

A Scalable XML P2P Query System

Giovanni Conforti Giorgio Ghelli Paolo Manghi Carlo Sartiani
Dipartimento di Informatica - Università di Pisa Largo B. Pontecorvo 3 Pisa, Italy
{confor,ghelli,manghi,sartiani}@di.unipi.it

ABSTRACT

This paper presents the architecture and the self-management algorithms of XPeer, a p2p XML database system. Unlike existing p2p systems, XPeer is capable of self-organizing its administrative layers, so to adapt to changes in the network topology and in the workload.

The architecture of XPeer is based on two innovative concepts: the presence of two distinct subsystems (which we call *overlays*), devoted, respectively, to the management of queries and of schema update requests; and the use of *cloning* in order to distribute load among the administrative peers. By exploiting these key features, the actual workload processing power of XPeer can scale linearly in the number of peers in the system.

1. INTRODUCTION

Peer-to-peer is a term used with different meanings. We use it for a distributed system where every node is both a client and a server, and where nodes can freely join and freely leave the system. This implies that, if some peers perform special administrative task, the system must be able to dynamically substitute them, whenever they just “go away”. The potential availability of many servers, and the tolerance to any sudden server disconnection, makes this architecture a good foundation for systems which may be extremely robust and scalable.

The huge popularity of p2p systems is mainly due the diffusion of some *file-sharing* and *file-transfer* protocols, which proved that such systems can actually be efficient and robust in face of very high volatility. However, such systems are extremely limited in the kind of queries they can support.

Much of the research on the construction of real p2p data bases is currently aimed at the p2p decentralization of data integration mediators (e.g., Piazza [8], and CoDB [6]). These systems, usually based on the GLAV paradigm [7], enable each peer to reformulate queries according to a given set of mappings, with no need of centralized mediator. Schema

translation is crucial in many application fields. However, the need to set up the translation implies that some human administrative work is needed in order to join such systems.

Other systems, like XPeer [12], address application fields where schema integration is not an issue, such as communities where a well-known common schema is exploited, or situations where nobody is going to define a schema mapping anyway, hence any query has to be exploratory. In this context, dynamicity and scalability are the central concerns.

Our Contribution. This paper studies the working cost as well as the scalability properties of the XPeer p2p XML query system. XPeer supports a non-trivial query language, allowing the user to formulate queries in the FLWR core of XQuery [5], and its architecture can be used in any system supporting lookups on multiple keys or more complex queries.

The system is characterized by the presence of two distinct overlays (subsystems), one handling query requests, and the other managing update requests. These overlays communicate through periodic synchronization operations, and manage themselves with no need of human intervention. We study the cost of XPeer operations by first defining a general model for p2p systems, and then by using the model to describe a family of systems incorporating more and more features of XPeer. In particular, we introduce two novel techniques for organizing p2p systems: the *cloning* of peers participating in an overlay, and the presence of two distinct and functionally different overlays, and we show how the loose connection among the two overlays allows XPeer management protocols to scale up to any number of peers.

Paper Outline. The paper is structured as follows. Section 2 illustrates the model we use for studying the query and update processing cost. Section 3 describes the XPeer system and analyzes its working cost and scalability; we proceed *step-by-step* by enriching a basic system with new features till the complete XPeer system is formed. Section 4 discusses some related work. Section 5 concludes.

2. SYSTEM MODEL

We represent a p2p system as a set of interconnected peers $\mathcal{P} = \{p_1, \dots, p_n\}$. Each peer p_i manages a piece of XML data instance, as well as a schematic description of the data. The schema language of choice is XML Schema, but it can be any other language, provided that it meets two requirements: schema *selectivity* and schema *brevity*. *Selectivity* refers to the main application of schemas in XPeer: XPeer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

matches each query against a collection of peer schemas to find peers with potentially interesting data. A schema is *selective* if, when it matches a query, there is a high probability that the query will match the described data. Schemas can be made more selective by, for example, decorating a schema leaf with the set of literals that the peer stores in the corresponding leaf of the XML instance. The administrative peers match each query against a data structure that represents the schema of every peer in the system. *Brevity* refers to the fact that this data structure has to be small enough to fit, ideally, in main memory, or, at least, to occupy a limited amount of disk space (XPeer stores schemas in persistent store). Brevity holds for sure if schemas contain structural information only, but extensional information, such as the leaf literals, may lead to a significant size increase, so that a trade-off is necessary.

Each peer may participate to one or more *overlay networks*, which is our name for a connected subset of peers that perform an “administrative” task.

In our model, we assume that query compilation and query execution are not interleaved, as in many p2p systems, but clearly separated. To solve a query, a peer first locally translates it into an *incomplete query plan*, i.e., an algebraic representation of the query where information about data location is omitted. Then, the peer submits this incomplete plan to the *query overlay*, which matches the plan against peer schemas, and completes the plan with data location information. The query overlay then sends the plan back to the originating peer, which directly contacts the peers in the access plan and coordinates the execution of the query. This approach minimizes the load imposed on the query overlay, which is not involved in query execution, hence has no intermediate query results to forward or to store.

In this paper we completely abstract away from a specific query language, hence the solutions we are proposing can be applied to any p2p system having a non-trivial query language, e.g., a language allowing lookups over multiple keys or more complex queries. In particular, in the XPeer system we use a subset of the FLWR core of XQuery [5], as described in [11].

In the rest of the paper, we will use the following notation:

N_p	\triangleq	number of peers in the system;
e	\triangleq	number of queries in the time unit;
u	\triangleq	number of updates in the time unit;
t	\triangleq	total number of operations in the time unit;
W	\triangleq	workload of t operations, comprising e queries and u updates.

For the sake of simplicity, we assume that all peers have similar computational power, and we use k_{max} to denote the maximum number of messages that a peer p_i can *efficiently* manage.

Finally, we will use, as primary cost measure, the total number of incoming and outgoing messages processed by a generic administrative peer p_i during the processing of a workload W . We will compare the peer load with k_{max} in order to find the operational conditions of the system.

3. XPEER

XPeer is an *unstructured* p2p query system for XML data. XPeer consists of a collection of *autonomous* peers, sharing

possibly *heterogeneous* data described by (possibly) inferred schema information.

Some of its peers participate to the *query overlay*, a subsystem that matches queries against peers schemas to decide which peers each query should be addressed to. Some of the peers participate to the *update overlay*, which manages peer connection, disconnection, and schema update.

The two overlays may coincide, as in usual p2p systems. In the following sections, we will show three versions of XPeer, the first with a single overlay, and the others with distinct overlays. These versions allow us to incrementally introduce new features, and to separately study the effect of each.

XPeer is able to adapt its organization to changes in the workload or in the topology; the self-organization process is performed by expanding or contracting the system overlays, and is guided by load equations.

3.1 Cloned Single-overlay XPeer

The *cloned single-overlay* system is formed by a single overlay, managing both queries and updates (the *query-update* overlay). To enhance system scalability, the overlay is not formed by a single peer, but comprises a set of peers, called *clones*, that host copies of the peers schemas and that are functionally identical. Peers send queries to a randomly chosen clone, which processes the query. Update requests are sent to a randomly chosen clone, which updates its schema information and propagates the update to the remaining clones.

Load Analysis

We will now compute the load of the administrative nodes, which, in this case, are just the *clones*. We want to discover the *operational conditions* of the system, that is, which is the maximum workload that the system can manage, and how the system must evolve when the load changes.

We measure the load of a generic clone in terms of exchanged messages. Every query requires a randomly chosen clone to receive a message and send an answer, hence the generic clone has a query load $\frac{2e}{d_c}$, where d_c is the number of clones. An update request is randomly routed to a clone, but it propagates the request to the other peers. Hence, every clone finally receives all u update requests in the interval and, once every d_c updates, the generic clones has to send out $d_c - 1$ propagation messages, and this gives the second and third addenda in the cost below. The protocol dictates that each peer tags its schema-update messages with consecutive version numbers. When a clone realizes it missed a version number in the series of the update messages that originate from a peer, the clone contacts the peer to rebuild the current schema version. This recovery activity imposes a further load of $2u\pi$ messages, where π denotes the average fraction of update messages that get lost.

Proposition 3.1 (Clone load) *The load of a clone during the processing of the workload W is given by:*

$$L(W) = \frac{2e}{d_c} + u + \frac{(u)(d_c - 1)}{d_c} + 2u\pi \sim \frac{2e}{d_c} + 2u$$

The approximation \sim is valid when the number of clones is high and $\pi \ll 1$; when $d_c = 1$, the update cost is $(1 + 2\pi)u$.

Remark: Query matching and schema update may actually cost more than message processing. This can be easily ac-

counted by generalizing the formula above to $\frac{c_e e}{d_c} + c_u u + \frac{c_f u(d_c - 1)}{d_c} + c_s u \pi$, where c_e , c_u , c_f , and c_s are the relative costs of query matching, schema update, update forwarding, and schema resynchronization. This generalization would not significantly affect our results, hence we stay with the traditional approach, in order to avoid constant proliferation. For the same reason, we will hereafter ignore the addendum $2u\pi$, not only because it is dominated by the others, but also because adding $2u\pi$ is equivalent to applying a multiplicative factor $(1 + 2\pi)$ to the u addendum.

Proposition 3.1 shows that, as expected, if the update load is low, an increase in the query load can be matched by a proportional increase in the number of clones d_c . On the other side, cloning does not help sharing the update load. The non-approximate form of the equation shows that an increase of d_c actually *increases* the update-induced load, because of the cost of update forwarding. The *marginal variation* of the clone load w.r.t. d_c shows immediately that an increase of d_c is convenient if, and only if, queries are more than updates; more precisely, if $2e > u$.

Proposition 3.2 *The marginal variation of the clone load wrt d_c in the CSO system is given by:*

$$\Delta(W) = \frac{-2e}{d_c(d_c + 1)} + \frac{u}{d_c(d_c + 1)}$$

Corollary 3.3 *In the CSO system, cloning is convenient when: $u \leq 2e$.*

Assuming that each clone can process at most k_{max} messages before significantly reducing its performance, we can now find the system operating conditions.

It is convenient to distinguish the query dominated case $u \leq 2e$ (i) from the update dominated case $u \geq 2e$ (ii).

In the query dominated case, the system is operational until the clone load is lower than k_{max} :

$$\begin{aligned} \frac{2e}{d_c} + u + \frac{u(d_c - 1)}{d_c} &\leq k_{max} && \Leftrightarrow \\ 2e + u(d_c) + u(d_c - 1) &\leq k_{max}(d_c) && \Leftrightarrow \\ (k_{max} - 2u)d_c &\geq 2e - u \end{aligned}$$

Now, we have to consider cases $k_{max} - 2u > 0$ and $k_{max} - 2u \leq 0$. A simple computation shows that no solution exists in the second case, hence we impose that $k_{max} - 2u > 0$ (i.1) and obtain the condition $d_c \geq \frac{2e - u}{k_{max} - 2u}$, which specifies the minimal amount of clones needed to process the workload, under assumptions (i) and (i.1). This condition on d_c can be satisfied if, and only if, the right hand side is smaller than N_p , since we cannot have more clones than peers:

$$\begin{aligned} \frac{2e - u}{k_{max} - 2u} &\leq N_p && \Leftrightarrow \text{(by (i.1))} \\ 2e - u &\leq N_p(k_{max} - 2u) && \Leftrightarrow \\ \frac{2e - u}{N_p} + 2u &\leq k_{max} \end{aligned} \quad (i.2)$$

By assumption (i), (i.2) subsumes (i.1), hence we obtain our first result: when $u \leq 2e$, the system can work iff $\frac{2e - u}{N_p} + 2u \leq k_{max}$, and in this case the following relation identifies the correct values for d_c , and it admits some solutions:

$$\frac{2e - u}{k_{max} - 2u} \leq d_c \leq N_p \quad (3.1)$$

Case (ii) is simpler. In this case, if any solution exists, then $d_c = 1$ is a solution, and is the optimal one. Hence, by

substituting $d_c = 1$ into the equation of Proposition 3.1, we obtain that the system can work iff $2e + u \leq k_{max}$. Hence, we have our second result: when $u \geq 2e$, the system can work iff $2e + u \leq k_{max}$, and in this case we can set $d_c = 1$.

Observe that, by (ii), $2e + u \leq k_{max} \Rightarrow 2u \leq k_{max}$, which means that, in both cases (i) and (ii), the operating condition can be somehow approximated with $2u \leq k_{max}$, which means that, as expected, the system is operational until the update processing load $2u$ exceeds the capacity of a single node.

Self-administration

The CSO system can adapt its configuration to changes in the workload, both in the total number of operations and in the query/update ratio, by modifying the number of clones deployed to process queries and updates. To illustrate the self-administration algorithms, we exemplify the evolution of the system starting from the bootstrapping.

The initial configuration of the system comprises a single clone. Until $u > 2e$, the clone will not try and create new clones, since this is not going to give any benefit. If, instead, $u < 2e$, when the load becomes close to k_{max} , the clone activates the cloning process.

Suppose now that the system, after some time, comprises d_c clones. When the total number of operations in the workload increases again, the clones must cooperatively start the evolutionary process, by relying on a distributed consensus algorithm, like, for instance, Paxos [9]; if $u - 2e < 0$, then a new clone is introduced; instead, if $u - 2e > 0$, then only a decrease in the number of clones can reduce the clone load, hence clones choose one clone to dismiss.

Similar considerations apply when the ratio between queries and updates changes. If the number of queries increase, and $u - 2e < 0$, then clones may decide to introduce a new clone; on the contrary, when the number of updates increases, and $u - 2e > 0$, then the dismissal of one or more clones may be necessary.

In order to avoid dangerous oscillations between clone creation and dismissal, due to situations where u is near to $2e$, each operation of clone creation inhibits clone dismissal for a period of time. The strong dependency of self-administration on the condition $u - 2e < 0$ is unpleasant, and will be solved in the next section.

3.2 Basic Dual-overlay XPeer

We have seen that clones help reducing the query load for each clone, but they actually increase the update load for clone, because of synchronization. Hence, our first attempt to increase the system scalability is based on splitting the overlay into a query overlay, where nodes are cloned, and an update overlay, where cloning is not used.

In this way, we obtain the *basic dual-overlay* XPeer (BDO in the following). The query overlay is formed by a set of clones, as for the CSO system, while the update overlay consists of a single super-peer (the *root*), which receives all the update requests and periodically, β times every time interval, sends the collected update information to all the clones. By decoupling updates from queries, we substitute the clone synchronization cost with the cost of receiving β schema refresh messages in each time unit. While the synchronization cost in the CSO system increases with the number of clones, in the BDO system the update cost is constant for the clones. Of course, a high update rate can still affect the root, but

this problem will be solved in the next section.

Load Analysis

We first present the load equations for clones and root.

Proposition 3.4 (Clone load) *The load of a clone in the query overlay during the processing of the workload W , is given by:*

$$L(c, W) = \frac{2e}{d_c} + \beta$$

Proposition 3.5 (Root load) *The load of the root in the update overlay, during the processing of the workload W , is given by:*

$$L(\text{root}, W) = u + \beta \cdot \text{broadcast}(d_c)$$

The first addendum describes the number of update requests that reach the root (u), and the second is the schema refresh load. $\text{broadcast}(d_c)$ is the cost of broadcasting the same message to d_c destinations. This cost can be easily reduced to a constant, even in a point-to-point network, by connecting the root to h helper nodes, each with a further fan-out of h helper nodes, reaching the clones in $\log_h(d_c)$ steps. Hereafter, we will only assume that $1 \leq \text{broadcast}(d_c) \leq d_c$.

Before starting the analysis of the operational conditions of the system, some words about β are necessary. β is automatically set by the root of the update overlay so that $\beta < u$ and $\beta < k_{max}$. The first relation says that the system *compacts* updates so to decrease the update processing load, at the price of a lesser degree of correctness for query results; the second relation is obvious, so we always assume that $k_{max} - \beta > 0$.

To study the behavior of the dual-overlay system, we must compare both the clone load and the super-peer load to the maximum node load (e.g., k_{max}). The load equations imply that the system can work iff the following relations hold.

$$\begin{cases} \frac{2e}{d_c} + \beta \leq k_{max} \\ u + \text{broadcast}(d_c)\beta \leq k_{max} \end{cases}$$

If we assume that every peer may become a clone when needed, the first condition can be rewritten as follows:

$$\frac{2e}{N_p} + \beta \leq k_{max}$$

Since $\frac{e}{N_p}$ is the average number of queries submitted by a peer, it can be assumed to be far less than k_{max} . β is chosen by the root, and can be also chosen to be far less than k_{max} . Hence, the condition above says that the query overlay, if allowed to freely expand, is never going to be overloaded. The second condition is much more problematic. The second addendum is not really a problem, since both $\text{broadcast}(d_c)$ and β can be made quite small. However, u grows linearly with the number of peers, hence it will eventually overcome k_{max} . This shows that this architecture is still not ready to deal with arbitrary size systems.

Self-administration

Self-administration policies of the BDO system are more complex than those of the CSO system, because of the presence of two correlated parameters, β and d_c , that appear both in the clone load and in root load equations.

Since the system features two distinct overlays, they react independently to changes in query load or in the update load. If the query load increases, so that $L(c, W)$ becomes too close to k_{max} , then clones immediately introduce a new clone, which lowers their load. When $L(c, W)$ goes below a k_{min} threshold, a clone is dismissed; k_{min} is very far from k_{max} in order to avoid any oscillating behavior. The creation and dismissal of new clones is governed by the clones themselves through a distributed consensus algorithm, so that all clones always know all other clones (we use Paxos for this purpose [9]). This is useful because the same algorithm is used by the clones to substitute a failed root, and Paxos requires a reliable knowledge of the electoral base.

When the update load increase, the root reacts by modifying β according to the equation $\beta = \min(u, \frac{k_{max}-u}{\text{broadcast}(d_c)})$. This equation implies that $\beta \leq u$, which ensures that the root sends out no more update messages than it receives. When the update load is high, i.e., when $u \geq \frac{k_{max}}{1+\text{broadcast}(d_c)}$, the system will operate with $\beta = \frac{k_{max}-u}{\text{broadcast}(d_c)}$, which is the highest value of β that is compatible with efficient root operation. Finally, when these equations impose the root to increase β , it is always increased very slowly, in order to give time to the query overlay to generate new clones if needed.

The creation of a new clone in general increases the root load (Proposition 3.5). As a consequence, if $L(\text{root}, W)$ is operating with $\beta = \frac{k_{max}-u}{\text{broadcast}(d_c)}$, β will decrease, which will marginally reduce the clone load. This is not going to cause any oscillation in the number of clones, however, because of the distance between the dismissal threshold k_{min} and the typical clone load k_{max} .

The combination of $\beta \leq u$ and $\beta \leq \frac{k_{max}-u}{\text{broadcast}(d_c)}$ implies that $\beta \leq \frac{k_{max}}{2}$, hence that, even in situation with a very high update load, clones will spend most of their time processing queries rather than updates, which is the main advantage of this architecture over the previous one. On the update side, the equation $\beta = \frac{k_{max}-u}{\text{broadcast}(d_c)}$, implies that the update overlay can be protected by the effects of a high query load by a clever implementation of broadcast communications. However, as already observed, the update overlay has no protection against situations where $u > k_{max}$.

3.3 Nested Dual-overlay XPeer

The scalability of the BDO system on the query side is excellent, but its scalability for update requests is limited by the maximum load that the root of the update overlay can sustain. To overcome this issue, we finally introduce the *nested* dual-overlay XPeer. As in the basic dual-overlay XPeer, updates are decoupled from queries, and are managed by a proper subset of peers. However, this time the update overlay is organized as a two-level tree: each leaf (*super-peer*, in the following) collects update requests coming from a subset of the peers and propagates them to the root, while the root collects requests from the super-peers and propagates them to the query overlay. In both cases, updates are only propagated periodically, in bunches. This structure allows the system to push its operational conditions far beyond those of the other systems.

Load Analysis

We first determine the load of clones, root, and super-peers. The load of clones in the query overlay is the same as for

the basic dual-overlay system, while the load of nodes in the update overlay significantly changes. (For the sake of simplicity, we assume that the update propagation periods of super-peers to root and root to clones are the same, and we ignore here the issue of lost messages.)

Proposition 3.6 (Super-peer load) *The load of super-peers in the update overlay, during the processing of the workload W , is given by:*

$$L(sp, W) = \frac{u}{d_s} + \beta$$

where d_s is the number of super-peers in the update overlay, and β is the number of update propagation messages in the time unit.

The first addendum is the cost of receiving all the updates of a $\frac{1}{d_s}$ fraction of the peers. The second is the cost of sending β messages to the root in the time unit.

Proposition 3.7 (Root load) *The load of the root in the update overlay, during the processing of the workload W , is given by:*

$$L(root, W) = \beta d_s + \beta \cdot \text{broadcast}(d_c)$$

The first addendum is the cost of receiving β update requests from each super-peer, and the second is the cost of distributing schema refresh messages to the clones.

By comparing the load equations with k_{max} , we can identify the minimum number of clones and super-peers needed for a given workload.

Proposition 3.8 *The minimum number of clones necessary for processing a workload $W = \{q_1, \dots, q_e, u_{e+1}, \dots, u_t\}$ is given by:*

$$d_c \geq \frac{2e}{k_{max} - \beta}$$

Proposition 3.9 *The minimum number of super-peers necessary for processing a workload $W = \{q_1, \dots, q_e, u_{e+1}, \dots, u_t\}$ is given by:*

$$d_s \geq \frac{u}{k_{max} - \beta}$$

By comparing the load equations with k_{max} , we can identify the conditions under which the system works properly.

Proposition 3.10 *The nested dual-overlay system works properly when:*

$$\begin{aligned} \frac{u}{d_s} + \beta &\leq k_{max} && (\text{super-peers}) \\ \beta d_s + \beta(\text{broadcast}(d_c)) &\leq k_{max} && (\text{root}) \\ \frac{2e}{d_c} + \beta &\leq k_{max} && (\text{clones}) \end{aligned}$$

The second relation can always be satisfied, provided that the value for β is chosen low enough. However, in a huge system, a high value of d_s may force β to be too low. This could be solved by augmenting the height of the tree; this would be a minor variation.

If we assume that every peer can also be a clone and a super-peer, the first and third conditions can be rewritten as follows (of course, if each peer can play three roles at a time, the value of k_{max} would decrease correspondingly):

$$\begin{aligned} \frac{u}{N_p} + \beta &\leq k_{max} && (\text{super-peers}) \\ \frac{2e}{N_p} + \beta &\leq k_{max} && (\text{clones}) \end{aligned}$$

The fractions $\frac{u}{N_p}$ and $\frac{e}{N_p}$ correspond to the average number of update and query requests submitted by each peer, hence can be assumed to be much smaller than k_{max} ; β can be kept much smaller than k_{max} as well, hence super-peers and clones are never going to be overloaded.

These relations show that the NDO system pushes the operational conditions till the theoretical limit $O(N_p \cdot k_{max})$ of a system of N_p peers. This is obtained by decoupling queries and updates, and by deferring update propagation and schema refresh.

Self-administration

Self-administration policies for the NDO system generalize those of the BDO system to deal with the richer structure of the update overlay. The update overlay, indeed, must set the values of d_s and of β , which, in turn, is influenced by the value of d_c chosen by the query overlay.

When the query load changes, the clones in the query overlay behave just like in BDO system. The super-peers adopt a similar policy: when $\frac{u}{d_s} + \beta$ approaches k_{max} , new super-peers are generated. Super-peer dismissal threshold k_{min} is very low, to avoid oscillations.

The value of β is set by the update overlay root to

$$\min\left(u, \frac{k_{max}}{d_s + \text{broadcast}(d_c)}\right).$$

As for the BDO system, β will never be greater than u for obvious reasons, and is kept below $\frac{k_{max}}{d_s + \text{broadcast}(d_c)}$, in order to allow the root to work. As in the previous case, β is always lower than $\frac{k_{max}}{2}$, and is usually much lower than that. For this reason, β is a minor addendum in the load equations of clones and super-peers, hence the root is allowed to modify it as needed. However, while β can be abruptly decreased, it is always increased slowly, in order to allow new clones and super-peers to be created, if needed.

Although decision taken by one overlay affect those which will be taken by another, the self-adaptation algorithms are quite stable, partly because of the distance between k_{min} and k_{max} , and partly because the effects of a change on the update load are mostly confined to the update overlay, while changes on the query load are mostly confined to the query overlay.

4. RELATED WORK

We briefly compare XPeer with three notable p2p data sharing systems: Piazza [8], Maier's system [10], and KadoP [4]. These systems are representative of unstructured and structured data sharing systems. For an ample review of p2p database systems, we refer the reader to [2].

Piazza. Piazza is a decentralized data integration system for XML data. The system is formed by a network of loosely interconnected and autonomous peers, which share XML data and schemas. Each peer is connected to a very limited set of neighbors (usually one or two) through a set of *schema mappings*, that are used for reformulating queries. Peer schema mappings are designed by a local administrator, which is also in charge of maintaining and updating local mappings. The presence of mappings limits the scope of Piazza to almost static environments, where the network topology changes very rarely. Query processing is based on a *flooding* algorithm that, even if optimized, severely limits

the scalability of the system.

Hence, Piazza focusses on sophisticated schema integration but does not address our core issues of high-dinamicity, zero administration, and query routing.

Maier's System. In [10], the authors describe a *coordinator-free* architecture for distributed XML query processing in the context of p2p systems. The architecture is based on two key ideas: *mutant query plans* (MQP), and *multi-hierarchic* namespaces. An MQP is a logical query plan, where leaf nodes may consist of URN/URL references, or of materialized XML data. An MQP traverses the system, carrying partial results and unevaluated sub-plans, until it is fully evaluated, i.e., it becomes a constant XML fragment.

MQPs are routed in the system according to information derived from multi-hierarchic namespaces. Indeed, authors assume that data contributed by peers are semantically connected, i.e., they are part of the same namespace, where a namespace is formed by several category hierarchies. This assumption makes the system not adequate when data are semantically heterogeneous. Moreover, since MQPs browse the network on the basis of namespace information, the number of messages required for query evaluation is comparable with that of flooding systems.

KadoP. KadoP is a p2p content sharing system based on the use of DHT tables (in the form of FreePastry [1]) and Active XML documents [3]. KadoP allows users to share XML documents, web services, as well as AXML documents (e.g., XML documents with embedded service calls); resource sharing is improved by the use of ontologies. Peers are organized in a DHT ring, where a distributed full-text index about documents and web services is stored; this index is used during query processing for locating interesting data and services. The presence of the index makes peer connections quite expensive, since newly shared documents must be indexed by the system; for the same reason, document updates, while automatically managed by service call-backs, may require expensive index updates, which imposes a clear limit to the volatility of the system. On the contrary, XPeer processes updates, both in the topology and in the data, with a constant number of messages.

Queries are processed in KadoP by (1) locating interesting resources on the network, and by (2) directly contacting involved peers, as happens with XPeer. Resource location requires the system to perform many key lookups in the DHT index, with a significant messaging cost, which depends on the dimension of the query, as well as on the data and services involved by the query. On the contrary, XPeer resolves source location which a simple peer-clone exchange.

5. CONCLUSIONS

This paper described XPeer, a p2p XML database system. The most distinctive features of XPeer are the presence of two functionally distinct overlays, specialized for query processing and update processing respectively, the presence of clones, i.e., administrative peers with fully replicated schema information, as well as the ability of the system to *self-configure* the overlays, so to adapt them to changes in the network topology and/or in the workload.

We presented the overall architecture of the system, analyzed the load of administrative peers during workload pro-

cessing, showed how the system uses load equations to self-manage its overlays, and studied the scalability properties of the system. In particular, we showed how the processing power of the nested dual-overlay system can scale linearly in the number of peers.

XPeer is mostly implemented and we are in the early stages of testing its performance and scalability.

6. REFERENCES

- [1] The FreePastry System. www.cs.rice.edu/cs/systems/pastry/freepastry/.
- [2] Sigmod Record, Volume 32, Number 3, 2003.
- [3] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active xml: A data-centric perspective on web services. In M. Levene and A. Pouloussis, editors, *Web Dynamics*, pages 275–300. Springer, 2004.
- [4] S. Abiteboul, I. Manolescu, and N. Preda. Sharing Content in Structured P2P Networks. Technical report, INRIA, Mar. 2005.
- [5] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, May 2003. W3C Working Draft.
- [6] E. Franconi, G. M. Kuper, A. Lopatenko, and I. Zaihrayeu. Queries and updates in the codb peer to peer database system. In *VLDB*, pages 1277–1280, 2004.
- [7] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational plans for data integration. In *Intelligent Information Integration*, 1999.
- [8] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *WWW2003*, 2003.
- [9] L. Lamport and M. Massa. Cheap paxos. In *DSN*, pages 307–314. IEEE Computer Society, 2004.
- [10] V. Papadimos, D. Maier, K. Tufté. Distributed Query Processing and Catalogs for Peer-to-Peer Systems. In: *CIDR 2003*, 2003.
- [11] C. Sartiani. On the Correctness of Query Results in XML P2P Databases. In *P2P2004*, 2004.
- [12] C. Sartiani, G. Ghelli, P. Manghi, and G. Conforti. XPeer: A self-organizing XML P2P database system. In *Proceedings of the First EDBT Workshop on P2P and Databases (P2P&DB 2004)*, 2004, 2004.