Data Management in the APPA P2P System[1]

Reza Akbarinia[1,2], Vidal Martins[1,3]

[1]ATLAS group, INRIA and LINA, University of Nantes, France
[2]Shahid Bahonar University of Kerman, Iran
[3]PPGIA/PUCPR – Pontifical Catholic University of Paraná, Brazil
Firstname.Lastname@univ-nantes.fr

Abstract. Peer-to-peer (P2P) computing offers new opportunities for building highly distributed data systems. Unlike client-server computing, P2P is a very dynamic environment where peers can join and leave the network at any time and offers important advantages such as operation without central coordination, peers autonomy, and scale up to large number of peers. However, providing high-level data management services (schema, queries, replication, availability, etc.) in a P2P system implies revisiting distributed database technology in major ways. In this paper, we present APPA (Atlas Peer-to-Peer Architecture) and its high-level data management services. APPA has a network-independent architecture that can be implemented over various structured and super-peer P2P networks. It uses novel solutions for persistent data management with updates, data replication with semantic-based reconciliation and query processing. APPA's services are implemented using the JXTA framework.

1. Introduction

Data management in distributed systems has been traditionally achieved by distributed database systems [16] which enable users to transparently access and update several databases in a network using a high-level query language (*e.g.* SQL). Transparency is achieved through a global schema which hides the local databases' heterogeneity. In its simplest form, a *distributed database system* is a centralized server that supports a global schema and implements distributed database techniques (query processing, transaction management, consistency management, etc.). This approach has proved effective for applications that can benefit from centralized control and full-fledge database capabilities, *e.g.* information systems. However, it cannot scale up to more than tens of databases. Data integration systems [21][23] extend the distributed database approach to access data sources on the Internet with a simpler query language in read-only mode. Parallel database systems [24] also extend the distributed database approach to improve performance (transaction throughput or query response time) by exploiting database partitioning using a multiprocessor or cluster system. Although data integration systems and parallel database systems can scale up to hundreds of data sources or database partitions, they still rely on a centralized global schema and strong assumptions about the network.

[1] Candidate to the Best Student Paper Award.

In contrast, peer-to-peer (P2P) systems adopt a completely decentralized approach to data sharing. By distributing data storage and processing across autonomous peers in the network, they can scale without the need for powerful servers. Popular examples of P2P systems such as Gnutella [7] and Freenet [6] have millions of users sharing petabytes of data over the Internet. Although very useful, these systems are quite simple (*e.g.* file sharing), support limited functions (*e.g.* keyword search) and use simple techniques (*e.g.* resource location by flooding) which have performance problems. To deal with the dynamic behavior of peers that can join and leave the system at any time, they rely on the fact that popular data get massively duplicated.

Initial research on P2P systems has focused on improving the performance of query routing in the unstructured systems which rely on flooding. This work led to structured solutions based on distributed hash tables (DHT), *e.g.* CAN [18] and Chord [20], or hybrid solutions with super-peers that index subsets of peers [15]. Although these designs can give better performance guarantees, more research is needed to understand their trade-offs between fault-tolerance, scalability, self-organization, etc.

Recently, other work has concentrated on supporting advanced applications which must deal with semantically rich data (*e.g.* XML documents, relational tables, etc.) using a high-level SQL-like query language, *e.g.* ActiveXML [1], Edutella [15], Piazza [22], PIER [9]. As a potential example of advanced application that can benefit from a P2P system, consider the cooperation of scientists who are willing to share their private data (and programs) for the duration of a given experiment. For instance, medical doctors in a hospital may want to share some patient data for an epidemiological study. Medical doctors may have their own, independent data descriptions for patients and should be able to ask queries like “age and last weight of the male patients diagnosed with disease X between day1 and day2” over their own descriptions.

Such data management in P2P systems is quite challenging because of the scale of the network and the autonomy and unreliable nature of peers. Most techniques designed for distributed database systems which statically exploit schema and network information no longer apply. New techniques are needed which should be decentralized, dynamic and self-adaptive.

In this paper, we present high-level data management services in APPA (Atlas Peer-to-Peer Architecture), a P2P data management system which we are building. The main objectives of APPA are scalability, availability and performance for advanced applications [2]. APPA has a network-independent architecture that can be implemented over various structured and super-peer P2P networks. This allows us to exploit continuing progress in such systems. To deal with semantically rich data, APPA supports decentralized schema management and uses novel solutions for persistent data management with updates, data replication with semantic-based reconciliation and query processing. APPA’s services are implemented using the JXTA framework [10].

The rest of the paper is organized as follows. Section 2 describes the APPA architecture. Section 3 introduces the APPA’s solution to persistent data management and support for updates. Section 4 describes its solution to high level data replication and distributed semantic reconciliation. Section 5 introduces the query processing strategy in APPA. Section 6 concludes.

2. APPA Architecture

APPA has a layered service-based architecture. Besides the traditional advantages of using services (encapsulation, reuse, portability, etc.), this enables APPA to be network-independent so it can be implemented over different structured (*e.g.* DHT) and super-peer P2P networks. The main reason for this choice is to be able to exploit rapid and continuing progress in P2P networks. Another reason is that it is unlikely that a single P2P network design will be able to address the specific requirements of many different applications. Obviously, different implementations will yield different trade-offs between performance, fault-tolerance, scalability, quality of service, etc. For instance, fault-tolerance can be higher in DHTs because no peer is a single point of failure. On the other hand, through index servers, super-peer systems enable more efficient query processing. Furthermore, different P2P networks could be combined in order to exploit their relative advantages, *e.g.* DHT for key-based search and super-peer for more complex searching.

There are three layers of services in APPA: P2P network, basic services and advanced services.

P2P network. This layer provides network independence with services that are common to different P2P networks:

- **Peer id assignment:** assigns a unique id to a peer using a specific method, *e.g.* a combination of super-peer id and counter in a super-peer network.
- **Peer linking:** links a peer to some other peers, *e.g.* by locating a zone in CAN.
- **Key-based storage and retrieval (KSR):** stores and retrieves a (*key*, *data*) pair in the P2P network, *e.g.* through hashing over all peers in DHT networks or using super-peers in super-peer networks. An important aspect of KSR is that it allows managing data using object semantics (*i.e.* with KSR it is possible to get and set specific data attributes).
- **Key-based timestamping (KTS):** generates monotonically increasing timestamps which are used for ordering the events occurred in the P2P system. This service is useful to improve data availability.
- **Peer communication:** enables peers to exchange messages (*i.e.* service calls).

Basic services. This layer provides elementary services for the advanced services using the P2P network layer:

- **Persistent data management (PDM):** provides high availability for the *(key, data)* pairs which are stored in the P2P network.
- **Peer management:** provides support for peer joining, rejoining and for updating peer address (the peer ID is permanent but its address may be changed).
- **Group membership management:** allows peers to join an abstract *group*, become *members* of the group and send and receive membership notifications. This is similar to group communication [4][5].

Advanced services. This layer provides advanced services for semantically rich data sharing including schema management, replication [13][14], query processing [3], security, etc. using the basic services.

3. Persistent Data Management

One of the main characteristics of P2P systems is the dynamic behavior of peers which can join and leave the system frequently, at anytime. When a peer gets offline, the data it stores becomes unavailable. To improve data persistence, we can rely on data replication by storing (*k, data*) pairs at several peers. If one peer is unavailable, the data can still be retrieved from the other peers that hold a replica. However, the mutual consistency of the replicas after updates can be compromised as a result of peers leaving the network or concurrent updates. Therefore, some of the replicas may not be *current*, *i.e.* they do not reflect the latest data stored with *k* in the P2P network. For some applications (*e.g.* agenda management, bulletin boards, cooperative auction management, reservation management, etc.) having the ability to get a current replica is very important.

In APPA, the PDM service provides data persistence through replication by using multiple hash functions. It also addresses efficiently the problem of retrieving current replicas based on timestamping. For doing its tasks, PDM takes advantage of KSR and KTS which are two services in the APPA's P2P network layer.

In this section, we first discuss how PDM provides data persistence, then we introduce the concept of timestamping, and finally we present the update operations which are the main operations of the PDM service.

3.1 Data Persistence Using Multiple Hash Functions

In APPA, the KSR service maps a key *k* to a peer *p* using a hash function *h*. We call *p* the *responsible for k wrt. h*, and denote it by *rsp(k, h)*. A peer may be responsible for *k wrt.* a hash function h_1 but not responsible for *k wrt.* another hash function h_2. There is a set of hash functions *H* which can be used for mapping the keys to peers. The KSR service has an operation $put_h(k, data)$ that, given a hash function $h \in H$, a data item *data* and its associated key *k*, stores the pair *(k, data)* at *rsp(k,h)*. This operation can be issued concurrently by several peers. There is another operation $get_h(k)$ that retrieves the data associated with *k* stored at *rsp(k,h)*.

To improve data persistence, PDM stores each data and its associated key at several peers using a set of hash functions $H_r \subset H$. the set H_r is called the set of *replication hash functions*. The number of replication hash functions, *i.e.* $|H_r|$, can be different for different P2P networks. For instance, in a P2P network with low peer's availability, data availability can be increased using a high value of $|H_r|$ (*e.g.* 20).

Over time, some of the replicas stored with *k* at some peers may get stale, *e.g.* due to the absence of some peers at update time. To be able to return current replicas, before storing a data, PDM "stamps" it with a logical timestamp which is generated by KTS. Therefore, given a data item *data* and its associated key *k*, $\forall h \in H_r$, PDM replicates the pair *(k, {data, timestamp})* at *rsp(k,h)*. Upon a request for the data associated with a key, PDM returns one of the replicas which are stamped with the latest timestamp.

3.2 Timestamping

To generate timestamps, APPA uses KTS which is a distributed service. The main operation of KTS is *gen_ts(k)* which, given a key *k*, generates a real number as a *timestamp for k*. The timestamps generated by KTS have the *monotonicity* property, *i.e.* two timestamps generated for the same key are monotonically increasing. In other words, for any two timestamps ts_1 and ts_2 generated for a key *k* respectively at times t_1 and t_2, if $t_1 < t_2$ then we have $ts_1 < ts_2$. This property permits us to order the timestamps generated for the same key according to the time at which they have been generated.

KTS generates the timestamps in a completely distributed fashion, using local logical counters. At anytime, it generates at most one timestamp for a key *k*. Thus, regarding the monotonicity property, there is a total order on the set of timestamps generated for the same key. However, there is no total order on the timestamps generated for different keys. In addition to *gen_ts*, KTS has another operation denoted by *last_ts(k)* which, given a key *k*, returns the last timestamp generated for *k* by KTS.

The idea of timestamping by KTS is like the idea of data storage in DHTs which is based on having a responsible for storing each data and determining the responsible dynamically using a hash function. In KTS, for each key we have a responsible of timestamping which is determined dynamically using a hash function. Due to space limitations, we don not describe the details of KTS.

3.3 Update Operations

The main operations of the PDM service are *insert* and *retrieve* operations. The detail of these operations is as follows.

Insert(k, data): replicates a data and its associated key in the P2P network as follows. First, it uses KTS to generate a timestamp for *k*, *e.g. ts*. Then, for each $h \in H_r$ it stores the pair *(k, {data, ts})* at the peer that is *rsp(k,h)*. When a peer *p*, which is responsible for *k wrt.* one of the hash functions involved in H_r, receives the pair *(k, {data, ts})*, it compares *ts* with the timestamp, say ts_0, of its data (if any) associated with *k*. If $ts > ts_0$, *p* overwrites its data and timestamp with the new ones. Recall that, at anytime, *KTS.gen_ts (k)* generates at most one timestamp for *k*, and different timestamps for *k* have the monotonicity property. Thus, in the case of concurrent calls to *insert(k, data), i.e.* from different peers, only the one that obtains the latest timestamp will succeed to store its data in the P2P network.

Retrieve(k): retrieves the most recent replica associated with *k* in the P2P network as follows. First, it uses KTS to determine the latest timestamp generated for *k*, *e.g.* ts_1. Then, for each hash function $h \in H_r$, it uses the KSR operation $get_h(k)$ to retrieve the pair *{data, timestamp}* stored along with *k* at *rsp(k,h)*. If *timestamp* is equal to ts_1, then the data is a current replica which is returned as output and the operation ends. Otherwise, the retrieval process continues while saving in $data_{mr}$ the most recent replica. If no replica with a timestamp equal to ts_1 is found (*i.e.* no current replica is found) then the operation returns the most recent replica available, *i.e.* $data_{mr}$.

4. Data Replication

Data replication is largely used to improve data availability and performance in distributed systems. In APPA, PDM is a low-level service that employs data replication to improve the availability of pairs (*key*, *data*) stored in the P2P network. For solving update conflicts by taking into account application semantics, APPA provides a higher-level replication service. This service is an optimistic solution [19] that allows the asynchronous updating of replicas such that applications can progress even though some nodes are disconnected or have failed. As a result, users can collaborate asynchronously. However, concurrent updates may cause replica divergence and conflicts, which should be reconciled.

In this section, we present the DSR algorithm (Distributed Semantic Reconciliation) [13][14], a dynamic distributed version of the semantic reconciliation provided by IceCube [11][17]. Unlike IceCube, DSR is based on a distributed and parallel approach. With DSR, a subset of nodes, called reconcilers, are selected to concurrently reconcile conflicting updates. DSR works properly over cluster systems and grid architectures (*e.g.* we have implemented a DSR prototype [13] and validated this prototype on the cluster of the Paris team at INRIA [8]); however, our research is focused on P2P systems. We now describe the main terms and assumptions we consider for DSR followed by the main DSR algorithm itself.

We assume that DSR is used in the context of a virtual community which requires a high level of collaboration and relies on a reasonable number of nodes (typically hundreds or even thousands of interacting users) [25].

In our solution, a *replica R* is a copy of a collection of objects (*e.g.* copy of a relational table, or an XML document). A *replica item* is an object belonging to a replica (*e.g.* a tuple in a relational table, or an element in an XML document). We assume *multi-master* replication, *i.e.* a replica *R* is stored in several nodes and all nodes may read or write *R*. Conflicting updates are expected, but with low frequency.

In order to update replicas, nodes produce *tentative* actions (henceforth actions) that are executed only if they conform to the application semantics. An *action* is defined by the application programmer and represents an application-specific operation (*e.g.* a write operation on a file or document, or a database transaction). The application semantics is described by means of constraints between actions. A *constraint* is the formal representation of an application invariant (*e.g.* an update cannot follow a delete).

On the one hand, users and applications can create constraints between actions to make their intents explicit (they are called *user-defined constraints*). On the other hand, the reconciler node identifies conflicting actions, and asks the application if these actions may be executed together in any order (*commutative* actions) or if they are mutually dependent. New constraints are created to represent semantic dependencies between conflicting actions (they are called *system-defined constraints*). Details about the language used to express constraints can be found in [17].

A *cluster* is set of actions related by constraints, and a *schedule* is a set of ordered actions that do not violate constraints.

With DSR, data replication proceeds basically as follows. First, nodes execute local actions to update replicas while respecting user-defined constraints. Then, these actions (with the associated constraints) are stored in the P2P network using the PDM service. Finally, reconciler nodes retrieve actions and constraints from the P2P network and produce a global schedule, by performing conflict resolution in 5 distributed steps based on the application semantics. This schedule is locally executed at every node, thereby assuring eventual consistency [17]. The replicated data is eventually consistent if, when all nodes stop the production of new actions, all nodes will eventually reach the same value in their local replicas.

In order to avoid communication overhead and due to dynamic connections and disconnections, we distinguish *replica nodes*, which are the nodes that hold replicas, from *reconciler nodes*, which is a subset of the replica nodes that participate in distributed reconciliation.

We now present DSR in more details. We first introduce the reconciliation objects necessary to DSR. Then, we present the five steps of the DSR algorithm. Finally, we describe how DSR deals with dynamic connections and disconnections.

4.1 Reconciliation Objects

Data managed by DSR during reconciliation are held by *reconciliation objects* that are stored in the P2P network giving the object identifier. To enable the storage and retrieval of reconciliation objects, each reconciliation object has a unique identifier. DSR uses five reconciliation objects:

- **Action log *R* (noted L_R)**: it holds all actions that try to update the replica *R*.
- **Action groups of *R* (noted G_R)**: actions that manage a common replica item are put together into the same action group in order to enable the parallel checking of semantic conflicts among actions (each action group can be checked independently of the others); every replica *R* may have a set of action groups, which are stored in the *action groups of R* reconciliation object.
- **Clusters set (noted *CS*)**: all clusters produced during reconciliation are included in the *clusters set* reconciliation object; a cluster is not associated with a replica.
- **Action summary (noted *AS*)**: it comprises constraints and action memberships (an action is a *member* of one or more clusters).
- **Schedule (noted *S*)**: it contains a set of ordered actions.

The node that holds a reconciliation object is called the *provider node* for that object (*e.g. schedule provider* is the node that currently holds *S*).

4.2 DSR Algorithm

DSR executes reconciliation in 5 distributed steps as showed in Figure 1.

- **Step 1 – actions grouping**: for each replica *R*, reconcilers put actions that try to update common replica items of *R* into the same group, thereby producing G_R.
- **Step 2 – clusters creation**: reconcilers split action groups into clusters of semantically dependent conflicting actions (two actions a_1 and a_2 are *semantically independent* if the application judge safe to execute them together, in any order, even if a_1 and a_2 update a common replica item; otherwise, a_1 and a_2 are *semantically dependent*). Clusters produced in this step are stored in the clusters set, and the associated action memberships are included in the action summary.

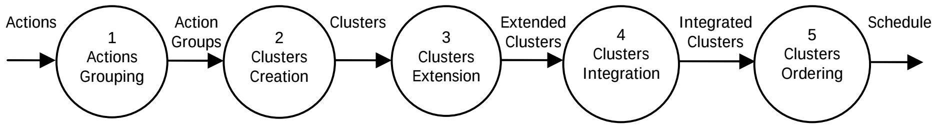

Fig. 1. DSR Steps

- **Step 3 – clusters extension**: user-defined constraints are not taken into account in clusters creation. Thus, in this step, reconcilers extend clusters by adding to them new conflicting actions, according to user-defined constraints. The associated action memberships are also included in the action summary.
- **Step 4 – clusters integration**: clusters extensions lead to the overlap of clusters' actions (an overlap occurs when different clusters have common actions, and this is identified by analyzing action memberships). In this step, reconcilers bring together overlapping clusters, thereby producing integrated clusters.
- **Step 5 – clusters ordering**: in this step, reconcilers produce the global schedule by ordering actions of integrated clusters; all replica nodes execute this schedule.

At every step, the DSR algorithm takes advantage of data parallelism, *i.e.* several nodes perform simultaneously independent activities on a distinct subset of actions (*e.g.* ordering of different clusters). No centralized criterion is applied to partition actions. In fact, whenever a set of reconciler nodes request data to a provider, the provider node naively supplies reconcilers with about the same amount of data (the provider node knows the maximal number of reconcilers because it receives this information from the node that launches reconciliation).

DSR avoids network overhead by minimizing the number of exchanged messages and the amount of transferred data. The number of messages is linear *wrt.* the number of reconcilers, and the number of reconcilers is not large. Concerning data transfer, most of messages carry only data identifiers (*e.g.* actions' identifiers) instead of the entire data items.

4.3 Managing Dynamic Disconnections and Reconnections

Whenever distributed reconciliation takes place, a set of nodes N_d may be disconnected. As a result, the global schedule is not applied by nodes of N_d. Moreover, actions produced by N_d nodes and not yet stored in the P2P network via APPA PDM service are not reconciled. In order to assure eventual consistency despite disconnections, the APPA replication service proceeds as follows. Each node locally stores the identifier of the last schedule it has locally executed (noted S_{last}). In addition, the replication service stores in the P2P network (using the APPA PDM service) a chronological sequence of schedules' identifiers produced by reconciliations, which is called *schedule history* and noted $H = (S_{id1}, S_{id2}, \ldots, S_{idn})$. As any reconciliation object, the schedule history has a unique identifier. The application knows this identifier and can provide it to the reconciler nodes. When a node n of N_d reconnects, it proceeds as follows: (1) n checks whether S_{last} is equal to S_{idn}, and, if not (*i.e.* n's replicas are out of date), n locally applies all schedules that follow S_{last} in the H history; (2) actions locally produced by n and not yet stored in the P2P network using the APPA PDM service are put into the involved action logs for later reconciliation.

At the beginning of reconciliation, a set of connected replica nodes must be allocated to proceed as reconciler nodes. To minimize reconciliation time, such allocation should be dynamic, *i.e.* nodes should be allocated based on the reconciliation context (*e.g.* number of actions, number of replicas, network properties, etc.). Currently, we are elaborating a cost model and the associated algorithms for allocating reconciler nodes based on communication costs. These algorithms take into account cost changes due to dynamic disconnections and reconnections.

5. Query Processing

Query processing in APPA deals with schema-based queries and considers data replication. In this section, we first present schema mapping in APPA, and then we describe the main phases of query processing.

5.1 Schema Mapping

In order to support schema-based queries, APPA must deal with heterogeneous schema management. In a P2P system, peers should be able to express queries over their own schema without relying on a centralized global schema as in data integration systems [21]. Several solutions have been proposed to support decentralized schema mapping, *e.g.* [15][22]. For instance, Piazza [22] proposes a general, network-independent, solution that supports a graph of pair-wise mappings between heterogeneous schema peers. APPA uses a simpler solution that takes advantage of the collaborative nature of the applications. It assumes that peers that wish to cooperate, *e.g.* for the duration of an experiment, agree on a *Common Schema Description* (CSD). Given a CSD, a peer schema can be specified using views. This is similar to the local-as-view approach in data integration [12] except that, in APPA, queries at a peer are expressed against the views, not the CSD.

When a peer decides to share data, it needs to define a peer schema, only once, to map its local schema to the CSD. To simplify the discussion, we use the relational model (APPA uses XML) and the Datalog-like notation of [21] for mapping rules. Thus, a peer schema includes peer mappings, one per local relation. Given 2 CSD relation definitions r_1 and r_2, an example of peer mapping at peer p is:

$p{:}r(A,B,D) \subseteq csd{:}r_1(A,B,C),\ csd{:}r_2(C,D,E)$

In APPA, mapped schemas are stored in the P2P network using the PDM service.

5.2 Query Processing Phases

Given a user query on a peer schema, the objective is to find the minimum set of relevant peers (query matching), route the query to these peers (query routing), collect the answers and return a (ranked) list of answers to the user. Since the relevant peers may be disconnected, the returned answers may be incomplete.

Query processing proceeds in four main phases: (1) query reformulation, (2) query matching, (3) query optimization and (4) query decomposition and execution.

Query reformulation. The user query (on the peer schema) is rewritten in a query on CSD relations. This is similar to query modification using views. For instance, the following query at peer *p*:

select A,D from r where B=b

would be rewritten on the CSD relations as:

select A,D from r_1, r_2 *where B=b and* $r_1.C=r_2.C$

Query matching. Given a reformulated query *Q*, it finds all the peers that have data relevant to the query. For simplicity, we assume conjunctive queries. Let *P* be the set of peers in the P2P system, the problem is to find $P' \subseteq P$ where each *p* in *P'* has relevant data, *i.e.* refers to relations of *Q* in its mapped schema. These peers can be iteratively (for each *Q*'s relation) retrieved using the PDM service. Let *R* be the set of relations involved in *Q*, and *ms(p,r)* denote that the mapped schema of peer *p* involves relation *r*, query matching produces:

$P' = \{ p \mid (p \in P) \wedge (\exists r \in R \wedge ms(p,r)) \}$

Query optimization. Because of data replication, each relevant data may be replicated at some peers in *P'*. The optimization objective is to minimize the cost of query processing by selecting best candidate peer(s) for each relevant data based on a cost function. Selecting more than one candidate peer is necessary in a very dynamic environment since some candidate peers may have left the network. Thus, selecting several candidate peers increases the answer's completeness but at the expense of redundant work. This step produces a set $P'' \subseteq P'$ of best peers.

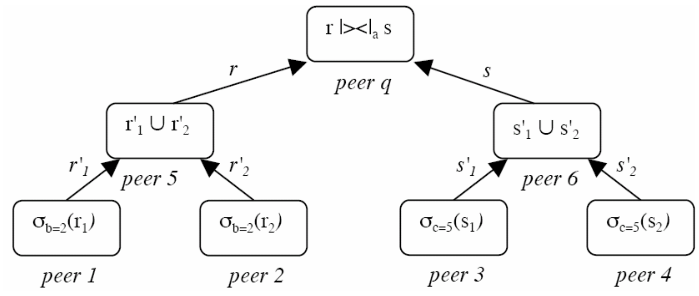

Fig. 2. Example of parallel execution using intermediate peers. This strategy exhibits independent parallelism between peers 1-4 (the select (σ) operations can all be done in parallel) and peers 5-6 (the union operations can be done in parallel). It can also yield pipelined parallelism. For instance, if the left-hand operand of an intermediate peer is smaller than the right-hand operand, then it would be entirely transferred first so the other operand could be pipelined thus yielding parallelism between peers 2-5-*q* and peers 4-6-*q*. Parallel execution strategies improve both the query response time and the global efficiency of the P2P system.

Query decomposition and execution. This phase is similar to that in data integration systems and APPA reuses well-known, yet sophisticated techniques. Since some peers in *P"* may have only subsets of *Q*'s relations, query decomposition produces a number of subqueries (not necessarily different), one for each peer, together with a composition query to integrate, *e.g.* through join and union operations, the intermediate results [12]. Finally, the subqueries are sent to the peers in *P"*, which reformulate it on their local schema (using the peer mappings), execute it, and send the results back to the sending peer, who integrates the results. Result composition can also exploit parallelism using intermediate peers. For instance, let us consider relations r_1 and r_2 defined over CSD *r* and relations s_1 and s_2 defined over CSD *s*, each stored at a different peer, and the query *select * from r, s where r.a=s.a and r.b=2 and s.c=5* issued by a peer *q*. A parallel execution strategy for *Q* is shown in Figure 2.

6. Conclusion

In this paper, we presented APPA (Atlas Peer-to-Peer Architecture), a P2P data management system for supporting advanced applications which must deal with semantically rich data (*e.g.* XML documents, relational tables, etc.). Such applications typically have a collaborative nature as in distributed scientific experimentation where scientists wish to share data and programs.

APPA has a network-independent architecture that can be implemented over various structured and super-peer P2P networks. It provides network services (peer id assignment, peer linking, peer communication, key-based storage and retrieval, etc.), basic services (persistent data management, peer management, group membership) and advanced services such as schema management, replication and query processing. The main advantage of such architecture is to be able to exploit rapid and continuing progress in P2P networks.

APPA uses novel solutions for persistent data management, data replication and query processing. APPA provides data persistence with high availability through replication by using multiple hash functions. It also addresses efficiently the problem of retrieving current replicas based on timestamping. APPA also provides a higher-level replication service with multi-master replication. This service enables asynchronous collaboration among users. In order to resolve conflicting updates, we use a distributed semantic-based reconciliation algorithm which exploits parallelism. Query processing in APPA deals with schema-based queries and considers data replication. The main phases of query processing are query reformulation on a common schema description, query matching to find relevant peers, query optimization to select best peers, and query decomposition and execution.

We have started the implementation of APPA using the JXTA framework [10]. APPA's advanced services are provided as JXTA community services. Only the P2P network layer of the APPA implementation depends on the JXTA platform. Thus, APPA is portable and can be used over other platforms by replacing the services of the P2P network layer. We validated some of the APPA's services on the cluster of Paris team at INRIA [8], which has 64 nodes. Additionally, in order to study the scalability of these services with larger numbers of nodes, we implemented simulators. The current version of the APPA prototype and its service simulators manage data using a Chord DHT. Experimental results showed that simulators are well calibrated and the implemented services have good performance and scale up. Details about APPA implementation can be found in [2].

References

[1] Abiteboul, S., Bonifati, A., Cobena, G., Manolescu, I., Milo, T. Dynamic XML documents with distribution and replication. *ACM SIGMOD Conf.*, 2003.

[2] Akbarinia, R., Martins, V., Pacitti, E., Valduriez, P. *Design and Implementation of Atlas P2P Architecture. Global Data Management* (Eds. R. Baldoni, G. Cortese, F. Davide), IOS Press, 2006.

[3] Akbarinia, R., Martins, V., Pacitti, E., Valduriez, P. Top-k Query Processing in the APPA P2P System. *Int. Conf. on High Performance Computing for Computational Science (VecPar)*, 2006.

[4] Castro, M., Jones, M.B., Kermarrec, A., Rowstron, A., Theimer, M., Wang, H., Wolman, A. An Evaluation of Scalable Application-level Multicast Built Using P2P Overlays. *IEEE Infocom*, 2003.

[5] Chockler, G., Keidar, I., Vitenberg, R. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(427-469), 2001.

[6] Clarke, I., Miller, S., Hong, T.W., Sandberg, O., Wiley, B. Protecting Free Expression Online with Freenet. *IEEE Internet Computing, 6(1)*, 2002.

[7] Gnutella. http://www.gnutelliums.com/.

[8] http://www.irisa.fr/paris/General/cluster.htm.

[9] Huebsch, R., Hellerstein, J., Lanham, N., Thau Loo, B., Shenker, S., Stoica, I. Querying the Internet with PIER. *VLDB Conf.*, 2003.

[10] JXTA. http://www.jxta.org/.

[11] Kermarrec, A., Rowstron, A., Shapiro, M., Druschel P. The IceCube approach to the reconciliation of diverging replicas. *ACM Symp. on Principles of Distributed Computing*, 2001.

[12] Levy, A., Rajaraman, A., Ordille, J. Querying heterogeneous information sources using source descriptions. *VLDB Conf.*, 1996.

[13] Martins, V., Akbarinia, R., Pacitti, E., Valduriez, P. Reconciliation in the APPA P2P System. *Proc. of IEEE ICPADS,* 2006.

[14] Martins, V., Pacitti, E., Valduriez, P. A Dynamic Distributed Algorithm for Semantic Reconciliation. *Distributed Data & Structures 7 (WDAS)*, 2006.

[15] Nejdl, W., Siberski, W., Sintek, M. Design issues and challenges for RDF- and schema-based peer-to-peer systems. *ACM SIGMOD Record, 32(3)*, 2003.

[16] Özsu, T., Valduriez, P. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[17] Preguiça, N., Shapiro, M., Matheson, C. Semantics-based reconciliation for collaborative and mobile environments. *Int. Conf. on Cooperative Information Systems (CoopIS)*, 2003.

[18] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S. A scalable content-addressable network. *Proc. of SIGCOMM*, 2001.

[19] Saito, Y., Shapiro, M. Optimistic Replication. *ACM Computing Surveys, 37(1)*, 2005.

[20] Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications*. Proc. of ACM SIGCOMM*, 2001.

[21] Tanaka, A., Valduriez, P. The Ecobase environmental information system: applications, architecture and open issues. *ACM SIGMOD Record, 3(5-6)*, 2000.

[22] Tatarinov, I., Ives, Z.G., Madhavan, J., Halevy, A., Suciu, D., Dalvi, N., Dong, X., Kadiyska, Y., Miklau, G., Mork, P. The Piazza peer data management project. *ACM SIGMOD Record 32(3)*, 2003.

[23] Tomasic, A., Raschid, L., Valduriez, P. Scaling access to heterogeneous data sources with DISCO. *IEEE Trans. on Knowledge and Data Engineering, 10(5)*, 1998.

[24] Valduriez, P. Parallel Database Systems: open problems and new issues. *Distributed and Parallel Databases, 1(2)*, 1993.

[25] Whittaker, S., Issacs, E., O'Day, V. Widening the Net: Workshop report on the theory and practice of physical and network communities. *ACM SIGCHI Bulletin*, 29(3), 1997.