Replication and Query Processing in the APPA Data Management System

Reza Akbarinia, Vidal Martins, Esther Pacitti, Patrick Valduriez
Atlas group, INRIA and LINA,
University of Nantes, France
FirstName.LastName@lina.univ-nantes.fr, Patrick.Valduriez@inria.fr

ABSTRACT
Advanced P2P applications are likely to need general replication capabilities such as variable granularity and multi-master mode. However, existing replication solutions do not address important properties of P2P systems such as self-organization. In this paper, we address replication and query processing in the context of the APPA (Atlas Peer-to-Peer Architecture) data management system. APPA has a network-independent architecture that can be implemented over various P2P networks. It supports decentralized schema management and uses XML for the shared data. We propose an optimistic replication solution which provides eventual consistency for multi-master mode. We also propose a schema-based optimistic replication solution that deals with replication.

1. INTRODUCTION
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 [8] and Kaaza [11] 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 [21] and CHORD [23], or hybrid solutions with super-peers that index subsets of peers [27]. 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 [2], Edutella [14], Piazza [25], 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.

Most of the work on sharing semantically rich data in P2P systems has focused on schema mappings between peers [5], and query processing and optimization [2][9][14][25]. However, there has been little work on managing data replication in the presence of updates. The data sharing P2P systems like Gnutella and Kaaza deal with static, read-only files (e.g. music files) for which update is not an issue. Freenet [7] partially addresses updates which are propagated from the updating peer downward to close peers that are connected. However, peers that are disconnected do not get updated. ActiveXML [2] supports the definition of replicated XML fragments as Web service calls but does not address update propagation. Update is addressed in P-Grid [1], a structured network that supports self-organization. The update algorithm uses rumor spreading to scale and provides probabilistic guarantees for replica consistency. However, it only considers updates at the file level in a mono-master mode, i.e. only one (master) peer can update a file and changes are propagated to other (read-only) replicas.

Advanced applications are likely to need more general replication capabilities such as various levels of replication granularity and multi-master mode, i.e. whereby the same replica may be updated by several (master) peers. For instance, a patient record may be replicated at several medical doctors and updated by any of them during a visit of the patient, e.g. to reflect the patient’s new weight. The advantage of multi-master replication is high-availability and high-performance since replicas can be updated in parallel at different peers. However, conflicting updates of the same data at different peers can introduce replica divergence.

Then the main problem is to assure replica consistency. In distributed database systems [16], synchronous replication (e.g. Read-Once-Write-All) which updates all replicas within the same transaction enforces mutual consistency of replicas. However, it does not scale up because it makes use of distributed transactions, typically implemented by 2 phase commit. Preventive replication [19] can yield strong consistency, without the constraints of synchronous replication, and scale up to large configurations. However, it requires support for advanced distributed services and a high speed network with guaranteed maximum time for message reception as is the case in cluster systems. This assumption does not hold for P2P systems. A more practical solution is optimistic replication [17] which allows the
independent updating of replicas and divergence until reconciliation. However, existing optimistic replication solutions do not address important properties of P2P systems such as self-organization.

In this paper, we address replication and query processing in the context of 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. APPA has a network-independent architecture in terms of basic and advanced services that can be implemented over different P2P networks (unstructured, DHT, super-peer, etc.). This allows us to exploit continuing progress in such systems. To deal with semantically rich data, APPA supports decentralized schema management. To capitalize on Web service standards, the shared data are in XML format (which may be interfaced with many data management systems). In APPA, we propose an optimistic replication solution which provides eventual consistency and other useful properties found in distributed systems. To the best of our knowledge, there is no replication solution which assures eventual consistency in P2P environments. We exploit application semantics to reduce conflicts during reconciliation. We also propose a schema-based query processing strategy that deals with replication in a way that reduces redundant work and optimizes peer load balancing.

The rest of the paper is organized as follows. Section 2 describes the APPA architecture. Section 3 introduces our replication solution. Section 4 introduces our query processing strategy. Section 5 concludes an discusses implementation issues.

2. APPA Architecture

APPA has a layered service-based architecture. Besides the traditional advantages of using services (encapsulation, reuse, portability, etc.), APPA is network-independent so it can be implemented over different P2P networks (unstructured, DHT, super-peer, etc.). 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 unstructured P2P systems 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 all P2P networks:

- **Peer id assignment**: assigns a unique id to a joining peer using a specific method, e.g. a combination of super-peer id and counter in a super-peer network.
- **Peer linking**: links a joining peer to some other peers, e.g. by setting neighbors in an unstructured network, by locating a zone in CAN [21], etc.

- **Key-based storage and retrieval**: stores and retrieves a (key, value) pair in the P2P network, e.g. through hashing over all peers in DHT networks or using super-peers for storage and retrieval in super-peer networks.

Basic services. This layer provides services for peer management and communication over the network layer:

- **P2P data management**: stores and retrieves P2P data (e.g. meta-data, index data) by key in the P2P network.
- **Peer management**: provides support for peer joining (and rejoining) and for storage, retrieval and removal of peer ids.
- **Peer communication**: enables peers to exchange messages (i.e. service calls) even with disconnected peers using a persistent message queue.
- **Group membership management**: allows peers to join an abstract group, become members of the group and send and receive membership notifications. This is similar but much weaker than group communication [6].
- **Consensus module**: allows a given set of peers to reach agreement on a common value despite failures.

Advanced services. This layer provides advanced services for semantically rich data sharing including schema management, replication, query processing, security, etc. using the basic services. In addition, we assume each peer has data management capabilities (e.g. a DBMS) for managing its local XML data, possibly through a traditional wrapper interface.

Figure 1 shows an APPA architecture based on a DHT network. In this case, the three service layers are completely distributed over all peers. Thus, each peer needs to manage P2P data in addition to its local data.

![Figure 1: APPA architecture with DHT](image)

Figure 2 shows an APPA architecture based on a super-peer network. In this case, super-peers provide P2P network services and basic services while peers provide only the advanced services.

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 [25]. Thus the main problem is to support decentralized schema mapping. Several solutions have already been proposed. In PeerDB [15], assuming an unstructured network, schema mapping is done on the fly during query processing using IR techniques. In Edutella [14], RDF-based schema descriptions are provided by super-peers. Piazza [25] proposes a more general, network-independent, solution that supports a graph of pair-wise
mappings between heterogeneous schema peers. Although we plan to study this solution for APPA, we use a simpler one that takes advantage of the collaborative nature of the applications we target. We assume that peers that wish to cooperate, e.g. for the duration of an experiment, are likely to agree on a Common Schema Description (CSD). Our experience with scientific applications taught us this assumption is realistic [24]. Given a CSD, a peer schema can be specified using views. This is similar to the local-as-view approach in data integration [13] except that, in APPA, queries at a peer are expressed against the views, not the CSD.

In this context, we can assume that the P2P system is self-permanently (crash failures). In addition, communication links can fail temporarily (transient faults) or unpredictably. In this section, we define our replication model with our assumptions regarding the P2P environment. Then, we discuss log-based reconciliation, a general optimistic replication solution designed for mobile and distributed environments, and the problems introduced to adapt it in P2P environments. Finally, we introduce our optimistic replication protocol and the architecture of APPA’s replication service.

3.1 Replication model

We assume that the network is subject to frequent and unpredictable changes: peers can join or leave the system arbitrarily often, and they can fail temporarily (transient faults) or permanently (crash failures). In addition, communication links can experience transient failures (e.g. message loss).

To deal with advanced collaborative applications, we follow the small world assumption [10] which allows us to deal with groups of peers of manageable size. Although the entire P2P system may have a very high number of peers, the small world assumption exploits the group locality and time locality characteristics:

- **Group locality**: users tend to work in groups. A group of users, although not always located in geographical proximity, tends to use the same set of resources (e.g. files).
- **Time locality**: the same user may request the same data multiple times within short time intervals.

In this context, we can assume that the P2P system is self-organized [3]: the system should force peers to be logically adjacent with respect to the content they replicate. This yields fast communication between peers of the same group. It is also useful to allow a peer to belong to several groups.

Our replication model is based on the lazy multi-master scheme of distributed databases which we transpose here to P2P. With lazy replication [18], a peer (called master peer) can update a replica. Afterwards, the update is propagated to the other peers that hold a replica which get updated (refreshed) separately. In mono-master mode, the peers that receive the refresh updates are slaves in read-only mode. Hence, the property of mutual consistency is relaxed and strong consistency is eventually assured. Although quite effective in many applications, lazy mono-master replication is not appropriate in the P2P context since the master peer is a single point of failure and a potential bottleneck. Multi-master replication allows multiple peers to update the same replicas, thus improving availability and performance. However, conflicting updates of the same data at different peers can introduce replica divergence. To solve this problem, we use log-based reconciliation.

For simplicity, we assume full reconciliation. A replica $r$ may be a file or a relation represented as an XML document. Let $g(R,P)$ define a group where $R$ is a set of replicas and $P$ is a set of peers, full reconciliation states that each $p$ in $P$ can update each $r$ in $R$.

3.2 Log-based reconciliation

The solution we consider to manage multi-master replication is log-based reconciliation [20]. In this solution, each peer locally executes tentative actions which respect its static and dynamic constraints, and, additionally, records these actions in a replication log. Periodically, all the logs are merged by a central peer in order to define a global schedule (an interleaved order of operations from different peers) respecting all defined constraints.

In the following, we recall the main terms employed:

- **Action**: an action is a tentative update operation executed by an application program on a replica. For instance, this can be a write operation in a file or document, or a database transaction.
- **Static constraint**: defines conditions on action ordering (ex.: an update cannot follow a delete) in order to restrict the search space of the schedule when merging logs.
- **Dynamic constraint**: verifies the success of a single action depending on the current database state, as in traditional databases.

This solution assures eventual consistency among replicas because it preserves the following properties, as defined in [22]:

- **Eventual Consistency**: a system is eventually consistent if when all clients stop submitting actions (and constraints), then eventually all peers will reach the same value (and not change thereafter).
- **Mergeability**: a system has the mergeability property if it is possible to schedule any arbitrary collection of operations of their logs respecting their constraints.
- **Eventual Decision**: every submitted action is eventually decided. An action is said decided once major decisions regarding its execution have been made (for instance, it is always non-executed, or it is always executed and its order is known).
- **Eventual Action Propagation**: the system ensures that the actions known at peer $p$ at time $t$, are eventually known by an arbitrary peer of the group.
• **Eventual Constraint Propagation**: similar to Eventual Action Propagation wrt. constraints. They are known together as Eventual Propagation properties.

• **Safe decisions**: peers may not make conflicting decisions.

This solution is quite good for mobile systems but not entirely well suited for P2P systems because of the centralization of all decisions at a single peer to guarantee the above properties. Furthermore, we must consider the case of peers joining a group, which introduces new constraints which may contradict recent decisions, or peers leaving a group which implies that some constraints are dropped, which may impact previously taken decisions. As a consequence, the Eventual Consistency and Eventual Decision properties may be violated because previously taken decisions may be undone due to the conflicts introduced by the incoming constraints.

### 3.3 Optimistic replication in APPA

In this section, we outline our multi-master optimistic replication solution. A good approach to solve the problem of scalable self-managed applications is by providing a complete architecture that combines several P2P services and known classical distributed computing solutions [4].

Our solution works as follows. A group of peers that collaborate holds a set of common global actions \( \{a_1, a_2, a_3, \ldots \} \) and constraints \( \{c_1, c_2, \ldots \} \). The semantics of the actions and constraints depends on the application. For instance, an agenda application could recommend scheduling all appointment cancellation actions before appointment record actions. In addition, each peer may introduce local actions and constraints that do not conflict with the global ones. For instance, a peer could define a local action, e.g. an appointment change, as a group of actions containing an appointment cancellation followed by an appointment record by an appointment record. A replica \( r \) is updated by local or global actions when these actions do not violate the set of local and global constraints. Note that an action corresponds to the definition of an update operation and an action instance is its execution by a peer using particular parameters. For simplicity, we use the term action in both cases.

Whenever a peer \( p \) updates a local replica, it stores the correspondent tentative action (henceforth action) in its local data store. In addition, \( p \) stores its actions in an action queue within the P2P system to guarantee that it will be available to all peers involved, even those that may be disconnected.

Using the consensus service, the peers of the involved group agree on a time interval \( \Delta_t \) in which actions stored in the action queue are grouped together to form a schedule unit. Hence, a **schedule unit** holds an unordered set of actions that may have been produced by any peer during a time interval \( \Delta t \).

Figure 3 shows an example of three peers that execute each a sequence of actions \( a_1 \to a_2 \to a_3 \). We note \( a_i^j \) the action \( a_i \) executed by peer \( p_j \). The actions produced by each peer are stored in the action-queue following the execution order of \( a_i^j \). The established time interval to form a schedule unit \( s_i \) is \( \Delta_t \). Therefore using the action queue contents, three schedule units \( (s_1, s_2, s_3) \) are defined. The sequence \( s_1 \to s_2 \to s_3 \) and the last schedule unit in the sequence, \( s_3 \), are said incomplete because the time interval \( \Delta_t \) did not expire, hence any peer can still store actions, whereas \( s_1 \) and \( s_2 \) are said complete and do not accept new incoming actions.

Whenever a peer \( p \), wishes to reconcile its actions wrt. the other peers’ actions, it executes locally the reconciliation algorithm using each sequence of available complete and unordered schedule units as input parameter. The reconciliation algorithm orders actions within each schedule unit. Again, each ordered schedule unit is stored by peer \( p \) in the P2P system, using the P2P data management service, to guarantee that it will be available to any peer in the group, even those that may be eventually disconnected. Finally the effects of the ordered schedule units are applied by peer \( p \) to its persistent data.

**Figure 3: Merging peer actions through schedule units**

Using the communication service, peers may be aware of which schedule unit are being ordered and choose to wait to avoid redoing on-going reconciliation, thereby lightening the peer load. In addition, we consider that whenever a peer leaves a group, this event is notified using the group membership service in order to possibly remove all related actions from schedule units and also to possibly re-order some ordered units.

Our solution guarantees eventual consistency among replicas by assuring mergeability, eventual decision, eventual propagation and safe decisions properties. It is completely distributed and respects the autonomy of peers to join and leave the group, and to introduce new actions and constraints. Furthermore, information about replicas is systematically captured and can be exploited for other services, e.g. query processing.

### 3.4 Replication architecture

Our replication architecture in APPA has three advanced services (see Figure 4): log manager, schedule unit manager, and reconciler. Furthermore, in our solution, global actions and constraints are stored together with common schema definitions (CSD) whereas local actions are stored at each peer.

- **Log manager**: this service extracts actions executed by peer \( p \) on its local persistent data and stores them in the log following the execution order.

- **Schedule unit manager**: interacts with the other schedule unit managers, using the consensus service, to define the schedule unit time interval. In addition the schedule unit manager reads the contents of the log and stores it in the action queue (in the P2P persistent store).

- **Reconciler**: this service orders actions that belong to complete schedule units, applies these actions on local persistent data and make the ordered schedule units available to all involved peer, using the P2P data management service.
of peer mapping at peer system, the problem is to find assume conjunctive queries. Let
relation. Given 2 CSD relation definitions Thus, a peer schema includes peer mappings, one per local
XML) and the Datalog-like notation of [25] for mapping rules.
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 [25] 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:

\[ p : \text{rel}(A,B,D) \subseteq \text{csd} : \text{rel}(A,B,C), \text{csd} : \text{rel}(C,D,E) \]

In APPA, the peer schemata are stored as P2P data using the key-
based storage and retrieval (KSR) module, where the key is a
combination of attribute and relation.

Query processing proceeds in four main phases: (1) query
reformulation, (2) query matching, (3) query optimization and (4)
query decomposition and execution. For each phase, we give a
simple solution although there is much room for improvement.

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 \(p\):

\[ \text{select} A,D \text{ from } r \text{ where } B=b \]

would be rewritten on the CSD relations as:

\[ \text{select} A,D \text{ from } r_1,r_2 \text{ where } B=b \text{ and } R_1 \cap R_2 \]

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 peer schema
definition. Using the KSR service on P2P data, we can iteratively
retrieve these peers. Let \(R\) be the set of relations involved in \(Q\),
and \(ps(p,r)\) denote that the peer schema of peer \(p\) involves relation
\(r\), query matching produces:

\[ P' = \{ p \mid p \in P \wedge \exists r \in R ps(p,r) \} \]

Query optimization. Because of data replication, some peers in
\(P'\) may also belong to groups and thus have replicated data. Thus,
the optimization objective is to avoid (or minimize) the amount of
redundant work that would be incurred in accessing replicas. For
instance, to avoid any redundant work, this would yield \(P'' \subseteq P'\)
such that, for any two peers in \(P''\), their relevant data are not
replicated. Given \(P'\) and \(Q\), the optimization algorithm proceeds
in three steps:

1. Determining peers with replicated data. Assuming that the
unit of replication is the relation, we use the replication service
and the KSR service to retrieve all groups which refer to relations
involved in \(Q\). Let \(G\) be the set of all groups and \(c(g,r)\) denote that
group \(g\) holds a replica (copy) of relation \(r\), this step produces the
set \(G'\) of all relevant groups:

\[ G' = \{ g \mid g \in G \wedge \exists r \in R c(g,r) \} \]

For instance, \(G' = \{ g_1(R_1,p_1,p_2), g_2(R_1,p_2,p_3) \} \).

2. Determining most relevant groups. Each group in \(G'\) may refer
to a subset of the relations in \(Q\). However, as shown in the
above example, the same peer \((p_2,3)\) may belong to several
relevant groups and thus have more relevant data. To determine the most
relevant groups, we produce a set \(G''\) of new temporary groups
by iteratively combining the initial groups of \(G'\) by 2, 3, etc. until
a single group can be formed with all groups. Given two groups
\(g_1(R_1,p_1)\) and \(g_2(R_2,p_2)\), we can produce a new group
\(g_3(R_1 \cup R_2 : p_1 \cap p_2)\), for instance, \(g_3(R_1,p_2,p_3)\) for the example above.

3. Selecting best peer(s) per group. For simplicity, we assume
\(G''\) is not empty and the union of all relations in its group is a
superset of \(R\) (the set of relations in \(Q\)). Relaxing this assumption
requires dealing with \(P''\) and \(G''\) as well. Thus we can restrict
ourselves to the groups in \(G''\) with the largest number of relevant
relations. We want to minimize the amount of redundant work
that would be incurred by querying each peer in each such
group. First, we use the peer linking service to identify in each
group the peers that are disconnected and remove them. Second,
for each group, we select one or more candidate peers based on a
cost function. Selecting more than one candidate peer is useful in
a very dynamic environment since, at the time of the next
execution phase, 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. Studying this
trade-off in different application contexts is the subject of future
work. The cost function can also be sophisticated and take into
account communication cost, peer’s load, etc. in order to optimize
peer load balancing. This step produces a set \(P''\) of best peers.

The cost of this optimization algorithm is relatively low. There
are \(O(\text{card}(R)+\text{card}(P''))\) calls to the KSR service (in steps 1 and
3) and each call incurs a number of messages which depends on
the P2P network, e.g. \(O(1)\) in some super-peer and \(O(\log n)\) with
peers in DHT. Furthermore, processing these calls can be
parallelized.

Query decomposition and execution. This phase is similar to
that in data integration systems and we can reuse well-know, yet
sophisticated techniques. Since some peers in $P^{-}^0$ may have only subsets of relations in $R$, 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 [26]. Finally, $Q$ is sent to each peer in $P^{-}^0$ which (if connected) reformulates it on its local schema (using the peer mappings), executes it and sends back the results to the sending peer which integrates the results. Result composition could also be parallelized by using intermediate peers.

5. Conclusion
In this paper, we have addressed replication and query processing in the context of the APPA data management system. The paper has several contributions. First, APPA has a network-independent that can be implemented over different P2P networks (unstructured, DHT, super-peer, etc.), thus allowing us to exploit continuing progress in such systems. Second, we proposed an optimistic replication solution based on log reconciliation which provides eventual consistency and other useful properties found in distributed systems. To the best of our knowledge, there is no replication solution which assures eventual consistency in P2P environments. Finally, we proposed a schema-based query processing strategy that deals with replication in a way that reduces redundant work and can optimize peer load balancing.

We plan to implement APPA on structured (DHT) and super-peer networks to study various trade-offs and experiment with collaborative applications. We have started the super-peer implementation using Web standards such as XML, SOAP, Xschema, and Xquery. The techniques and solutions proposed in this paper are simple. Future work will include the relaxing of simplifying assumptions and their improvement.

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