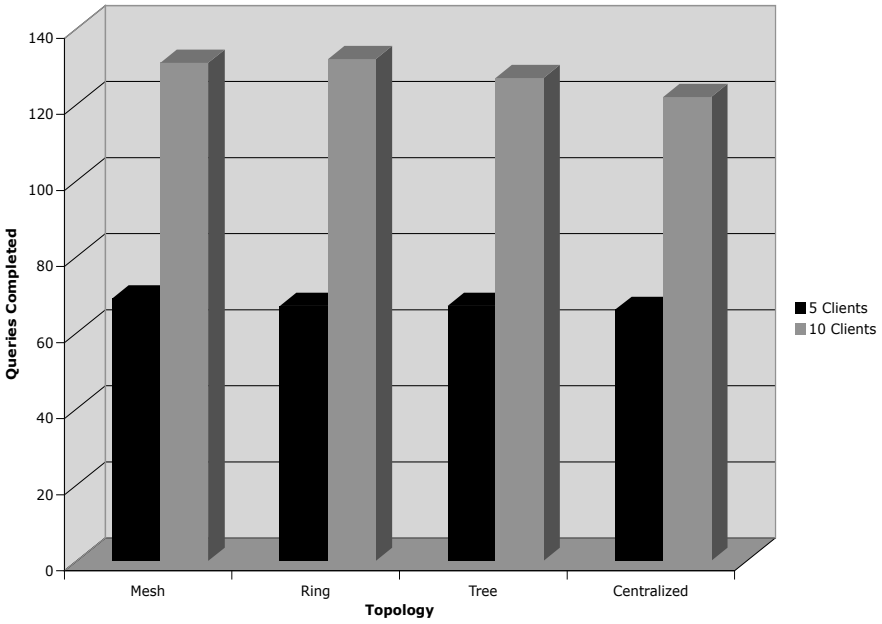

Figure 5.7: NetTraveler federation organized in a centralized topology

#### 5.2.2.2 Results

From the result chart and table for this experiment, shown in Figure 5.8, we see that all topologies performed almost equally when query recovery was "turned on". The throughput achieved by all topology organizations is almost equal when 5 or 10 clients submitted queries with 5 disconnections each. Even the centralized topology organization achieve a good level of performance in comparison with the other topology organizations. The throughput for all organizations with 5 clients was on average 3.36 queries per minute. With 10 clients the average throughput was 6.45 queries per minute. In all cases the percentage of completed queries was greater than 88 percent. The reason for these numbers is that with query recovery the system never reached its saturation point with the number of clients and disconnections per client that we selected for this experiment. This does not implies that the system lacks a saturation point. More research and work is needed to further study the impact of federation organization on throughput.

### 5.2.3 Topology Experiment 2

In this experiment we wanted to compare the performance of NetTraveler, in terms of achievable throughput, against that of a middleware system that is centralized and that lacks a




(a) Throughput

| Topology | Clients | Submitted | Completed | %Completed | Throughput |
|---|---|---|---|---|---|
| Mesh | 5 | 74 | 69 | 93 | 3.45 |
| | 10 | 141 | 131 | 93 | 6.55 |
| Ring | 5 | 72 | 67 | 93 | 3.35 |
| | 10 | 143 | 132 | 93 | 6.6 |
| Tree | 5 | 72 | 67 | 93 | 3.35 |
| | 10 | 143 | 127 | 88 | 6.35 |
| Centralized | 5 | 71 | 66 | 93 | 3.3 |
| | 10 | 132 | 127 | 92 | 6.35 |

(b) Comparison

Figure 5.8: Throughput results for experiment 2

query recovery mechanism. We part from the throughput results obtained in the second experiment (Section 5.2.2) for several topologies that used query recovery, including a system with a centralized architecture.

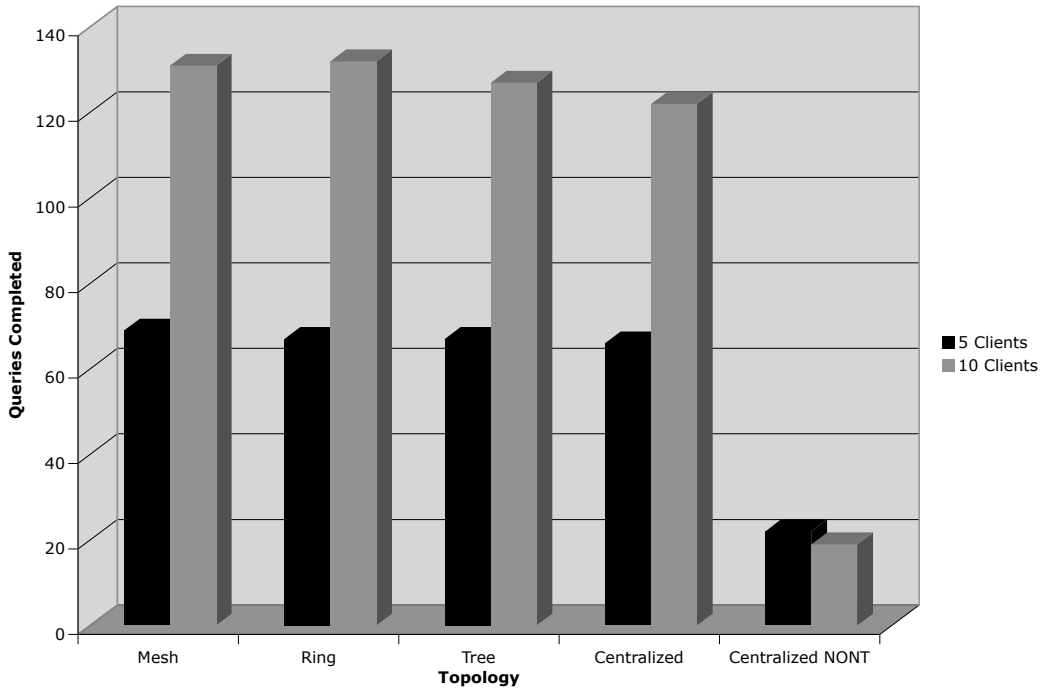
#### 5.2.3.1 Methodology

For this experiment the federation was organized using a centralized topology, exactly as the one used for the second experiment (Section 5.2.2), but with query recovery "turned off". The organization of the federation for this experiment is depicted in Figure 5.7. Disconnections were held constant at 5 per client and the number of clients was varied, starting with 5 and then 10 clients. Clients choose a QSB randomly each time they submitted a query. Clients submitted queries of type *Query 2* only in order to be consistent with the results obtained in the second experiment.

#### 5.2.3.2 Results

The results for this experiment are shown in Figure 5.9. This results also include the results that where obtained from the second experiment. We can see that the throughput and the percentage of completed queries for a centralized topology without query recovery, denoted by the bars labeled *Centralized NONT* in the $x$ axis of the graph in Figure 5.9(a), are much lower than for any of the other topologies tested in the second experiment, including a centralized topology with query recovery. When the centralized topology was used with query recovery the average throughput for 5 clients was 3.3 queries per minute. Without query recovery the same topology yielded a throughput of 1.1 queries per minute, 33 percent less queries per minute than with query recovery.

For 10 clients the throughput of the centralized topology with query recovery was .95 queries per minute, less than 1 query per minute. With query recovery, the throughput for the same topology was 6.35 queries per minute when 10 clients submitted queries. A throughput improvement of 668 percent. Looking at the percentage of completed queries in the *%Completed*




(a) Throughput

| Topology | Clients | Submitted | Completed | %Completed | Throughput |
|---|---|---|---|---|---|
| Mesh | 5 | 74 | 69 | 93 | 3.45 |
| | 10 | 141 | 131 | 93 | 6.55 |
| Ring | 5 | 72 | 67 | 93 | 3.35 |
| | 10 | 143 | 132 | 93 | 6.6 |
| Tree | 5 | 72 | 67 | 93 | 3.35 |
| | 10 | 143 | 127 | 88 | 6.35 |
| Centralized | 5 | 71 | 66 | 93 | 3.3 |
| | 10 | 132 | 127 | 92 | 6.35 |
| NONT Centralized | 5 | 143 | 22 | 15 | 1.1 |
| | 10 | 770 | 19 | 2 | .95 |

(b) Comparison

Figure 5.9: Throughput results for *Q3*

column of Table 5.9(b) we see that for a centralized topology without query recovery (*NONT Centralized*) the system reached the saturation point in the presence of 5 and 10 clients. When query recovery was used the system did not reached a saturation point in any of the topologies tested.

## 5.3 Summary of Experimental Results

The experiments carried out showed that NetTraveler was able to solve more queries per unite of time when using query recovery than without it. Also, it was found that when query recovery was not used the system quickly reached a saturation point. From the saturation point on the system behave erratically. The saturation point was not reached in the experiments where query recovery was used. Hence, the system appears more stable when query recovery is enabled in the presence of many client disconnections. However, this does not imply that there is not a saturation point with query recovery, only that it was not found through the experiments that were carried out. The results found after these experimental tests to the first implementation of NetTraveler are encouraging. They are a motivation to continue with the implementation of the NetTraveler system and also to further investigate the capabilities of NetTraveler's connectionless query execution engine through experimentation.

# CHAPTER 6

# Conclusion and Future Work

With the broad acceptance of mobile devices in our modern society, companies have been led to put more effort on the development of more powerful and accessible mobile devices than ever. This new breed of devices, ranging from laptops and Tablets PC to PDAs and smart cellphones, are designed with Internet connectivity in mind. With the ever increasing affordability of broadband networks, we can expect these type of devices to be playing a more active role as clients of Database Middleware Systems.

Most of the existing Database Middleware Systems treat all query request with the same policies, regardless of the nature of the computing device that is holding the client application. However, mobile devices have power limitations, intermittent connectivity and may get connected through pricey Internet links. A network failure, the lost of power or a software crash are all conditions that can cause a mobile device to lose network connectivity, disrupting the client-side execution of a query. In current middleware solutions, the query being processed before the disconnection is aborted and the query must be re-computed. This leads to slow response times as the query needs to be re-started from scratch. More importantly, the resources invested by both the client and the server in processing the query are lost.

This thesis has presented NetTraveler, a Database Middleware System for Wide Area Networks designed to overcome these limitations. WANs are modeled in NetTraveler as a collection

of applications, each having a specific role in helping a client to solve a given query. The elements in NetTraveler are logically organized into groups of cooperative applications known as *ad-hoc federations*. Federations are ad-hoc because they can be formed or dissolved over time, based on the decisions taken by its members. A federation can spawn one or more Local Area Networks (LAN), and a LAN can have elements that belong to more than one federation.

NetTraveler relies on a connectionless query execution engine to execute queries submitted by clients. This execution engine allowed the implementation of a query recovery mechanism where queries that are interrupted due to a failure in a client can be resumed from the point of the failure, and without having to restart the query from scratch. NetTraveler also exhibits Peer-to-Peer functionality to prevent a centralized operational model, in which a central broker needs to know all data sources and becomes a focal point through which all queries must pass.

In the rest of this chapter we shall see a summary of the major contributions of this thesis and also a discussion of the direction in which work concerning NetTraveler shall be directed in the near future.

## 6.1 Summary of Contributions

In Chapter 3, we saw the NetTraveler Middleware System from an architectural point of view. The chapter began by presenting how Wide Area Networks (WANs) are modeled in NetTraveler and then introduced each of the server components in the framework. In order to achieve the goals of this thesis, it was necessary to identify the minimum set of components needed for query execution to take place. These components, which are called the *Query Services*, are:

- *Query Service Broker* (QSB) - Takes on the responsibility of finding the computational resources (data, disk access, CPU time, network time, etc.) required to extract from the data sources the data to answer the queries posed by clients. QSBs perform query tasks such as *query parsing*, *query optimization* and *query execution*. QSBs exhibit Peer-to-Peer behavior since a broker might contact other brokers in a federation to help it solve a given query.

- *Information Gateway* (IG) - Has the role of providing access to the wealth of information contained in the data sources. IGs extract data from the data sources and can execute query operators, particularly those that can filter out unwanted results, such as predicates.
- *Data Synchronization Server* (DSS) - Helps clients in caching query results and in obtaining extra disk space. More importantly, a DSS can become a **proxy** for a client, gathering the results intended for the client if it goes offline or experiences some type of failure that causes it to lose network connectivity.

We also saw an overview of query execution in Chapter 3. The model of query execution that has been used in middleware systems is the *connection-oriented* model. In NetTraveler, a *connectionless oriented* query execution model is used. We saw that for mobile clients in a Wide Area Environment, a connectionless oriented query execution engine provides several benefits over a connection oriented engine. For one part, the execution of queries that must access several data sources is greatly improved because the execution of query operators in parallel is maximized. Also, a disconnection failure will not necessarily imply the abortion of a query, since the parties can reconnect and attempt to reestablish the computation of a query.

In Chapter 4, we saw the implementation details of the NetTraveler components that were identified as indispensable to achieve the goals of this thesis. We saw that the QSB, IG and DSS achieve the completion of their corresponding tasks by following a common iterator-like approach of execution. The methods defined by each component are:

- *execute()* - Used to submit a query request to a QSB, IG or DSS.
- *next()* - Used to retrieve the results of a query that is being executed by a QSB or IG. This method is also used by clients to get results from a DSS.

The QSB defines an extra method named *reroute()*. A call to this method will cause the client-side execution of a query to be rerouted to a DSS. We saw that the Query Services rely on a multithreaded scheme that uses two entities, known as the *Query Coordinator* and the *Query Worker*, to actually execute the part of the query that corresponds to them.

Chapter 4 also detailed how recovery of the client-side work of a query is performed in NetTraveler. Recovery of the client-side execution of a query is achieved by *rerouting* the client-side execution of a query to a DSS. There are two possible scenarios for client-side query recovery in NetTraveler: (1) *automatic recovery* and (2) *explicit recovery*. Automatic recovery occurs when a QSB finds a query that has been idle beyond a threshold time value $t_i$. This time value $t_i$ is defined as the maximum amount of time that a client expects to wait between successive *next()* calls for a query. The QSB contains a *monitor thread* that periodically monitors all queries being executed. For each query that is found idle, the monitor thread in the QSB coordinates the rerouting of the client-side execution of the query. In explicit rerouting, the client application will determine that it must abandon the execution of the current query, based on some criteria specified by the user, and will trigger an event that will issue a *reroute()* call on the QSB. With rerouting of the client-side execution of a query, system and client resources are not wasted since queries that suffer a client-side disconnection are never re-started.

In Chapter 5, we discussed the experiments that were carried out in order to validate the ideas presented in this thesis about a connectionless query execution engine and query recovery. The experiments showed that NetTraveler achieved a greater throughput, defined as the number of completed queries per unite of time, when using query recovery than without it. Also, it was found that when query recovery was not used the system quickly reached a saturation point. From this saturation point on the system behaves erratically. The saturation point was not reached in the experiments where query recovery was used. Hence, the system appears more stable when query recovery is enabled in the presence of many client disconnections. However, this does not imply that there is not a saturation point with query recovery, only that it was not found through the experiments that were carried out. Hence, the saturation point might be reached with a far larger set of clients.

## 6.2 Future Work

This section gives directions for future work concerning the NetTraveler middleware system:

- Implementation of the *Registration Server* (RS) and the *Data Processing Server* (DPS). The RS is needed to actually allow dynamic *ad-hoc* formation of federations, dissolution of federations and metadata dissemination. The DPS component is needed to test the possible advantages of using a dedicated server when performing complex computations in queries.
- Native support for other type of query languages such as XML Query Language.
- Implementation of a full-fledged query optimizer. Research is being carried towards developing a query optimizer suited for a system like NetTraveler. This query optimizer will use a randomized optimization model to build a query plan that has the unique characteristic of being dynamic, in the sense that it can be changed in the middle of execution. In this manner, the optimization process ends when the query completes execution.
- Design and implementation of *server-side* query recovery based on dynamic query plans. With the ability to reroute the execution of any node in the query plan, NetTraveler will be able to recover from failures suffered by server applications. A node in the plan can also be rerouted for any other reason, such as specific rules set for each operation. NetTraveler will be the first system with the novel approach of dynamic query plans.
- Implementation of the DSS query execution engine. This will allow the DSS to work on behalf of QBSs and IGs. This will be helpful for the implementation of server-side query recovery and of dynamic query plans in general.

# BIBLIOGRAPHY


[1] Swarup Acharya, Rafael Alonso, Michael J. Franklin, and Stanley B. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *SIGMOD Conference*, pages 199–210, 1995.

[2] Daniel Barbará and Tomasz Imielinski. Sleepers and workaholics: Caching strategies in mobile environments. In *SIGMOD Conference*, pages 1–12, 1994.

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

[4] 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.

[5] 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.

[6] Expedia.com. http://www.expedia.com/.

[7] 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.

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

[9] 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.

[10] ODBC Basics. http://www.datadirect.com/developer/odbc/basics/index.ssp.

[11] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliff, New Jersey, 1991.

[12] Alexios Delis and Nick Roussopoulos. Performance and Scalability of Client-Server Database Architectures. In *Proc. 18th VLDB Conference*, pages 610–623, Vancouver, British Columbia, Canada, 1992.

[13] M. Astrahan and et.al. System R: Relational Approach to Database Management. *ACM Transactions of Database Systems*, 1(2):97–137, 1976.

[14] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R.A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM SIGMOD Conference*, pages 23–34, Boston, Massachusetts, USA, 1979.

[15] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, December 2000.

[16] M.J. Franklin, B.T. Jónsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *ACM SIGMOD*, pages 149–160, Montreal, Canada, 1996.

[17] P.A. Bernstein and D.W. Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, 28(1), January 1981.

[18] Hyunchul Kand and Nick Roussopoulos. Using 2-way Semijoins in Distributed Query Processing. Technical Report CSTR-1681, University of Maryland, 1986.

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

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

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

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

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

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

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

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

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

[29] Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt. Processing Queries in a Large Peer-to-Peer System. In *CAiSE*, pages 273–288, 2003.

[30] 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.

[31] E. Pagán, M. Rodríguez-Martínez, et. al. Registration and Discovery of Services in the NetTraveler Integration System for Mobile Devices. In *ITCC (2)*, pages 275–281, 2004.

[32] M. Rodríguez-Martínez. NetTraveler Project Description. 2005.

[33] Sun Microsystems. JDBC Technology. http://java.sun.com/products/jdbc/.

[34] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1987.

[35] Apache Software Foundation. Web Services - Axis. http://ws.apache.org/axis/index.html.

[36] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[37] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[38] Sun Microsystems. What is an Interface? http://java.sun.com/docs/books/tutorial/java/concepts/interface.h

[39] E. Christensen, F. Curbera, Greg Meredith and S. Weerawarana. Web Services Definition Language. http://www.w3.org/TR/wsdl, March 2001.