DObjects: Enabling Distributed Data Services for Metacomputing Platforms

Pawel Jurczyk, Li Xiong, and Vaidy Sunderam

Emory University, Atlanta GA 30322, USA
{pjurczy,lxiong,vss}@emory.edu

Abstract. Many applications rely heavily on large amounts of data in the distributed storages collected over time or produced by large scale scientific experiments or simulations. The key constraints for building a distributed data query infrastructure for such applications are: scalability, consistency, heterogeneity and network and resource dynamics. We designed and developed DObjects, a general-purpose query and data operations infrastructure that can be integrated with metacomputing middleware. This paper describes the architecture of our data services and shows how those services were integrated with the metacomputing framework offering users an open platform for building distributed applications that require access to data integrated from multiple distributed data sources.

Keywords: Metacomputing, Distributed systems, Distributed databases, Data integration.

1 Introduction

Many applications rely heavily on large amounts of data in the distributed storages collected over time or produced by large scale scientific experiments or simulations. Consider a system that integrates the air and rail transportation networks with demographic data in order to model the large scale spread of infectious diseases (such as the SARS epidemic or pandemic influenza). Rail and air transportation databases are distributed among hundreds of local servers and demographic information is provided by a few global database servers. Such class of high-performance applications can be enabled by the construction of a networked virtual supercomputer, or *metacomputer* [1], in which high-speed networks are used to connect computers and resources located at geographically distributed sites. However, in order to integrate the distributed and heterogeneous data sources for such applications, the general metacomputing platform needs to be extended to meet a number of requirements. First, the scale of the applications can vary from a handful of nodes to several hundreds of nodes and requires good *scalability* as well as data query and update functionalities with *data consistency*. Second, the *heterogenity* of data sources requires a unified and seamless data representation and query interface for the applications. Lastly, the *dynamics* of resource and network conditions require applications to adapt

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 136–145, 2008.

dynamically in both query processing and transaction management in order to achieve scalability and data consistency.

Contributions. In this paper we introduce DObjects, a general-purpose infrastructure that extends distributed *metacomputing* platforms and provides data services for querying and operating data from heterogeneous data sources. Contribution of our research can be summarized as follows. First, the system extends the metacomputing paradigm with data services where nodes can share resources and form a virtual supercomputer and offers a scalable way for accessing and operating distributed and heterogeneous data sources. Second, it includes a distributed query execution and optimization engine that deploys and executes (sub)queries on system nodes in a *dynamic* (based on nodes' on-going knowledge of the data sources, network and node conditions) and *iterative* (right before the execution of each query operator) manner to optimize response time and throughput. Third, it includes an extension of three-phase commit protocol (3PC) that is *non-blocking* (allowing resource unlocking when nodes become unavailable), *resilient to failures* (including dynamic network partitioning and node failures even during state transitions), and *flexible* (with adjustable parameters to guarantee consistency for systems with different characteristics).

Organization. Section 2 provides a review of related work. Section 3 presents an overview of our framework, including its architecture, data operations, and query language. Section 4 and 5 present some details of our query execution engine and commit protocol respectively. Section 6 presents an initial evaluation of the system and finally Sect. 7 provides a brief conclusion.

2 Related Work

Distributed systems. It is important to position DObjects among the existing distributed system frameworks. Distributed system technologies such as DCOM[1], RMI[2], and CORBA[3] support distributed objects paradigm and can be used to build distributed applications. There are many distributed computing architectures (e.g. client-server or P2P) and platforms (BOINC [2] or Globus Toolkit [3] just to name few) built on top of these technologies. Some systems run on a volunteer basis where participants donate their unused computational power to work on interesting computational problems (such as BOINC). Others are more strict about the participants of the computing network. For instance, Grid systems and platforms, such as Globus Toolkit [3], provide general frameworks for running software on Grid architecture. However, large administrative effort is required to set up the Grid infrastructure. P2P systems are more suitable for ad-hoc collaborations, characterized by more dynamic participation patterns than those observed in Grid systems. H2O [4] is a metacomputing platform for resource sharing designed to avoid the administrative burden related to using

[1] `http://msdn2.microsoft.com/en-us/library/ms809340.aspx`
[2] `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`
[3] `http://www.corba.org/`

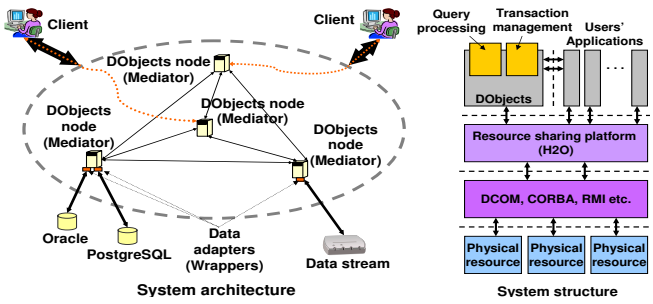

Fig. 1. System architecture

Grid systems. It implements a model where the roles of resource providers, service deployers and users can be separated. This makes resource sharing easier for providers, in the spirit of the P2P model. Our system builds on top of these technologies and extends the general metacomputing platform with support for distributed data access and operation services. Current implementation of DObjects builds on top of the H2O platform. The data services provided by DObjects offer query processing and transaction management substrates fully integrated with the metacomputing middleware and can be used easily and transparently in distributed applications for scalable data operations with flexible consistency guarantee. The most relevant to our work are OGSA-DAI and its extension OGSA-DQP [5] which were introduced by Grid community as a middleware assisting with access and integration of data from separate sources via the Grid. While the two approaches share a similar set of goals with DObjects, they were built on the grid/web service model and DObjects is built on the P2P metacomputing model and hence suits better middleware platforms providing resource sharing on a peer-to-peer basis.

Distributed databases. Distributed database systems have been extensively studied and many systems have been proposed over the years. Earlier distributed database systems, such as R* and SDD-1, share modest targets for network scalability (a handful of distributed sites) and assume homogeneous databases. The focus is on encapsulating distribution with ACID guarantees. Later distributed database or middleware systems, such as DISCO [6], target large-scale heterogeneous data sources and employ a centralized mediator-wrapper based architecture to address the database heterogeneity in the sense that a single mediator server integrates distributed data sources through wrappers. As the query load increases, the centralized mediator may become a bottleneck. Most recently, Internet scale query systems, such as PIER [7], target thousands or millions of massively distributed data sources and focus on efficient query routing schemes for network scalability. But they sacrifice on functionalities of complex queries and data updates and typically relax the consistency guarantee. DObjects distinguishes itself from existing solutions by addressing an important problem space that has been overlooked, namely, integrating large-scale heterogeneous data

sources with network and query load scalability as well as maintaining transaction semantics. In spirit, DObjects is a distributed mediator-based system where a federation of mediators and wrappers form a virtual system. Instead of focusing on traditional cost-based query optimization, the query processing engine of DObjects focuses on dynamically placing (sub)queries on the mediators for query-load balancing and scalability. In addition, it includes an extension of the three-phase commit protocol (3PC) [8] to provide a non-blocking, resilient, and flexible consistency guarantee for the dynamic environment.

3 DObjects Overview

Figure 1 presents our vision of deployed DObjects framework. The system has no centralized services and thus allows system administrators to avoid the burden in this area. It uses the *metacomputing* paradigm as a resource sharing substrate. Each node in the system provides its *computational power* that can be used by others during query execution. In addition, nodes can run *data adapters* which pull data from external data sources and transform it to a uniform format that is expected while building query responses. Front-end users can connect to any system node; however, while the physical connection is established between a client and one of the system nodes, the logical connection is established between a client node and a virtual database system consisting of all the participating nodes. Our current implementation builds on top of a Java metacomputing platform, H2O, that provides light-weight, decentralized and peer-to-peer resource sharing and communication.

Data model. When user starts a query, DObjects returns persistent entities which are data represented as objects. From user's perspective, query responses are objects of desired type. Each data object has a set of *attributes*, divided into two groups: *simple* and *referential*. Simple attributes represent simple types, such as numbers or strings. Referential attributes follow an object-oriented idiom and allow the definition of *association*, *composition* or *collection* relations between data objects. Thus, when a referential attribute is accessed, another persistent entity, or a collection of persistent entities, is obtained. A set of available data types in the system along with their attributes is defined in the system configuration. Each configuration entry has a full description of an object, i.e. its type name and a list of simple attributes and referential attributes. When a referential attribute is defined, one has to specify the foreign key information that is required to join the referencing object and referenced object. It also specifies a list of nodes (sources) where given objects can be found. Each source is specified with: 1) name of the node, 2) remote data object name, and 3) attribute mappings that define the semantic mappings between the remote data object and the current object. There is no centralized copy of the global configuration. For systems with a handful DObjects nodes (the number of data sources can be still large), the configuration can be replicated and synchronized at every node as the cost of synchronization will be relatively small. For larger scale systems with more DObjects nodes, the global schema can be replicated at a subset of

```
<persistent-entity name="CityInformation">
 <definition><attribute name="name" type="String"/>
  <list name="lRails" type="RailroadConnection" local-key="id" remote-key="c_id"/>
  <list name="lFlights" type="Flight" local-key="id" remote-key="c_id"/></definition>
 <sources> <source name="place_a" remote-object="X">
   <attribute-mapping name="name" remote-attr="city_name"/>
          </source> </sources> </persistent-entity>
```

Fig. 2. Persistent entity configuration

the DObjects nodes such as landmark nodes. An example configuration corresponding to the example mentioned in Sect. 1 is provided in Fig. 2. It defines a persistent entity of CityInformation that has 2 referential attributes: a list of RailroadConnections, and a list of Flights.

Data operations and query language. DObjects supports all standard data operations including queries and updates. Both synchronous and asynchronous queries are supported. In case of the latter user can get results *incrementally* and operate on partial results. The query language for our system could be implemented using any language that allows one to specify attributes or conditions for a given attribute in objects hierarchy. XPath or XQuery as well as OQL-like language are all valid approaches. An example of query for Dobjects using OQL-like language is presented in Fig. 3.

```
select  c.name, r.destination,
        f.flightNumber, p.lastName
from    CityInformation c, c.lRails r, c.lFlights f,
        f.lPassengers p
where   c.name like `San%` and p.lastName=`Adams`
```

Fig. 3. Query example

4 Query Processing

In this section, we introduce the query processing and optimization engine in DObjects. It is important to highlight that our approach does not attempt to optimize physical query execution performed on local databases. Responsibility for this is pushed to data adapters and data sources. Our optimization goal is at a higher level focusing on building effective sub-queries and optimal placement of those sub-queries on the system nodes to minimize the query response time and maximize system throughput. While adapting "textbook" distributed query processing techniques such as distributed join algorithms and the learning curve approach for keeping statistics about data adapters, our query processing framework presents a number of innovative aspects. First, instead of generating a set of candidate plans, mapping them physically and choosing the best ones as in conventional cost based query optimization, we create one initial abstract plan for a given query (Fig. 5 presents plan for query presented in Fig. 3). It consists of such operators as joins, data merges and select operators executed on data sources. Second, when the query plan is being executed, the node chooses active elements from the query plan in a top-down manner for execution. However, placement decisions and physical plan calculation are performed *dynamically* and *iteratively* to guarantee the best reaction to changing load or latency

```
1: generate high-level query plan tree
2: active element ← root of query plan tree
3: choose execution location for active element
4: if chosen location ≠ local node then
5:    delegate active element and its subtree to
      chosen location
6:    return
7: end if
8: execute active element;
9: for all child nodes of active element do
10:   go to step 2
11: end for
12: return result to parent element
```

Fig. 4. Local algorithm for query processing

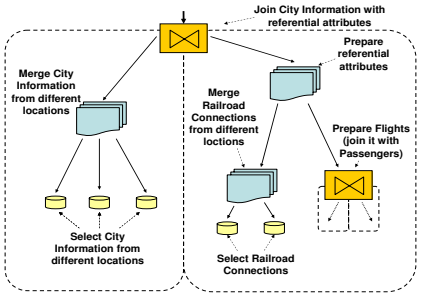

Fig. 5. Example of high-level query plan

conditions in the system. If a node finds that the best candidate for executing current element is a remote node, a *migration of workload* occurs. In order to choose the best node for migration, we deploy a network and resource-aware cost model that dynamically adapts to network conditions (such as delays in interconnection network) and resource conditions (such as load of nodes). Figure 4 presents a sketch of the local query execution process. Below we briefly describe the two cost features of our cost model. More details can be found in [9].

Latency between nodes. To compute the network latency between each pair of nodes efficiently, each DObjects node maintains a virtual coordinate [10], such that the Euclidean distance between two coordinates is an estimate for communication latency. The overhead of maintaining a virtual coordinate is small because a node can calculate its coordinate after probing a small subset of nodes such as well-known landmark nodes or randomly chosen nodes.

Load of nodes. The second feature of our cost metric is *load* of the nodes. Given our desired feature to support *cross-platform* applications, instead of depending on any OS specific functionalities for the load information, we incorporated a solution that assures good results in heterogeneous environment. The main idea is based on time measurement of execution of predefined test program that considers computing and multithreading capabilities of machines.

5 Transaction Management

In this section we introduce the distributed commit protocol in DObjects. To support a wide range of dynamic and large-scale distributed networks, the desired features for our protocol include: (1) efficiency and scalability, (2) resistance to dynamic environment, and (3) flexibility of consistency level. The basic version of 3PC does not support *network partitioning* or *multiple node failures*. Moreover, it assumes atomicity of transitions (e.g. messages can either be sent to all the cohorts or to none of them). Such an assumption is hard to implement in distributed computing paradigms. Extensions of 3PC, such as Q3PC [11], while addressing some of the issues through the idea of

Table 1. Consistency levels for commit protocol (TT is transaction timeout)

	No logging	Optimistic logging	Full logging
Finite TT	Consistency if protocol finishes before timeout (node failures cause inconsistency). Possibly no information about transaction outcome.	Consistency if protocol finishes before TT (node failures can cause inconsistency).	Consistency if protocol finishes before TT.
Infinite TT	Resistant to network failures, node failures cause inconsistency. Possibly no information about transaction outcome.	Resistant to network failures, node failures can cause inconsistency.	Full consistency, blocking protocol.

quorum, suffer cost of blocking. The commit protocol we designed and implemented in DObjects is presented in Fig. 6. The protocol proceeds similarly to classical 3PC. First, coordinator sends commit request to all cohorts. If any cohort does not agree, the transaction is aborted. Next, after all cohorts agreed to the transaction, coordinator sends prepare to commit message. After all cohorts acknowledge receiving of the prepare message, coordinator sends commit messages and commits. The key idea in our protocol is to introduce an additional coordinator's state u and use quorum to handle the node failures and non-atomic state transitions and to use two adjustable transaction parameters to offer a non-blocking alternative and the flexibility. The first parameter, *log level*, allows users to switch between the amount of logged information by each node (no log, optimistic log or full log). The second, *transaction timeout TT*, defines maximum time that an unresolved transaction can persist in the system (in addition to the node failure timeout between state transitions). When the timeout is reached, decision is made even if some participants of the commit protocol are not available. The communication among cohorts is similar to quorum-based protocols, namely when failures occur nodes contact others to find out the outcome of the transaction. The addition of the parameters, however, allows the protocol to offer a non-blocking alternative when maximum transaction timeout is reached. The different logging level also allows the system to have the flexibility of being able to conduct full recovery

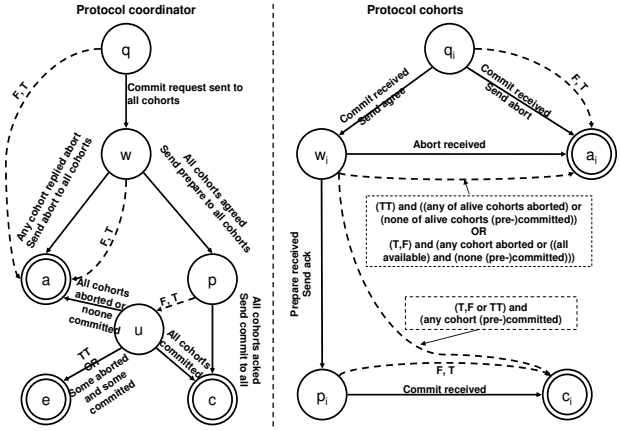

Fig. 6. Commit protocol. T is timeout waiting for next message, F is failure transition, TT is *transaction timeout*.

with a higher overhead or sacrifice consistency for efficiency. Table 1 summarizes the different consistency levels achieved by different parameter configurations. We believe that the new-generation distributed data systems need to offer this flexibility and leave decisions to users who can choose between full consistency or some risk of inconsistent state in case of failures for a trade-off of efficiency. For detailed description and analysis of the protocol we refer readers to [12].

6 Initial Deployment and Experimental Evaluation

The framework is fully implemented with current version available for download [4]. In this section we present an initial deployment and evaluation of our system in terms of its feasibility. For rigorous performance evaluations and simulations under different parameter settings of the query processing engine and commit protocol, we refer to [9] [12].

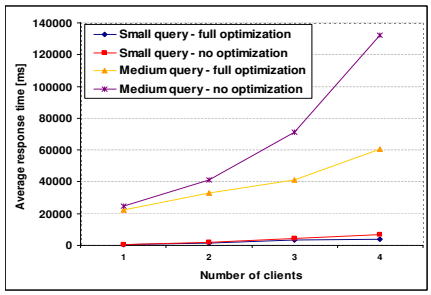

Fig. 7. Average query response time

We deployed DObjects framework on four nodes started on general-purpose PCs (Intel Core 2 Duo, 1 GB RAM, Fast Ethernet network connection). The configuration involved the following three objects: 10,000 CityInformation objects (provided by node 1), 50,000 Flight objects (20,000 provided by nodes 2 and 30,000 provided by node 3) and 70,000 RailroadConnection objects (provided by node 3). Node 4 was used only for computational purposes. The database engine nodes used was PostgreSQL 8.2.

Query processing. We measured response time for different query workloads including small queries (returning CityInformations only) and medium queries (returning CityInformations along with list of Flights and list of RalilroadConnections). Figure 7 presents results for small and medium queries for various number of parallel clients. Clearly the response time is significantly reduced when query optimization is used. The response time may seem a bit high at the first glance. To give an idea of the overhead introduced by our system, we integrated all the databases used in the experiment above into one single database and tested a medium query from Java API using Hibernate framework (using one client). The query along with results retrieval took an average of 16 s. For the same query, our system took 20 s that is in fact comparable to the case of a single database. While the overhead introduced by DObjects cannot be neglected, it does not exceed reasonable boundary and does not disqualify our system as every middleware is expected to add some overhead. In this deployment, the overhead is mainly an effect of network communication. In addition, the cost of distributed computing middleware adds to the overhead which is the price a

[4] `http://www.mathcs.emory.edu/Research/Area/datainfo/dobjects`

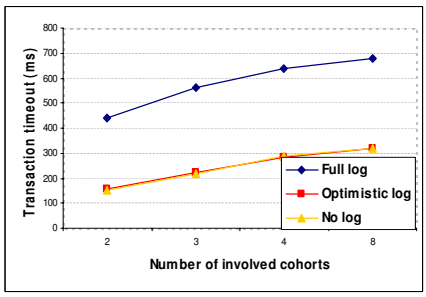

Fig. 8. Transaction response time

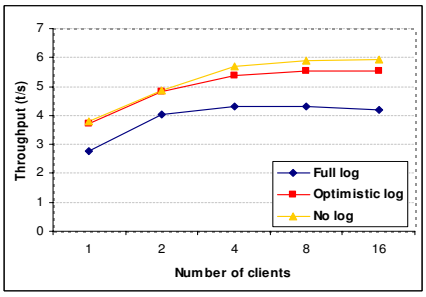

Fig. 9. Throughput of transactions

user needs to pay for convenient access to distributed data. However, for larger setup with larger number of clients, we expect our system to perform better than *centralized* approach as the benefit from distributed computing paradigm and load distribution will outweigh the overhead.

Transaction management. We also evaluated the performance of the commit protocol and the impact of different logging levels through the deployment as described earlier. The experiment was conducted by submitting data update requests (create and delete operations one by one) involving data objects hosted by a desired number of nodes (which corresponds to number of cohorts involved in transactions). Figure 8 presents average response time for varying number of cohorts. As expected, the reduction from full logging to optimistic logging in *log level* yields better performance for the transactions and the difference between optimistic logging and no logging is very small. Figure 9 presents average transaction throughput for different number of independent clients for three cohorts. As expected, limiting amount of logged information led to better throughput. Moreover, the difference between optimistic logging and no logging can be observed. Especially for larger number of clients, when the system becomes overloaded, no logging shows better performance. Such a phenomenon is not surprising, as writing even a small amount of information to persistent storage (e.g. hard drive) is considerably expensive in case of high system resources utilization.

7 Conclusion and Future Work

We have introduced DObjects, a distributed data objects framework that facilitates integration of data from large scale heterogeneous sources and can be used easily and transparently in distributed applications. We have discussed its architecture built on top of a metacomputing platform for addressing both geographic and load scalability, its dynamic query processing engine with local query migrations that dynamically adjusts to the network and resource conditions, and its commit protocol that is scalable and resistant to dynamic network and node failures and, most importantly, provides a configurable level of consistency depending on specific application and system deployment characteristics.

Our approach was validated through real implementation and deployment. The ongoing and future efforts include further enhancement for query optimization with a broader set of cost features (for instance dynamic migration of active operators in real-time from one node to another if load situation changes) and support for continuous queries. We are also considering fault tolerance properties of the commit protocol. In particular, we are planning to introduce data replications and extend the system with tolerance of Byzantine node failures.

References

1. Smarr, L., Catlett, C.E.: Metacomputing. Commun. ACM 35, 44–52 (1992)
2. Anderson, D.P.: BOINC: A system for public-resource computing and storage. In: Fifth IEEE/ACM International Workshop on Grid Computing, pp. 4–10. IEEE Computer Society, Washington (2004)
3. Foster, I.: Globus Toolkit Version 4: Software for Service-Oriented Systems. In: Jin, H., Reed, D., Jiang, W. (eds.) NPC 2005. LNCS, vol. 3779, pp. 2–13. Springer, Heidelberg (2005)
4. Kurzyniec, D., Wrzosek, T., Drzewiecki, D., Sunderam, V.: Towards Self-organizing Distributed Computing Frameworks: The H2O Approach. Parallel Processing Letters 13, 273–290 (2003)
5. Alpdemir, M.N., Mukherjee, A., Gounaris, A., Paton, N.W., Fernandes, A.A., Sakellariou, R., Watson, P., Li, P.: Using OGSA-DQP to Support Scientific Applications for the Grid. In: Herrero, P., S. Pérez, M., Robles, V. (eds.) SAG 2004. LNCS, vol. 3458, pp. 13–24. Springer, Heidelberg (2005)
6. Tomasic, A., Raschid, L., Valduriez, P.: Scaling access to heterogeneous data sources with DISCO. IEEE Trans. on Knowl. and Data Eng. 10, 808–823 (1998)
7. Huebsch, R., Hellerstein, J.M., Lanham, N., Loo, B.T., Shenker, S., Stoica, I.: Querying the internet with PIER. In: 29th International Conference on Very Large Data Bases, pp. 321–332, VLDB Endowment, Berlin (2003)
8. Skeen, D., Stonebraker, M.: A formal model of crash recovery in a distributed system. In: Concurrency control and reliability in distributed systems, pp. 295–317. Van Nostrand Reinhold Co., New York (1987)
9. Jurczyk, P., Xiong, L.: DObjects: a metacomputing framework with dynamic query processing for distributed data networks. Technical Report TR-2007-015, Emory University, Math&CS Dept., Atlanta, USA (2007)
10. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: A Decentralized Network Coordinate System. SIGCOMM Comput. Commun. Rev., 15–26 (2004)
11. Skeen, D.: A Quorum-Based Commit Protocol. Technical report, Cornell University, Ithaca, USA (1982)
12. Jurczyk, P., Xiong, L.: Adapting Distributed Commit Protocol for Dynamic Metacomputing Frameworks. Technical Report TR-2007-025, Emory University, Math&CS Dept., Atlanta, USA (2007)