Towards P2P XML Database Technology

Ying Zhang
(supervised by Peter Boncz)
Centrum voor Wiskunde en Informatica
P.O.Box 94079, 1090 GB
Amsterdam, the Netherlands
Y.Zhang@cwi.nl

ABSTRACT

To ease the development of data-intensive P2P applications, we envision a P2P XML Database Management System (P2P XDBMS) that acts as a database middle-ware, providing a uniform database abstraction on top of a dynamic set of distributed data sources. In this PhD work, we research which features such a database abstraction should offer and how it can be realised efficiently by extending and combining existing XML databases with P2P technologies.

The first step in this research is a distributed database extension called XRPC. Our planned future work builds upon this, adding P2P abstractions to all main database functionalities (query processing, transactions and data storage).

1. INTRODUCTION

Developing P2P applications that need non-trivial distributed data management facilities is still a cumbersome task, because the applications have to deal with information from different data sources. As in P2P settings this set of data sources is highly dynamic, foreseeing all possible combinations of available data sources is impractical, putting a high adaptivity burden on the shoulders of the application programmer. To ease development of P2P applications, we envision a middle-ware P2P XML database system (P2P XDBMS) that manages dynamic collections of heterogeneous data sources and provides a uniform database abstraction to the application. We choose XML as our basis, because XML is a flexible data format for integrating different data sources, is ideal for distributed environments and XML query languages have gained standardisation momentum, allowing the use of XQuery as lingua franca for PDMS data sources. In this PhD work, we research which features such a database abstraction should offer by extending and combining existing XML database management systems with P2P technologies.

Practical Challenges. While the P2P database concept has generated a research niche, it has not yet been widely recognised as relevant. A first problem is that P2P database technology is understood by different researchers to mean different things, and there is no "role model" system (like what System-R was for the RDBMS) as an orientation point for the community. Secondly, most proposed techniques (e.g. P2P query processing algorithms) are evaluated in simulations whose results are hard to extrapolate to behaviour in real-world circumstances. A third and related problem is that so far no "killer applications" for P2P database technology have been recognised (in contrast to P2P systems – of which various mostly file-downloading systems have found a large user audience).

VLDB '07, September 23-28, 2007, Vienna, Austria.

Strategy. Our strategy for advancing the state-of-the-art is to incrementally extend functionality of stand-alone XML database systems to P2P in all database related dimensions (including query execution, query optimisation, transaction management and data storage) by developing a working P2P XDBMS prototype as a test-bed for our research and to work on applications that benefit from P2P database technologies. This strategy requires – besides research effort – a significant investment in prototype engineering. We are glad to be able to build on MonetDB/XQuery [9], an open source XDBMS based on purely relational query processing that supports XQuery [7] and the XQuery Update Facility (XQUF) [12]. The choice for XML as the data model – and web standards in general – eases many aspects of distributed data management (i.e. the XML data format is platform independent, and there is ubiquitous support for URIs and specifically HTTP networking, that we use for data and query transport).

We obtain P2P XDBMS functionality in two steps, by *orthogonally* extending XQuery first with explicit distributed querying and second, by making this distribution more implicit, abstract and self-managing (i.e. P2P). At this stage we have performed the first step by introducing XRPC [38], a minimal XQuery language extension that enables efficient querying of heterogeneous XQuery data sources. Our future work addresses the second step. As our vision is a data management technology that supports building P2P applications on top of *heterogeneous* data sources, a strong requirement is adherence to standards, in this case XQuery.

XRPC enhances the existing concept of XQuery functions with the Remote Procedure Call (RPC) paradigm. By calling RPC functions inside an XQuery `for`-loop to multiple destinations, and by calling functions that themselves perform XRPC calls, complex P2P communication patterns can be achieved. An important goal of XRPC is interoperability, that is, the extension should allow *different* XQuery engines to jointly execute queries. Adoption of P2P technology would be greatly helped by interoperability, and thus we advocate XRPC as a step towards a standardised distributed XQuery extension. Even while XRPC is currently only supported in the open-source XDBMS MonetDB/XQuery, we provide a *wrapper* that allows any XQuery peer to participate in distributed queries.

Adding high level P2P services (i.e. transparent querying of shared data in P2P settings) to XQuery is the next item on our future work agenda. A first step towards this direction has been formulated in our other work [37], where we described two architectures for coupling Distributed Hash Tables (DHTs) with an XDBMS.

Research Questions. Our challenge of create P2P XDBMS technology is quite wide and concerns all three classical database research areas: query execution, transaction management and data storage. During the Ph.D. project, we will gradually focus on a subset of research questions from these areas.
– *P2P Query Execution:* How can we automatically rewrite normal (data-shipping) XQuery queries into distributed queries using XRPC to perform function shipping strategies? While such query distribution has been studied before, the new focus here is to ensure that proper XQuery semantics (in particular node identity) is respected in this transformation. How to capture P2P query execution cost in models (that take peer-, data-, network- and query-characteristics into account) and use those to guide the optimisation process?
– *P2P Transaction Management:* Fully functional distributed transaction management does not scale to P2P environments. We see Distributed Snapshot Isolation as particularly interesting for P2P XDBMS as it needs weaker locking protocols. Thus, one question is what semantics of Distributed Snapshot Isolation would still be useful in P2P environments? If a useful isolation level is identified, a further challenge is to develop proper transaction protocols for it.
– *P2P Storage and Placement:* How we abstract from the physical locations of distributed XML documents in an elegant XQuery extension? What can be the role of P2P data structures such as DHTs in transparent, but also robust, distributed XML storage?

Project Embedding. This PhD work is part of the AmbientDB project [8, 15], which in turn is a sub-project of the Dutch national research project MultimediaN (*www.multimedian.nl*), that unites multimedia and database researchers in various academic and industrial research institutes.

StreetTiVo is one of the demo applications being developed by the MultimediaN project. The StreetTiVo application is a plug-in for so-called Home Theatre PCs (MythTV and Windows Media Center Edition), which one can consider programmable digital video recorders. The StreetTiVo plug-in enables real-time content-based video retrieval and meta-data generation, by distributing compute -intensive video analysis over multiple peers that have recorded the same TV program. This application involves distributed collaborator discovery, work coordination, and meta-data exchange in a volatile WAN environment (but not video file exchange – the application is strictly legal). The hypothesis that we test in StreetTiVo, is whether a P2P XDBMS eases development of such an application.

Outline. Section 2 summarises our work until now around XRPC, the XQuery extension for distributed querying [38, 37] and section 3 elaborates the follow-up research directions. We discuss related work in Section 4 and draw conclusions in Section 5.

2. CURRENT STATUS: XRPC

The first main question towards P2P XDBMS was to devise an elegant distributed query extension that aligns fully with the W3C XQuery Formal Semantics [13]. The resulting XRPC proposal, presented in the VLDB conference co-located with this Ph.D. workshop [38] also takes into account the recent XQuery Update Facility (XQUF) [12].

While we think XRPC is elegant and creates a perspective for P2P interoperability between any peer that provides XQuery or even only a SOAP-based web service, XRPC only provides explicit distributed query execution (no transparency).

We also summarise some initial work on providing a higher level of distributed abstraction, by using DHTs to provide *logical* network locations for data shipping and function shipping [37] (rather than physical network locations).

2.1 The XRPC Language Extension

The XRPC syntax for remote function application is:

`execute at` { *Expr* }{ *FunApp* (*ParamList*) }

where *Expr* is an XQuery `xs:string` expression that specifies the URI of the peer on which *FunApp* is to be executed. The function to be applied can be built-in or user-defined. For user-defined functions, we currently restrict ourselves to functions defined in an XQuery module. A small future extension to the network protocol would also allow functions defined inside a query to be executed over XRPC.

Use cases. As a running example, we assume a set of XQuery DBMS (peers) that each store a movie database document `filmDB.xml` with contents similar to:

```
<films>
 <film><name>The Rock</name><actor>Sean Connery</actor></film>
 <film><name>Green Card</name><actor>Gerard Depardieu</actor></film>
</films>
```

We assume an XQuery module `film.xq` stored at `x.example.org`, that defines a function `filmsByActor()`:

```
module namespace f="films";
declare function f:filmsByActor($actor as xs:string) as node()*
{ doc("filmDB.xml")//name[../actor=$actor] };
```

We can execute this function on the remote peer `y.example.org` to get a list of films in the remote movie database, in which Sean Connery plays:

```
import module namespace f="films" at "http://x.example.org/film.xq";
<films> {
execute at {"xrpc://y.example.org"} {f:filmsByActor("Sean Connery")}
} </films>
```

which yields: `<films><name>The Rock</name></films>`

We introduce here a new `xrpc` network protocol, accepted in the destination URI of `execute at`. The generic form of such URIs is: `xrpc://`< *host* > [: *port*] [/[*path*]]. The `xrpc://` indicates the network protocol. The second part, < *host* > [: *port*], indicates the remote peer. The third part, [/[*path*]], is an optional local path at the remote peer.

The SOAP XRPC Protocol. The design goal of XRPC is to create a distributed XQuery mechanism with which *different* XQuery processors at different sites can jointly execute queries. This implies that our proposal also encompasses a *network protocol*, which uses the Simple Object Access Protocol (SOAP) [26] (i.e. XML messages) over HTTP. The choice for SOAP brings as additional advantages seamless integration of XQuery data sources with web services and Service Oriented Architectures (SOA) as well as AJAX-style GUIs. The complete specification of the SOAP XRPC protocol can be found in [38]. Here we show, as an example, the XRPC `request` message that should be generated for the query above:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:xrpc="http://monetdb.cwi.nl/XQuery"
              xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://monetdb.cwi.nl/XQuery
                                  http://monetdb.cwi.nl/XQuery/XRPC.xsd">
 <env:Body>
  <xrpc:request module="films" method="filmsByActor" arity="1"
                location="http://x.example.org/film.xq">
   <xrpc:call>
    <xrpc:sequence>
     <xrpc:atomic-value
      xsi:type="xs:string">Sean Connery</xrpc:atomic-value>
    </xrpc:sequence>
   </xrpc:call>
  </xrpc:request>
 </env:Body>
</env:Envelope>
```

2.2 XRPC Semantics

We have added two new semantic judgement rules ($\mathcal{R}_{\mathcal{F}_r}$ and $\mathcal{R}_{\mathcal{F}_u}$) to the XQuery Formal Semantics [13] to define the formal semantics of basic read-only XRPC calls and XRPC calls to updating functions, respectively. These rules specify how an XRPC call is handled according to its read-only or updating property, and how the execution of an XRPC call affects the database states of the local peer (from which the call originates) and the remote peer (on which the function is actually applied). The database state db is defined as the documents stored in an XML database. In short, a read-only XRPC call does not change the database states of either peer; an updating XRPC call *may* change the database of the remote peer at the end of the function application. A thorough explanation of these rules can be found in [38].

XRPC calls can be nested arbitrarily, i.e. a query can contain multiple XRPC function applications and the called functions can again perform XRPC calls. During the evaluation of a single XRPC query, it may happen that multiple XRPC requests are sent to the same peer p, and the database state of p (denoted as db^p) may thus be seen multiple times during query execution. In between those multiple function evaluations, other transactions may update the database and change db^p. Thus, those different XRPC calls to the same remote peer p from the same query may see different database states. This will not be acceptable for some applications, and therefore, we deem it worthwhile to define rules to provide a higher isolation level.

In [38], we defined two additional rules $\mathcal{R}'_{\mathcal{F}_r}$, which offers the *repeatable read* isolation for read-only XRPC queries, and $\mathcal{R}'_{\mathcal{F}_u}$, which offers both repeatable reads and atomic distributed commit for updating XRPC queries. In summary, both rules state that to evaluate XRPC calls on behalf of query q, peer p always uses the same database state db^p_q. To realise this, all requests sent for the same query are labelled with the unique identifier q of the query, such that upon receiving requests, a peer can recognise which requests belong to the same query and thus associates an isolated database state with them. In addition, the rule $\mathcal{R}'_{\mathcal{F}_u}$ specifies that updates produced by an individual XRPC call of query q are *stored* and only applied at the time query q commits. This semantics corresponds more closely to the intent of XQUF, in that no side effects of a query are visible at any involved peer before the query commits. Note that, atomically committing a distributed transaction requires a protocol like the Two-Phase Commit protocol (2PC) or one of its more advanced derivatives [18, 28]. We decided not to add 2PC to the SOAP XRPC network protocol, but rather rely on the recent industry standard WS AtomicTransaction [3, 2] that provides exactly this feature for distributed web-service transactions.

2.3 Bulk RPC

Our SOAP XRPC protocol allows computing multiple applications of the same function (with different parameters) in a *single* request/response network interaction. We call this *Bulk RPC* and it has several advantages. Firstly, Bulk RPC is much more efficient than repeated single RPCs as network latency is amortised over many calls, and performance becomes bounded by network bandwidth or CPU throughput (hardware factors that scale much better than network latency). Secondly, Bulk RPC exposes bulk execution opportunities, such that e.g. a function that selects with a constant argument is turned into a join against the sequence of all arguments. Bulk RPC thus has a direct correspondence with set-oriented processing as offered by query algebras, and we believe it can be generally applied to any algebraic XQuery implementation. Thirdly, for *simple* XRPC queries, i.e. those that contain only applications of the same remote function to one remote peer, bulk RPC helps to avoid using costly isolation mechanisms, since multiple RPCs are turned into a single request.

	total	compile	treebuild	exec
echoVoid `$x=1`	275	178	4.6	92
echoVoid `$x=1000`	590	178	86	325
getPerson `$x=1`	4276	185	1956	2134
getPerson `$x=1000`	8167	185	1973	6010

Table 1: Saxon Latency via the XRPC Wrapper (msec)

We conducted two experiments to study the effect of bulk RPC. [1] We define an `echoVoid` function and call it over XRPC. We compare the performance of bulk XRPC with one-at-a-time RPC by varying the number of loop iterations `$x`.

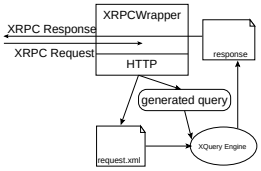
```
module namespace test = "test";
declare function echoVoid() { () };

import module namespace t="test" at "http://x.example.org/test.xq";
for $i in (1 to $x)
return execute at {"xrpc://y.example.org"} {t:echoVoid()}
```

	`$x=1`	`$x=1000`
one-at-a-time	2.6	2696
bulk	2.7	4

The table on the right shows the results of the experiments (in msec). It shows that the performance is fairly similar at `$x=1`, such that we can conclude that the overhead of Bulk RPC is small. At `$x=1000`, there is an enormous difference, caused by *(i)* serialisation/deserialisation of the request/response messages, *(ii)* network communication and *(iii)* overhead of function calls (1000 instead of 1). This is easily explained as the one-at-a-time RPC experiments involves performing 1000 times synchronous RPCs. Thanks to bulk execution, XRPC can achieve a minimum latency of 4 msec – which is close to that of commercial-strength software like .NET ([17]).

2.4 XRPC Wrapper



Cross-system distributed XRPC querying can be achieved even without XRPC being integrated into an XQuery processing engine. What is needed is a simple *XRPC wrapper* on top of the XQuery system, as shown here. The XRPC wrapper is a SOAP service handler that stores the incoming SOAP XRPC request message in a temporary location, generates an XQuery query for this request, and executes it on an XQuery processor. The generated query is crafted to compute the result of a Bulk XRPC by repeatedly calling the requested function on the parameters found in the message, and to generate the SOAP response message in XML using element construction. Such an XRPC wrapper only allows *handling* calls with normally XRPC-incapable systems, but obviously does not allow making outgoing XRPC calls from them.

We conducted two experiments by running a simple XRPC wrapper on the Saxon XSLT/XQuery processor (we used Saxon-B 8.7). The results are shown in Table 1. Again we vary the number of iterations (`$x`) to study the performance impact of Bulk RPC. By the absence of a function cache, Saxon latency is dominated by start-up and compilation time, so we focus here on the internal Saxon timings (compile, treebuild, exec) and disregard network communication cost, which is a few msec at most.

From the `echoVoid` experiment, we see that Bulk RPC again amortises XRPC latency well: with a 1000 times more work, the total latency increases just over a factor 2. As the execution time still is increased by a factor 30, the low impact is due to other amortised latencies, in parsing the XML request document, compiling the query, etc.

[1] All experiments presented in this paper have been done on 2GHz Athlon64 Linux machines connected on 1Gb/s Ethernet.

We also show the results of a `getPerson()` example, which returns the `person` node from an XMark document whose `@id` attribute matches the given `$pid`:

```
declare function getPerson($pid as xs:string) as node()?
{ zero-or-one(doc("xmark.xml")//person[@id=$pid]) };
```

This exposes an additional benefit of Bulk RPC over just amortised fixed latencies: whereas in the single-call case, `getPerson()` behaves like a selection over the XMark document, the Bulk version of `getPerson()`, which iterates over all calls in the request, becomes an *equi-join*. Again, the total time for a Bulk RPC with 1000 calls is only about twice as much as a single call, but here we see that the execution time impact has increased only by a factor of 3 (was 30 in `echoVoid`). The explanation is that Saxon is able to detect the join condition and builds a hash-table such that performance remains linear in the size of the XMark document, just like it was in the single call selection.

2.5 Distributed XQuery with XRPC

One of the design goals of XRPC is to have it serve as the target language for a distributed XQuery optimiser that takes queries without XRPC calls as input (only data shipping) and produces a decomposed query as output that uses XRPC for function shipping.

We conducted some experiments with the following example. Assume a distributed XDBMS with two peers $\{p_a, p_b\}$. An XMark document is distributed between these two peers, where p_a stores all persons in "`persons.xml`", and p_b stores all items and auctions in "`auctions.xml`".

```
for $p  in doc("persons.xml")//person,
    $ca in doc("xrpc://B/auctions.xml")//closed_auction        (Q1)
where $p/@id = $ca/buyer/@person
return <result>{$p,$ca/annotation}</result>
```

The query $\mathbb{Q}_1$ above is executed at peer p_a. For each person and for every item this person has bought, the query returns the `person` node and the `annotation` node of the bought item in a new `result` node. For the moment, assume that `fn:doc()` is invoked with a compile-time known constant URI from the special URI name scheme "`xrpc://`".

A minimal optimisation might be to push predicates that depend only on a single `fn:doc("xrpc://p/..")` into its data source p, which we call ***Predicate Push-down***. Thus, instead of copying the whole `auctions.xml` from p_b to p_a, we define a function to return only `closed_auction` nodes and execute this function on p_b

But the possibilities of query rewriting need not stop at push-down of `fn:doc("xrpc://..")`-dependent expressions. Even if a query depends on a set $\mathcal{P}$ of XRPC peers that contribute documents, one could decide to select one peer p_i from $\mathcal{P}$ and put *all* execution to p_i. We call this mechanism ***Execution Relocation***. For example, it might be beneficial to relocate all execution on p_b, wrapped in a function, if `"auctions.xml"` is much larger than `"persons.xml"`. Then peer p_a only needs to call that function to get the results.

The classical ***distributed semi-join*** strategy [5, 36] can be employed as well. The XRPC equivalent of the semi-join strategy uses an XRPC function call with a loop-dependent parameter. In this case, the person `@id` for all persons could be passed in a loop to a function executed at p_b that returns those closed auctions with buyers having that `@id`:

```
module namespace b = "functions_b";
declare function b:Q_B3($pid as xs:string) as node()*
{ doc("auctions.xml")//closed_auction[./buyer/@person=$pid] };

import module namespace b="functions_b" at "http://example.org/b.xq";
for $p  in doc("persons.xml")//person
let $ca := execute at {"B"} {b:Q_B3($p/@id)}                   (Q2)
return if(empty($ca)) then ()
       else <result>{$p, $ca/annotation}</result>
```

	Total Time	MonetDB Time	Saxon Time
data shipping	28122	16457	11665
predicate push-down	25799	2961	22838
execution relocation	53184	69	53115
distributed semi-join	10278	118	10160

Table 2: Execution time (msec) of query $\mathbb{Q}_2$ distributed on MonetDB/XQuery and Saxon (Saxon Time includes network).

This shows that federating data sources with XRPC (even via the XRPC Wrapper) is more powerful than the "wrapper-architecture" [23] used in federated database systems. Such wrappers typically lack the possibility to push table-valued parameters to data sources, which is required for the semi-join optimisation.

To demonstrate the interoperability, expressiveness and performance potential of XRPC we run query $\mathbb{Q}_2$ on two peers using all four strategies. On peer p_a (the local peer), we run MonetDB/XQuery with the document "persons.xml" (1.1MB, 250 `person` nodes); on peer p_b we run Saxon with the document "auctions.xml" (50MB, 4875 `closed_auction` nodes). There are 6 matches between the `person` nodes and the `closed_auction` nodes. All communication between MonetDB/XQuery and Saxon happens via XRPC. The XRPC wrapper described in section 2.4 is used to generate the XQuery query from an XRPC request message.

The measured execution times are shown in Table 2. Column "MonetDB Time" contains execution times on peer p_a and column "Saxon Time" contains execution times on peer p_b. The Saxon Time was measured by subtracting MonetDB Time from Total Time, such that it also included communication. We should stress that this experiment is not a rigorous evaluation of distributed query execution strategies, rather a demonstration of the possibilities of XRPC. The results here show that the "data shipping" query is relatively expensive, since it spends quite some Saxon time on shipping the 50MB document and then still needs to do the join. The "predicate push-down" approach improves the performance, as we would expect. The "execution relocation" largely relieves the MonetDB peer from execution responsibilities, but still ships a significant amount of data and tasks Saxon with the whole join and result construction effort. The "distributed semi-join" is the strategy that incurs the least data shipping, and is most efficient in this case.

2.6 DHT Coupling

A DHT-based network [35] provides *(i)* robust connectivity (i.e. tries to prevent network partitioning), *(ii)* high data availability (i.e. prevent data loss if a peer goes down by automatic replication), and *(iii)* a scalable (key,value) storage mechanism with $O(log(N))$ cost complexity (where N is the number of peers in the network). A number of P2P database prototypes have already used DHTs [10, 21, 22, 29]. An important design question is how a DHT could be exploited by an XQuery processor, and if and how the DHT functionality should surface in the query language.

We propose here to avoid any additional language extensions, but rather introduce a new `dht` network protocol, accepted in the URI of `fn:doc()`, `fn:put()` and `execute at`. The generic form of such URIs is `dht://`*dht_id*`/`*key*. The `dht://` indicates the network protocol. The second part, *dht_id*, indicates the DHT network to be used. Such an ID is useful to allow a PDMS to participate in multiple (logical) DHTs simultaneously. The third part, *key*, is used to store and retrieve values in the DHT.

In the following, we show how support for PDMS applications can be provided by the `fn:doc()` and `fn:put()` built-in functions plus our XRPC `execute at` language construction.

Loose DHT Coupling. The simplest architecture to couple a DHT network with an XDBMS is to just use the DHT API to imple-

ment the XQuery data shipping functions. That is, the XQuery function `fn:put($node, $uri)` is mapped to the DHT function `put(key, value)` to store XML documents as string values on DHT peers, and the XQuery function `fn:doc($uri)` is mapped to the DHT function `get(key):value` to retrieve an XML documents as a string value. The `$uri` is used as the `key` to identify the string value in the DHT.

The XRPC queries can be "simulated" by getting all documents with a relative URI name from the DHT and then evaluating the function locally. How this works exactly, is elaborated in [37]. Note that, while this approach allows zero-effort coupling of DHT technology with DBMS technology, we consider it nothing more than a workaround. Function-shipping is replaced by data-shipping, which defeats the purpose of XRPC.

Tight DHT Coupling. In a tight coupling scenario, rather than keeping XML as string blobs inside the DHT (in RAM), each DHT peer uses its local XDBMS to store the documents. To realise this, we need to extend the DHT API with a single new method:

$$\texttt{xrpc}\ (key,\ q,\ m,\ f_r(ParamList))\texttt{:item()}^*,$$

which allows an XRPC request to be routed through the DHT to the peer p_i responsible for *key*. When the DHT instance on p_i receives such a request, it passes the request to the MonetDB/XQuery instance on the same peer to be handled. The response is then transported back via the DHT towards the query originator. In this scenario, the data-shipping functions `fn:put()` and `fn:doc()` can be mapped to an XRPC call, which causes the functions to be actually executed on the remote peer that is responsible for the document's `$uri` as key.

In the tight coupling, we have to extend the DHT data structure. A positive side effect of this is that the DBMS gets access to information internal to the P2P network. This information (peer resources, connectivity, etc) can be exploited in query optimisation. Also, bulk XRPC requests routed over the DHT may be optimised (similar to Bulk RPC), by combining requests that follow the same route as long as possible in single network messages. In order to reap the robustness benefit of P2P data structures, an additional extension will need to be made in the automatic replication performed by the DHT, which replicates (key,value) pairs on multiple peers to avoid data loss under churn. As the tight coupling stores the values in the peer XDBMS, such replication actions need to perform distributed database updates. The needed protocols and efficiency/robustness trade-offs are subjects of our future work.

3. NEXT STEPS

Distributed XQuery Optimiser. We are working on a distributed XQuery optimiser that converts non-XRPC data shipping queries into function shipping queries using XRPC, respecting the original query semantics. The main new challenge is imposed here by the *by-value* parameter and result passing of XRPC, which causes nodes to lose their original identity at the remote side, and cuts them off from their original ancestors and siblings. Also, possible relationships (e.g. `descendant-or-self`) between node parameters (or results) will be destroyed at the remote side. Thus, a challenge is to identify those predicates in a data shipping XQuery that can be shipped over XRPC *without* affecting the query semantics. In principle, we can push predicates into other peers as long as these predicates depend only on a single `fn:doc()` function application. Therefore, we will need to extend the work in [25] to determine the dependence of XQuery sub-expressions for the purpose of XRPC distribution. However, for semijoin-like rewrites, part of the node processing needs to be done on the remote side. Simple non-join XPath navigation can be done remotely, as long as only the downwards axes are used. To expand our possibilities here, we can enhance XRPC with so-called *by-fragment* instead of by-value result/parameter passing. This alternative SOAP format eliminates any duplicate XML sent in the XRPC SOAP messages, and as a result conserves ancestor/descendant relationships. This allows e.g. to execute (semi) join remotely for join predicates that only use the downwards axes.

Scalable P2P Transaction Management. We deem it important to define less strict but more scalable protocols for managing transactions in P2P settings. We are especially interested in Distributed Snapshot Isolation (DSI), because it requires weaker locking protocols, which makes it likely to perform better than classical two-phase locking protocol in high-latency WAN environments. To our knowledge, there has not been much previous work on Distributed Snapshot Isolation, while lately in commercial applications (centralised) snapshot isolation has found wide user acceptance. One idea here is to use Lamport Clocks [24] as the timestamp for DSI, providing the notion of *Lamport consistency*. The objectives of this work will contain a formal definition of this consistency criterion, as well as an analysis of the protocols needed. We also plan to validate this approach in the StreetTiVo application and in experiments on PlanetLab (*www.planet-lab.org*).

Integrating XQuery and P2P. We will continue our initial work described in section 2.6 of integrating DHTs with XDBMS technology, and in particular XRPC. A first goal of using DHTs is to create "logical URL"s that specify XML data items without necessarily pinning down the actual URL (hostname, path). Such logical URLs may even be used as synonyms for XML data items that are spread over multiple locations. Another benefit of using DHTs is to allow $O(log(N))$ network cost equi-selections (N is the number of peers) and add self-managing properties to the distributed system under churn, i.e. maintaining connectivity, and providing an automatic replication mechanism that prevents data loss. Apart from these design aspects of the XDBMS-DHT integration, we are also interested in which query optimisation possibilities that a tight DHT coupling can provide. On the practical side, we will be testing algorithms and conducting experiments by coupling MonetDB/XQuery with the Bamboo [32] DHT; again carrying out experiments on PlanetLab.

4. RELATED WORK

P2P networks are an active topic in networking research, especially Distributed Hash Tables [30, 33, 35, 39]. For practical purposes, the systems Bamboo [32] and P-Grid [4] currently seem to be the most usable.

[19] might be the first paper that discusses database management issues in P2P environments and it proposes the Piazza system [20] to manage data placement in P2P settings. PIER [21] is a P2P information exchange and retrieval system on top of the Bamboo DHT. It uses a relational data model and query language and has some support for in-network joins. UniStore [22] provides an RDF-like triple storage on top of P-Grid. XPeer [34] is a P2P XDBMS for sharing and querying XML data, on top of a super-peer network. The query algebra of XPeer takes explicitly into account data dissemination, data replication and data freshness. [29] is a proposal for an XML-based database system on top of DHT (FreePastry), which is also named XPeer. [29] uses XPath as its query language and supports range queries. Of these systems, the PIER and UniStore systems seem to be the most developed prototypes so far.

In the area of extending XQuery with distributed querying capabilities, our work is closely related to DXQ [14], developed as an

extension of Galax. DXQ ships distributed query plans, in terms of the internal Galax execution algebra, generated by the Galax optimiser, to remote peers to be executed. The syntax of XRPC is inspired by XQueryD [31], which supports shipping of free-form queries by using a runtime rewriter to scan the XQuery expressions in the `execute` statement for variables and substitute them with the runtime values. In Active XML [6], calls to service functions are embedded in XML documents. Service functions are defined using the XML query language X-OQL ([1]), which itself does not allow distributed evaluation. The SOAP protocol used for AXML services has not been specified formally; like XRPC it uses a document/literal encoding to represent XML sub-tree values. Galax Yoo-Hoo! [27] is related to our work in the sense that web services are accessed using remote procedure calls and SOAP messages are used as the communication protocol, but messages must be manipulated explicitly with element construction.

The AquaLogic Data Service Platform (ALDSP) [11] is a middle ware product recently introduced by BEA. ALDSP provides an (XML-based) declarative way for building SOA applications and services that need to access and compose information from a range of (heterogeneous) enterprise data sources.

In the area of distributed query processing and transactions, much prior research has been done. There have been several surveys on these topics, such as [23], [36] and parts of [28]. Distributed XRPC updates with isolation need a distributed commit protocol, for which any of the 2PC protocol [28], the Paxos Commit algorithm [18] and the distributed Sagas [16] could be used.

5. CONCLUSION

In this paper, we *(i)* motivated the need for P2P XDBMS, *(ii)* presented the XML-based XRPC approach we took as the basis and the current progress of the project, and *(iii)* discussed the challenges we will address in the future. From the overview of the current progress and the presented performance results in this paper, we would conclude that XRPC is a sufficiently powerful foundation for further research towards the envisioned P2P XDBMS technologies.

6. REFERENCES

[1] The Active XML Project. http://activexml.net.

[2] Web Services Atomic Transaction (WS-AtomicTransaction), August 2005. ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf.

[3] Web Services Coordination (WS-Coordination), August 2005. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[4] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *CooplS*, 2001.

[5] P. Apers, A. R. Hevner, and S. B. Yao. Optimization algorithms for distributed queries. *IEEE TSE*, 9(1), 1983.

[6] O. Benjelloun. *Active XML: A data-centric perspective on Web services*. PhD thesis, September 2004.

[7] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. W3C Candidate Recommendation 8 June 2006. http://www.w3.org/TR/2006/CR-xquery-20060608.

[8] P. Boncz. AmbientDB: P2P Database Technology for Ambient Intelligent Multimedia Applications. *ERCIM News*, (55), October 2003.

[9] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, June 2006.

[10] A. Bonifati, E. Q. Chang, T. Ho, and L. V. Lakshmanan. HepToX: Heterogeneous Peer to Peer XML Databases. Technical Report UBC TR-2005-15, 2005.

[11] M. Carey and the BEA ALDSP Team. Data Delivery in a Service-Oriented World: The BEA AquaLogic Data Services Platform. In *SIGMOD*, 2006.

[12] D. Chamberlin, D. Florescu, and J. Robie. XQuery Update Facility. W3C Working Draft 11 July 2006. http://www.w3.org/TR/2006/WD-xqupdate-20060711.

[13] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Candidate Recommendation 8 June 2006. http://www.w3.org/TR/2006/CR-xquery-semantics-20060608.

[14] M. Fernández, T. Jim, K. Morton, N. Onose, and J. Siméon. Highly distributed xquery with dxq. In *SIGMOD demo*, June 2007.

[15] W. Fontijn and P. Boncz. AmbientDB: P2P Data Management Middleware for Ambient Intelligence. In *PERWARE*, 2004.

[16] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, 1987.

[17] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. van Engelen, and M. J. Lewis. Toward Characterizing the Performance of SOAP Toolkits. In *GRID*, 2004.

[18] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1), 2006.

[19] S. D. Gribble, A. Y. Halevy, Z. G. Ives, M. Rodrig, and D. Suciu. What Can Peer-to-Peer Do For Databases, and Vice Versa? In *WebDB*, 2001.

[20] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *WWW*, 2003.

[21] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. In *CIDR*, 2005.

[22] M. Karnstedt, K.-U. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John. UniStore: Querying a DHT-based Universal Storage. Technical report, 2006.

[23] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4), 2000.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

[25] A. Marian and J. Siméon. Projecting XML Documents. In *VLDB*, September 2003.

[26] N. Mitra. SOAP Version 1.2 Part 0: Primer. W3C Recommendation 24 June 2003. http://www.w3.org/TR/2003/REC-soap12-part0-20030624.

[27] N. Onose and J. Siméon. XQuery at Your Web Service. In *WWW*, 2004.

[28] M. T. Özsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., NJ, USA, 1999.

[29] W. Rao, H. Song, and F. Ma. Querying XML Data over DHT System Using XPeer. In *GCC*, 2004.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[31] C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *IIWeb*, September 2004.

[32] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference, General Track*, 2004.

[33] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware*, 2001.

[34] C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. XPeer: A Self-Organizing XML P2P Database System. In *P2P&DB*, 2004.

[35] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[36] C. Yu and C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4), 1984.

[37] Y. Zhang and P. Boncz. Integrating XQuery and P2P in MonetDB/XQuery*. In *EROW*, 2007.

[38] Y. Zhang and P. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *VLDB*, 2007. To appear.

[39] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE J-SAC*, 22(1), January 2004.