Design and Implementation of the NetTraveler Middleware System based on Web Services

Elliot A. Vargas-Figueroa,
Electrical and Computer Engineering Dept.
University of Puerto Rico, Mayagüez
Elliot.Vargas@ece.uprm.edu

Manuel Rodriguez-Martinez
Electrical and Computer Engineering Dept.
University of Puerto Rico, Mayagüez
manuel@ece.uprm.edu

Abstract

We present NetTraveler, a database middleware system for WANs that is designed to efficiently run queries over sites that are either mobile clients or enterprise servers. NetTraveler is designed to cope with dynamic WANs where data sources go off-line, change location, have limited power capabilities, and form ad-hoc federations of sites. We present how NetTraveler can continue running a query when the client posing the query leaves, and then delivers the query results when the client returns. We present an implementation experience of NetTraveler using Web services. We validate our design and implementation choices with a preliminary performance study of NetTraveler. 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 than solutions based on previous approaches.

1. Introduction

Mobile devices such as notebooks, PDAs, cell phones, and Tablet PCs are among the most popular devices available on the market today. These devices are Internet-ready and have enough capabilities to run popular applications. 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. The emergence of affordable broadband (e.g. DSL) and wireless networks (e.g. IEEE 802.11b) are rapidly making this once utopian dream a tangible reality.

We can expect mobile computing devices to have a dual role for their users. First, they will continue to act as the client sites that run the applications used to obtain content from the Internet. Second, they will become valuable sources of information that users need to carry out with them as they move across geographic locations. We will find data sources containing very diverse data sets such as mp3 files, lists of appointments, phone directories, sensor readings, relational data, and XML files, among others. Therefore, it will be necessary to build distributed systems to support seamless data access to these sources. 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). 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.

In this paper we present 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 hosts for query processing purposes. NetTraveler is designed to cope with dynamic WANs where 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.

1.1. Contributions of this work

The main contributions that we make are:

- We present the NetTraveler Architecture and argue about its benefits for supporting query processing in WANs. We show how NetTraveler can continue running a query when the client leaves, and then delivers the query results when the client returns.
- We present an implementation experience of NetTraveler using Web services. To the best of our knowledge, no other query execution engine has been implemented in this fashion.
- We start validating our design and implementation choices with a preliminary performance study. 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 than solutions based on previous approaches.

The remainder of this paper is organized as follows. Section 2 presents the architecture of NetTraveler. Section 3 discusses the issues involved in implementing NetTraveler with Web services. Section 4 contains an overview of the query execution framework of NetTraveler based on Web services. In section 5 we present a preliminary performance evaluation of our NetTraveler prototype. Related work is presented in section 6. Finally, section 7 contains the summary and conclusions of this work.

2 NetTraveler Architecture

We shall start with an overview of the architecture of the NetTraveler middleware system. We model the WANs as having a collection of applications $H = \{h_1, h_2, ..., h_n\}$, each having a specific role in helping the user 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 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 client capabilities to submit queries. These capabilities come from running a given interface 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

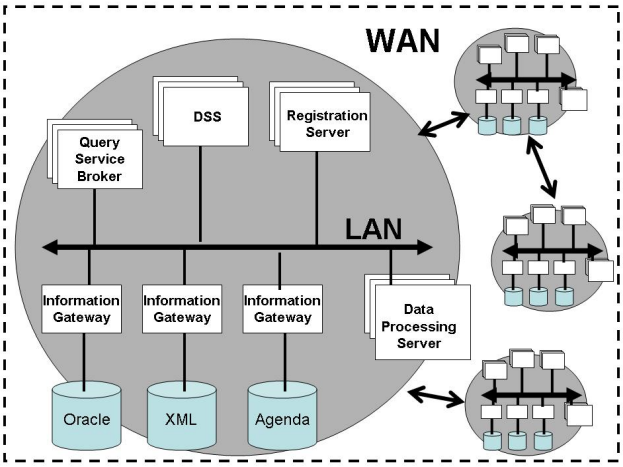

Figure 1. NetTraveler Architecture

application. When a client $c \in C$ wants to pose a query to sources in S, it needs to contact a server application known as a *Query Service Broker* (*QSB*). A collection of *QSB*s $B \subset H, B = \{b_1, b_2, ..., b_k\}, k < n$ take on the responsibility of finding the computational resources required to extract data from the target sources in S to answer the query posed by client $c \in C$. *QSB*s exhibit Peer-to-Peer (P2P) behavior. 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 *QSB*s in B can access the data sources in S by means of a server application known as the *Information Gateway* (*IG*). A collection of *IG*s, $G \subset H, G = \{g_1, g_2, ..., g_m\}, m < n$ have the role of providing brokers with access to the information contained in the data sources in S. It is at this level of the *IG*s that data extraction occurs, as well as schema mapping. *IG*s can execute query operators, particularly those that can filter out unwanted results, such as predicates and aggregate operators. Clearly, metadata is needed for the brokers to be able to find the required data sources. This metadata must be spread throughout the system to advertise the availability of resources. The responsibility of the metadata dissemination is given to a type of server known as the *Registration Server* (*RS*). The collection of *RS*s, $R \subset H, R = \{r_1, r_2, ..., r_p\}, p < n$ deal with the problem of advertising metadata, encoded in XML, and of 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 *RS*s work as peers to exchange the metadata, just as network routers advertise routes to each to other to enable fu-

ture packet forwarding decisions.

The last two groups of elements in our framework are the *Data Synchronization Server* (*DSS*), and the *Data Processing Server* (*DPS*). A collection of *DSS*s $D \subset H, D = \{d_1, d_2, ..., d_t\}, t < n$ groups server applications that help clients in caching query results, obtaining extra disk space, and keeping synchronized copies of the data natively stored by a client which also happens to behave as a data source occasionally. More importantly, a *DSS* $d \in D$ can be 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. We shall see more of this matter in section 4.5. Finally, a collection of *DPS*s, $P \subset H, P = \{\rho_1, \rho_2, ..., \rho_v\}$ contains the server applications that provide a commodity service for computational tasks during query processing. These tasks include query plan execution, sorting, or other type of specific computational operation.

The elements in our model 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 their 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 *IG*s, one or more clients, one *RS*, one *DSS*, and one *DPS*.

In NetTraveler, a local group provides the minimal infrastructure to execute a query. 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}. The RS in each local group also 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 cluster can be constructed in this fashion.

3. Prototyping NetTraveler

We begin the discussion of the NetTraveler prototype detailing the expected clients and how the system supports them. Then, we detail the implementation of three of the applications presented in section 2. The applications are the *QSB*, the *IG* and the *DSS*. These components were implemented first since they provide the minimum infrastructure needed to conduct the firsts tests to our system. The applications were developed as **Java Web Services** using the Apache Axis Soap Toolkit. Henceforth, we refer to them as NetTraveler Services. NetTraveler Services communicate by exchanging Soap messages based on RPC calls with support for Soap messages with attachments.

3.1. Clients

We expect to have clients running on full-fledged machines with constant power and connectivity, such as workstations, and also have clients running on limited-connectivity and/or low-powered devices. Depending on the specific condition of a client on a given time it can submit a query of four possible classes: 1) power constrained, 2) network constrained, 3) intermittent connectivity, and 4) unrestricted. *Power constrained* deals with devices for which the system must conserve energy by delivering data as quickly as possible. *Network constrained* deals with devices for which the network is expensive to use. *Intermittent connectivity* constrain deals with devices that are either moving, or plan to be on-and-off in the near future. In this case, the system must be ready to suspend and re-start a query. Finally, *unrestricted* queries capture those queries coming from workstations, desktops and other servers.

Clients in NetTraveler do not directly specify the type of query that they are submitting. Rather, the system determines this by virtue of a *profile* object that the client must send when submitting a query. In NetTraveler, we consider a query request Q as a pair $Q = (q, p)$, in which q is a query expressed in SQL, and p is a vector representing the profile of a client c. That is, $p = (m_1, m_2, ..., m_n)$ is a n-tuple that represents n characteristics of the device that poses the query q. Each characteristic $m_i \in p$ represents a property of the device that must be taken into consideration at the moment of both query optimization and query execution.

3.2. Query Services Broker (QSB)

Figure 2 depicts the current components of the *QSB*. The *Endpoint* is a Service Endpoint Interface (SEI), as defined by JAX-RPC, where the methods that can be invoked by clients and peer *QSB*s are defined. The *QSB* defines three types of requests through the *Endpoint*: *execute*, *next* and *reroute*. Execute requests are issued by clients and peers to pose a query. Next requests are used to retrieve the results of a query. Reroute requests are submitted by client applications that wish to hand the client side query execution to a *DSS* that will work on their behalf.

The *Endpoint* receives clients' and peers' requests and routes them to the *Execution Manager*. The *Execution Manager* is in charge of coordinating the execution of each request by invoking the appropriate underlying components. For example, when a client submits a query by issuing an *execute* request, the *Execution Manager* will invoke the *SQLParser* to obtain a tree representation of the query. This

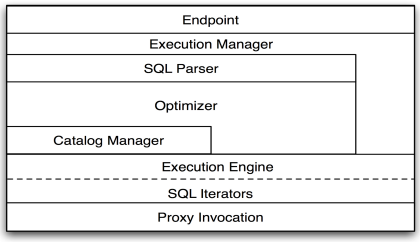

Figure 2. QSB Internal Organization

tree is then handed, along with the client *profile*, to the *Optimizer*. The *Optimizer* in turn will return an annotated query plan to the *Execution Manager*. Annotations specify the site at which every operator in the plan must be executed. The *Execution Manager* will then begin query execution by driving the *Execution Engine*. During query execution, the *QSB* will store the client profile, the query id and other parameters related to the query as state information in order to allow *next* and *re-route* requests to be issued.

The *QSB*'s *Execution Engine* is based on the iterator model [6]. There are iterators for performing local and remote operators.When a query is being executed, the *Execution Manager* will drive the engine to produce result tuples independently of whether results are being solicited or not. The results produced and the current state of the engine are stored, halting query execution, until another event triggers the production of more tuples. Notice that query execution has not been stopped, the query is still being executed but will be in a halted state because no one is consuming the output stream. Due to this fact, the *QSB*'s execution engine is said to be connectionless and asynchronous.

3.3. Information Gateway (IG)

The *IG* resides in the same site of the data source and provides the *QSB* with an uniform representation of a remote data source. The organization of the *IG* is depicted in figure 3. The *IG Endpoint* is analogous to the *QSB Endpoint* and defines two request: *execute* and *next*. The *execute* method receives the part of the query plan that the *IG* must execute. The *next* method returns the results of an evaluated plan. Only the *QSB* can invoke these methods.

Analogous to the *QSB*, all request received at the *Endpoint* are routed to the *Execution Manager* who is in charge of managing query processing. The *Execution Manager* of figure 3 is implemented in the same manner as the *Execution Manager* of the *QSB*. The *IG* also contains an iterator-based execution engine for the execution of query operators. The engine is asynchronous and connectionless.

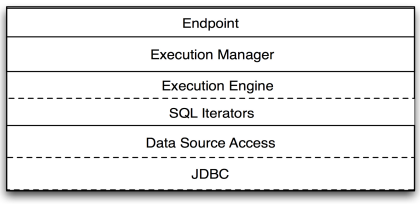

Figure 3. IG Internal Organization

3.4. Data Synchronization Server (DSS)

The current organization of the internal components of the *DSS* is depicted in figure 4. The *DSS* receives two types of requests : *fetch* and *next*. *Fetch* requests are issued by the *QSB* to ask the *DSS* to work as a proxy for a client. *Next* requests are issued by clients to retrieve tuples of a query that the *DSS* has executed on their behalf. All work within the *DSS* is coordinated by the *Execution Manager* who receives requests that are routed from the *Endpoint*.

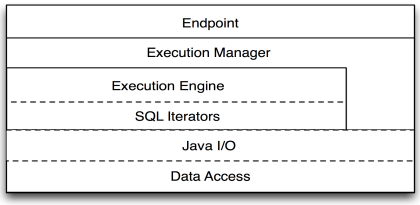

Figure 4. DSS Internal Organization

The *DSS* has its own *Execution Engine* that is also asynchronous and connectionless. The decision of implementing an execution engine in the *DSS* arose from the fact that the *DSS* can also be a proxy for another service. The *Data Access* layer provides the mechanisms needed to access the underlying data source.

4. Query Execution Framework

The execution framework is composed by the combined participation of several NetTraveler Services. Under normal query execution, the framework will be composed by one or more *QSB*s and *IG*s. However, as will be seen in section 4.5, a *DSS* can also participate in query execution.

4.1. Client Query Execution

Clients submit query requests and obtain results using an iterator-like interface. Each query request $Q = (q, p)$ (section 3.1) submitted also contains an idle time t_i and the address of a *DSS* d. The time t_i is the maximum amount of time that a client expects to be without consuming results.

This threshold value t_i is used by the *QSB* to determine if the client side execution must be rerouted to *DSS* d.

4.2. QSB Query Execution

The *QSB* b will first identify the set of relations $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ that are needed to solve a received query request Q. *QSB* b will then identify the *IG*s in its local group that have access to the data sources holding these relations. If a member of $\mathcal{T}$ is not directly referenced by an *IG* that *QSB* b knows, it will contact one or more peer *QSB*s that can help it solve the query. The contacted peers will either know an *IG* with access to one of the relations in $\mathcal{T}$ or know another peer *QSB* with access to one relation in $\mathcal{T}$.

After the parse and optimize stages, *QSB* b will build a working plan for the query. The first *QSB* b will send the part of the plan that each of the *IG*s and peer *QSB*s will execute. Then, the operators that *QSB* b must execute are converted into iterators, starting with remote operators and building the working plan in a bottom-up fashion. Query execution is started asynchronously using the top-most iterator of the execution plan. The tuples obtained are always cached before been served to the client.

4.3. IG Query Execution

An *IG* g will receive a query sub-plan from the *QSB* b with annotations of the operators that it must execute. This plan must be converted into a working plan before execution can take place. An *IG* g will convert the query operators that it must execute into iterators following the same bottom-up approach used by *QSB* b in the previous section. The bottom iterator will always be the one accessing the data source. Other iterators are build on top of this one until the top-most iterator is generated. *IG* g will store tuples from the disk in a cache from were they are served to the *QSB* b. The *IG* g also execute the working plans asynchronously.

4.4. DSS Query Execution

When *DSS* d receives from *QSB* b a request to *fetch* results tuples on behalf of a client, it will receive from *QSB* b the client profile p along with the query id qid and the *QSB* b URL. The *DSS* d will then generate a *proxy* and will start issuing *next* request on the *QSB* b. *DSS* d will continuously issue *next* requests until all tuples are fetched. When a client c contacts the *DSS* d, it will passed to it its profile p and the query id qid of the query that the *DSS* finished on behalf of client c. *DSS* d will verify if the query exists. If the query exists in the *DSS* d and the client is the owner, *DSS* d will serve tuples to the client.

4.5. Recovery of Query Work

In this section we shall discuss the mechanisms to recover query work when a query is aborted due to some failure. Such failure can be a network failure, power lost, or other type of condition that causes the client device to lose connectivity. In current middleware solutions, the query being processed is aborted, and the user needs to re-start the query. This can lead to slow response times as the query need to be started from scratch. Additionally, the resources invested in processing the query are lost. To give an example, consider a client that sends the following query: "Get the names of all drivers that have received a fine of over $100 on road PR-2". In this case, the client sends the query to the QSB, and the QSB access three different databases to get the data. Suppose that the expected query result is about 1MB. If a failure occurs after the client has recived some data, say 500KB, the query is aborted and the client needs to re-submit the query. Hence, the user would need to receive the results from the beginning, downloading the first 500KB again. Thus, the client downloaded 1.5MB of data, since the failure made some of the work to be redone.

There are two possible scenarios for client side query rerouting. The first one is automatic discovery of a query that has been idle beyond the threshold time value t_i (section 4.1). The *QSB* b that receives the query request Q has a thread that monitors periodically all queries that are actively running. If the *QSB* b finds a query that has been idle beyond the threshold time t_i it will automatically generate a *proxy* for the client's *DSS* and issue a *fetch* request.

The second possible scenario for re-route is that the client application running in client c explicitly issues a *reroute* request on *QSB* b. In this case, the client application running on client c determines that it must abandon the current query based on some criteria specified by the client. It may be that the client c specified a maximum amount of time to wait for results to arrive or that the device is running low on battery. *QSB* b will contact the client's *DSS* in the same manner as in automatic re-routing.

Once the client's *DSS* has been contacted, the query continues normal execution as described in section 4.2 since *next* requests issued by the *DSS* are seen by the *QSB* b as a *next* request issued by a client. Once the client c returns, it will contact its *DSS* and will fetch the remaining tuples of the query, completing query execution from the point of disconnection and without restarting the query.

5. Performance Evaluation

We have begun validating our ideas of query recovery and the advantages of a P2P architecture with the current prototype of NetTraveler. We prepared a set of experiments that helped us to start gaining an insight of the nature of the

system. Our first goal with these experiments was to test if a middleware system with the capability of recovering the query work of a client after a disconnection was capable of achieving a greater *throughput* than a system without this capability. We use the following definition for *throughput*: *the number of queries solved by the system per unit of time*. Second, we wanted to start gaining experience with the advantages of developing NetTraveler as a full P2P middleware system based on the known bottlenecks problems that plagued centralized systems.

5.1. Throughput Evaluation

We setup a configuration of three *QSB*s and six *IG*s for this experiment. Every *QSB* knew two *IG*s and no *IG* was shared between *QSB*s. All three *QSB*s where peers of one another. There were three types of queries that the client could submit. The first query returned 64 KB worth of data, the second query returned 300 KB worth of data and the third query returned 1.75 MB worth of data. We had 5 clients that ran for 20 minutes, selecting queries randomly and submitting them to one of the *QSB*s. Clients selected a *QSB* to which they submitted the query in a round robin fashion. We completed three runs simulating disconnection failures. We started with a number of disconnections per client of zero, followed by three disconnection per client and finished with five disconnection per client. Clients distributed the errors uniformly in time.

Each disconnection error lasted between one and two minutes and the threshold time t_i of the clients was set to 30 seconds. The *QSB* checked for idle queries every minute. Each run was performed with query recovery turned on and repeated with query recovery turned off. Figure 5 presents the results for this experiment. Here, NONT represents the system without query recovery and NT represents the system with query recovery. There is practically no noticeable difference in the system when there are no failures. As expected, when the number of errors increased the throughput of the system decreased. However, when the system employed the query recovery mechanism, it turned out that the throughput was always greater in the presence of failures. For three disconnections per client, the query recovery mechanism gave us 23% more queries solved than without query recovery. For five disconnections per client NetTraveler was able to solve 121% more queries in the same time. These results confirm our original idea and are a motivation to investigate the subject of query recovery more deeply.

5.2. Architecture Evaluation

The goal of this experiment was to determine if the P2P architecture of NetTraveler and its ability to recover query work is better than a system with a centralized architecture

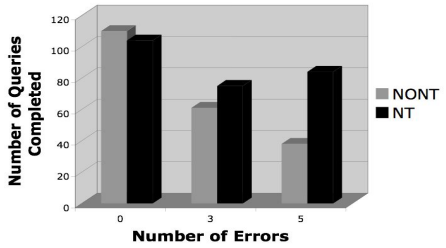

Figure 5. Throughput Evaluation

and without query recovery. To test the P2P architecture we setup the system exactly as described in section 5.1. For the centralized scheme we used only one *QSB* that communicated with all six *IG*s and with query recovery turned off. We used the same queries of the previous experiment and increased the number of clients from five to 15. Clients submitted queries over a period of 20 minutes as describe in section 5.1. We performed a run with zero disconnections per client over both setups and another run with five disconnections per client. Results are shown in figure 6. As can be seen, the P2P architecture of NetTraveler achieve a greater throughput on both cases (with zero disconnections and with five disconnections) than the centralized system. With zero disconnections NetTraveler solved 31% more queries and with five disconnections it achieved 40% more queries.

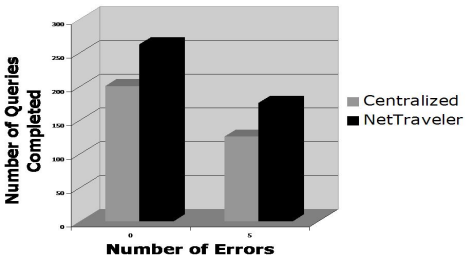

Figure 6. Architecture Evaluation

6. Related Work

Database Middleware Systems [3, 4, 5] have been used as a solution to integrate heterogeneous data sources and support applications that require integrated access to their data. The term Federated System is used to depict a group of data sources integrated via database middleware. Database middleware arise as an alternative to Distributed Database Engines, such as R*, which required existing data sources to be purged and their data re-ingested into a Distributed DBMS common to all sites. Existing Database middleware solutions have an architecture based on a central integration server that provide client applications with a

uniform view of the data, and a single-point of access to the federated sites. The integration server relies on the capabilities of translators to extract the data from the sources, and perform schema mapping operations to convert data from local schemas into a global schema specified by the client to the integration server. Once the data items have been translated, they are sent back to the integration server for further processing. Most of the query processing occurs at the integration server site and the data sources often act as mere I/O nodes. A catalog associated with the integration server provides the metadata necessary to find data sources, schema mapping rules, and query processing strategies.

Two approaches dominate the spectrum of possible database middleware implementation schemes. The first approach is to use a relational DBMS, such as Oracle, as the integration server, and use database gateways as the translators. In the second approach, a Mediator System specifically tailored for distributed processing is employed.

Distributed join processing, transaction processing, data consistency and synchronization, and data caching have been the major focus of research [9, 10, 11] on both Database Middleware and Distributed DBMS systems. More recent work [12, 13, 14] has concentrated on the problem of adapting query plans to system changes during query execution Data Streams have arise in this context. However, the basic assumption in all cases has been that server applications run on enterprise servers.

Data Dissemination Systems, such as SIFT [15], are used to deliver frequently accessed data items to a large population of client sites. In these systems, there are a few data sources that are the producers of information. These sources could be database servers, Web servers, or stock tracking applications, among others. The clients are numerous and diverse, ranging from workstations all the way cell phones. The fundamental assumption in these systems is that clients are "couch potatoes", only interested in getting similar information day after day. These systems are often called publish-subscribe systems since clients must subscribe with the data source to get the information being published.

7. Summary and Conclusions

In this paper we have presented the NetTraveler Architecture and argued about its benefits for supporting query processing in WANs. The benefits include Peer-to-Peer (P2P) organization, asynchronous execution of queries, and the ability to stop and resume queries when data sources or clients leave the system. We have presented how NetTraveler can continue running a query when the client posing the query leaves, and then delivers the query results when the client returns to the system. We have presented an implementation experience of NetTraveler using Web services. To the best of our knowledge, no other middleware query execution engine has been implemented in this fashion. We have begun validating our design and implementation choices with a preliminary performance study of the system. This study shows that NetTraveler can complete more queries per unit of time than middleware system that cannot gracefully recover from contingencies during query execution caused by clients leaving the system. This study also show that NetTraveler can handle a larger query load than solution based on previous approaches.

References

[1] S. Acharya, R. Alonso, M. J. Franklin, and S. B. Zdonik, "Broadcast disks: Data management for asymmetric communications environments," in *SIGMOD Conference*, pp. 199–210, 1995.

[2] D. Barbará and T. Imielinski, "Sleepers and workaholics: Caching strategies in mobile environments.," in *SIGMOD Conference*, pp. 1–12, 1994.

[3] M. Rodriguez-Martinez and N. Roussopoulos, "MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources," in *ACM SIGMOD*, June 2000.

[4] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. 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*, 1997.

[6] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, vol. 25, pp. 73–170, June 1993.

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

[8] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson, "Mariposa: A New Architecture for Distributed Data," in *Proc. 10th Int. Conference on Data Engineering*, (Houston, Texas), pp. 54–65, 1994.

[9] A. Delis and N. Roussopoulos, "Management of Updates in the Enhanced Client-Server Dbms," in *Proc. 14th ICDCS Conferece*, pp. 326–334, 1994.

[10] M. J. Carey, M. J. Franklin, M. Livny, and E. J. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures," in *Proc. ACM SIGMOD Congference*, pp. 357–366, 1991.

[11] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yans, "Optimizing Queries Across Diverse Data Sources," in *Proc. 23rd VLDB Conference*, pp. 276–285, 1997.

[12] T. Urhan, M. J. Franklin, and L. Amsaleg, "Cost Based Query Scrambling for Initial Delays," in *Proc. ACM SIGMOD Conference*, (Seattle, Washington, USA), pp. 130–141, 1998.

[13] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *SIGMOD Conference*, pp. 261–272, 2000.

[14] S. Viglas and J. F. Naughton, "Rate-based query optimization for streaming information sources.," in *SIGMOD Conference*, pp. 37–48, 2002.

[15] T. W. Yan and H. Garcia-Molina, "The sift information dissemination system.," *ACM Trans. Database Syst.*, vol. 24, no. 4, pp. 529–565, 1999.