Integrating XQuery and P2P in MonetDB/XQuery*

Ying Zhang and Peter Boncz

Centrum voor Wiskunde en Informatica, P.O.Box 94079, 1090 GB, Amsterdam, the Netherlands
{Y.Zhang, P.Boncz}@cwi.nl

Abstract. MonetDB/XQuery* is a fully functional publicly available XML DBMS that has been extended with distributed and P2P data management functionality. Our (minimal) XQuery language extension XRPC adds the concept of RPC to XQuery, and exploits the set-at-a-time database processing model to optimize the networking cost through a technique called Bulk RPC. We describe our approach to include the services offered by diverse P2P network structures (such as DHTs), in a way that avoids any further intrusion in the XQuery language and semantics, and show how this, similarly to Bulk RPC, will lead to further query optimization opportunities where the XDBMS interacts with the underlying P2P network. We also discuss some P2P data management applications were MonetDB/XQuery* is being used (an in-home small scenario and a wide-area collaborative application). As this research is work-in-progress, we outline some research questions on our path towards defining and realizing P2P XDBMS technology.

1 Introduction

In the AmbientDB [9] project, we are building MonetDB/XQuery* , an open-source XML DBMS (XDBMS) with support for distributed querying and P2P services. Our work is motivated by the hypothesis that P2P is a disruptive paradigm that should change the nature of database technology. Most of the existing distributed DBMS technologies were developed to be used in (small-scale) local-area networks (LAN). Those technologies usually assume that *(i)* there is a central controller and/or peers have complete knowledge of the whole system, *(ii)* peers are uniform and highly available, *(iii)* placement of data happens in a controlled way and is rarely changed and *(iv)* a global database schema is used. Peer-to-Peer (P2P) networks have led the distributed DBMS research to reconsider existing technologies in such a new environment, where *(i)* systems have decentralized architectures, *(ii)* peers join or leave the network at any time, *(iii)* placement of data is out of the system's control and it changes frequently, *(iv)* each peer can have its local database schema (or no schema at all), and *(v)* data owned by the peers are often incomplete, overlapping and even conflicting.

Challenges. While the P2P database concept has generated a research niche, the concept has not yet been widely recognized 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 recognized (in contrast to P2P systems – of which various mostly file-downloading systems have found a large user audience).

Strategy. Our strategy for advancing the state-of-the-art is to incrementally develop a working P2P database 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 large investment in prototype engineering. We are glad to be able to build on MonetDB/XQuery [8], an open source XDBMS based on purely relational query processing that supports XQuery [3] and the XQuery Update Facility (XUF) [11]. The choice for XML as a 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 by *orthogonally* extending XQuery with support for *(i)* distributed querying and *(ii)* P2P services. At this stage we have made the step *(i)* by introducing XRPC, an XQuery language extension (a full discussion is in [29]). We also formulate the requirements and current direction for achieving step *(ii)*, and illustrate the working of our envisioned system in a P2P application called StreetTiVo.

StreetTiVo is a showcase application being developed by the Dutch national research project MultimediaN, that unites multimedia and database researchers in various academic and industrial research institutes. 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 recorded the same TV program. This application involves distributed collaborator discovery, work coordination, and result exchange in a volatile WAN environment (but not video file exchange – it is strictly legal). We think that deploying ready-to-run P2P data management technology enables quick development of this application.

Outline. In Section 2, we present our first step, namely distributed XQuery processing using a language extension called XRPC. While XRPC already allows to perform P2P queries, it still misses a number of vital P2P functionalities (robust connectivity, peer and resource discovery, approximate query/transaction processing). In Section **??**, we outline our plans and research questions to address these open issues. We also illustrate how the described P2P XDBMS can be used in the StreetTiVo application. Finally, in Section 4 we describe related work, before concluding in Section 5.

2 XRPC: Distributed XQuery Processing

The XRPC syntax for remote function application is similar to XQueryD [22]:

"`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. Here we restrict the function application $FunApp$

to user-defined functions (UDF) that are defined in a module. Thus, the defining parameters of an XRPC call are: *(i)* a module URI, *(ii)* a function name, and *(iii)* the actual parameters. The module URI is the one bound to the namespace identifier in the function application. Just like a "`import module`" statement, the module URI may be supplemented by a so-called "`at`" hint, which also is a URI.

We chose to exclude calling *built-in* functions over XRPC, since remote execution of local parameters does not provide any functional benefit over local execution. We also exclude remote application of user-defined functions specified inside the query (rather than in a module). This latter restriction simplifies the issue of how to transport the query definition from caller to callee, as it allows the XQuery system implementing XRPC to re-use the existing mechanism for function resolution from imported modules.

We made a conscious choice for *by value* [1] parameter passing, as *by reference* semantics would make it very complicated to orthogonally support XPath/XQuery on parameters or results of RPC calls (think of calling `parent::*` on an XML node type parameter, passed by reference – it would require additional implicit communication).

Examples. As a running example, we assume a set of XQuery database systems (peers) that each store a film database document "`films.xml`" with contents similar to:

```
<films>
  <film><filmName>The Rock</filmName><actorName>Sean Connery</actorName></film>
  <film><filmName>Goldfinger</filmName><actorName>Sean Connery</actorName></film>
  <film><filmName>Green Card</filmName><actorName>Gerard Depardieu</actorName></film>
</films>
```

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

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

We can execute this function on the remote peer "`y.org`" to get a sequence of films in which Sean Connery plays in the remote film database:

```
import module namespace film="filmdb" at "http://x.org/film.xq";
<films>                                                                    (Q1)
{ execute at {"xrpc://y.org"} {film:filmsByActor("Sean Connery")} }
</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 above example yields:

```
<films>
  <filmName>The Rock</filmName>
  <filmName>Goldfinger</filmName>
</films>
```

Another example performs multiple remote function calls to a single peer:

```
import module namespace film="filmdb" at "http://x.org/film.xq";
<films>
{ for $actor in ("Julie Andrews", "Sean Connery")                          (Q2)
  let $dst := "xrpc://y.org"
  return execute at {$dst} {film:filmsByActor($actor)} }
</films>
```

[1] Only the subtree rooted at a node parameter is sent.

Note that one can also perform XRPC calls to many different peers from within the same query (since the destination expression is unrestricted).

2.1 SOAP XRPC Message Format

SOAP (Simple Object Access Protocol) is an XML-based message format used for web services [2]. We propose the use of SOAP messages over HTTP as the network protocol underlying XRPC. The choice for SOAP allows seamless integration of XRPC with web services and Service Oriented Architectures (SOA). SOAP web service interactions usually follow a RPC (request/response) pattern, though the SOAP protocol is much richer and allows multi-hop communications, and highly configurable error handling.

XRPC Request Message. SOAP messages consist of an `Envelope`, with a (optional) `Header` and a `Body`. Inside the body, we define a `request` that specifies a module URI `module`, an (optional) at-hint `location` and a function name `method`. The actual parameters of a single function call are enclosed by a `call` element. Each individual parameter consists of a `sequence` element, that contains zero or more values. Below we show the SOAP XRPC request message for the example query `(Q1)`:

```
<?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="filmdb" location="http://x.org/film.xq" method="filmsByActor">
      <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>
```

XRPC Response Messages follow the same principles. Inside the body is now a `response` element that contains the result `sequence` of the remote function call. In the messages, atomic values are represented with `atomic-value`, and are annotated with their (simple) XML Schema Type by the `xsi:type` attribute. Thus, the heterogeneously typed sequence containing an integer `2` and double `3.1` would become:

```
<xrpc:sequence>
  <xrpc:atomic-value xsi:type="xs:integer">2</xrpc:atomic-value>
  <xrpc:atomic-value xsi:type="xs:double">3.1</xrpc:atomic-value>
</xrpc:sequence>
```

XML nodes are passed *by value*, enclosed by an `element` tag:

```
<xrpc:sequence>
 <xrpc:element><filmName>The Rock</filmName></xrpc:element>
 <xrpc:element><filmName>Goldfinger</filmName></xrpc:element>
</xrpc:sequence>
```

Similarly, the XML Schema `XRPC.xsd`[2] defines enclosing elements for document, attribute, text, processing instruction, and comment nodes. XRPC fully supports the XQuery Data Model (XDM) [12], a requirement for making it an orthogonal XQuery language feature. User defined element types are annotated in the messages using `xsi:type` attributes, such that XRPC messages can be correctly validated [29].

[2] See `http://monetdb.cwi.nl/XQuery/XRPC.xsd`

2.2 XRPC Formal Semantics

In defining the semantics of XRPC [3], we take care to attach proper database semantics to the concept of RPC, to ensure that all RPCs being done on behalf of a single query see a consistent distributed database image and commit atomically. It is known that full serializability in distributed queries can come at a high cost, and therefore we first define a less strict isolation level that still may be useful for certain applications. We use the following notations and terms:

- $\mathcal{P}$ denotes a set of *peer identifiers*.
- $\mathcal{F}$ denotes a set of *XRPC function applications*. The focus of this paper is read-only function calls, which are indicated by f_r. If the evaluation of an XRPC call f requires evaluation of other XRPC call(s), we term f a *nested XRPC call*.
- $\mathcal{M}$ denotes a set of XQuery *modules*. A module consists of a number of function definitions d_f. Each XRPC call f must correspond to a definition d_f from some module $m_f \in \mathcal{M}$.
- Each query operates in a *dynamic context*. The XQuery 1.0 Formal Semantics [4] specifies the result of an expression to be defined by a semantic judgement $dynEnv \vdash Expr \Rightarrow val$. This judgement states that in the dynamic context $dynEnv$, the expression $Expr$ evaluates to the value val, where val is an instance of the XQuery Data Model [12]. For now, we simplify the dynamic environment to a database state db (i.e. the documents and their contents stored in the XML database): $dynEnv \simeq db@p$. The $dynEnv.docValue$ from the XQuery Formal Semantics [4] corresponds to db used here. To indicate a context at a particular peer p, we write $db@p$.

Basic Read-Only XRPC is defined by extending the XQuery 1.0 semantic judgements with a new rule $\mathcal{R}_{\mathcal{F}_r}$:

$$\frac{\begin{array}{c} send_{p_0 \to p_i} request(m, f_r, ParamList) \\ db@p_i \vdash f_r(ParamList) \Rightarrow val,\ db@p_i \\ send_{p_i \to p_0} response(val) \end{array}}{db@p_0 \vdash f_r(ParamList)@p_i \Rightarrow val,\ db@p_0} \quad (\mathcal{R}_{\mathcal{F}_r})$$

This rule states that in the dynamic context, evaluation of the read-only XRPC call $f_r(ParamList)@p_i$ starts with sending the request $(m, f, ParamList)$ to peer p_i. Here, m is the module URI (plus at-hint) in which function f_r is defined, and *ParamList* is a list of actual parameters. The function f_r is then evaluated as a normal local function in the dynamic context $dynEnv@p_i$, that consists of the database state $db@p_i$ of the remote peer p_i at the time the request arrived at p_i. The evaluation yields the value val, which is sent back to the local peer p_0. Hence, the final result of this XRPC call at p_0 is val.

Note that neither the local database state $db@p_0$ nor the remote database state $db@p_i$ were modified by the evaluation of a read-only XRPC function. The function evaluation result val is a transient value, which only exist in the runtime environment at both p_0 and p_i. Also note that this definition inductively relies on the XQuery Formal Semantics to evaluate f locally at p_i, and thus may trigger the evaluation of additional XRPC calls if these happen to be present in body of f.

[3] We adopt the notations from XQuery 1.0 and XPath 2.0 Formal Semantics [4]

Repeatable Reads. As a query may perform multiple XRPC calls and each XRPC call may again perform XRPC calls, it can happen that some peer p_i is contacted multiple times, even using different call sequences. When considering rule $\mathcal{R}_{\mathcal{F}_r}$, the dynamic environment $dynEnv@p_i$ containing the database state $db@p_i$ may thus be seen multiple times during query evaluation. In between those multiple function evaluations, other transactions may update the database and change $db@p_i$. Thus, those different XRPC calls to the same remote peer p_i from the same query q may see different database states. This will not be acceptable for some applications and therefore, we deem it worthwhile to define *repeatable read* isolation for queries that perform XRPC calls. For this purpose, we formulate a similar-looking rule $\mathcal{R}'_{\mathcal{F}_r}$, where we tag the database state with a query identifier: $db_q@p$.

$$\frac{\begin{array}{c} send_{p_0 \to p_i} request(q, m, f_r, ParamList) \\ db_q@p_i \vdash f_r(ParamList) \Rightarrow val,\ db_q@p_i \\ send_{p_i \to p_0} response(val) \end{array}}{db_q@p_0 \vdash f_r(ParamList)@p_i \Rightarrow val,\ db_q@p_i} \qquad (\mathcal{R}'_{\mathcal{F}_r})$$

This rule specifies all XRPC calls at peer p_i belonging to the same query q, will be evaluated using the same database state $db_q@p_i$. This $db_q@p_i$ is the state of the database at the time when the first XRPC request belonging to q arrived at p_i. Observe that a unique query identifier q is now passed as an extra parameter in the XRPC request, such that a peer can recognize which XRPC calls belong to the same query [4]. Clearly, XRPC with repeatable reads requires more resources to implement, as some *database isolation* mechanism (of choice) will have to be applied. The transaction mechanism of MonetDB/XQuery, for example, uses snapshot isolation based on shadow paging, which keeps copies of modified pages around.

A quite common reason why a peer is called multiple times in the same query is when an XRPC call appears inside a for-loop. In Section 2.3 we describe how *loop-lifting* helps avoid these costly isolation measures in case of *simple* XRPC queries (i.e. those that contain only one non-nested function application) [5].

Updates. XRPC is an orthogonal language extension, thus it also allows *updating* user defined functions to be called. MonetDB/XQuery supports such updates conforming to the W3C XQuery Update Facility [11]. Thus, XRPC also enables complex P2P update transactions, as a query may contain multiple such updating function XRPC calls, each call may be nested, such that the same peer may be involved in the execution of multiple updating XRPC calls originating from the same query. In [29] we define two update semantics: a lax semantics where each updating XRPC call is executed and committed in isolation, and a strict variant, that supports repeatable reads and atomic distributed commit (this latter semantics requires a costly distributed commit protocol such as 2PC). Also, we describe a deterministic update order (the XUF leaves this order open) and describe an extension to our SOAP message format to guarantee correct deterministic update order in distributed updates in [29].

[4] Note that the choice of desired semantic rule is made per query, not per request.

[5] XRPC currently only has repeatable-reads support for simple XRPC queries, at zero cost.

2.3 Bulk RPC

One of the major contributions of XRPC is allowing *Bulk RPC operations*, in which a *single* XRPC request message is sent to the destination peer to perform *multiple* function calls. The results of such a Bulk RPC operation are sent back, again, in a single (response) message. Obviously, this approach can dramatically reduce network latency and band-width usage if the number of interactions with XRPC calls to the same peer is large. While Bulk RPC can in principle be applied in any XQuery implementation, it followed quite naturally from the translation technique used by the Pathfinder [15] compiler (that is used in MonetDB/XQuery), which is is capable of translating arbitrarily shaped XQuery expressions into a single relational plan, consisting only of set-a-time relational algebra expressions.

actor

iter	pos	item
1	1	"JulieAndrews"
2	1	"SeanConnery"

dst

iter	pos	item
1	1	"xrpc://y.org/"
2	1	"xrpc://y.org/"

Bulk RPC to a single peer. Our earlier example query (Q2) contains a function application inside a `for`–loop. In the relational XQuery translation approach of Pathfinder, the values of the variables `$dst` and `$actor` of all iterations inside this loop are represented by the three-column relational tables shown at the right side. The actual values of a variable are stored in the column `item`. The column `iter` is used to preserve the iteration order. The column `pos` is used to indicate the position of the value in an XQuery sequence. This *loop-lifting* technique is extensively explained in [8] and [15]. From the tables, it can be seen that the values of `$dst` are the same in both iterations of the `for`-loop, whereas `$actor` takes on values "`Julie Andrews`" in the first and "`Sean Connery`" in the second iteration. Both `pos` columns in our tables contain only the value `1` indicating that there are no multi-item sequences in the query.

For this query, XRPC generates *one* request message as shown below (only the `request` is shown). Each call is represented by an individual `call` child element:

```
<xrpc:request module="filmdb" location="http://x.org/film.xq" method="filmsByActor">
  <xrpc:call>          <!-- first call  -->
    <xrpc:sequence>
      <xrpc:atomic-value xsi:type="xs:string">Julie Andrews</xrpc:atomic-value>
    </xrpc:sequence>
  </xrpc:call>
  <xrpc:call>          <!-- second call -->
    <xrpc:sequence>
      <xrpc:atomic-value xsi:type="xs:string">Sean Connery</xrpc:atomic-value>
    </xrpc:sequence>
  </xrpc:call>
</xrpc:request>
```

Bulk RPC returns results of multiple calls in *one* `response` element, with one `sequence` element representing the result of each call:

```
<xrpc:response module="filmdb" method="filmsByActor">
  <xrpc:sequence/>     <!-- result of first call  -->
  <xrpc:sequence>      <!-- result of second call -->
    <xrpc:element><filmName>The Rock</filmName></xrpc:element>
    <xrpc:element><filmName>Goldfinger</filmName></xrpc:element>
  </xrpc:sequence>
  </xrpc:call>
</xrpc:request>
```

Bulk RPC to multiple peers. In the previous example the `execute at` expression `$dst` happened to be constant, such that all loop-lifted function calls had the same

destination peer, and could be handled by the single Bulk RPC request above. In the general case, however, there are multiple destination peers, which means that we have to split the input tables in sub-tables with the parameter values for each distinct destination peer. From these separate sub-inputs, we perform a separate (Bulk) RPC requests to each peer, *in parallel*. The results extracted from the response messages are then combined ($\cup$) and re-ordered [29].

In the remainder of the paper, we will define some additional semantics of XRPC functions as if there is only a single destination URI. This is done for simplicity; the technique of splitting at run-time all XRPC inputs per unique destination and handling them separately trivially applies to any case.

2.4 XRPC Implementation

The XRPC module in MonetDB/XQuery contains an ultra-light HTTP daemon implementation [19] that runs a request handler (the XRPC server), and contains a message sender API (the XRPC client). We also had to add support for the `execute at` syntax to the Pathfinder XQuery compiler, and change its code generator to generate *stub code* that invokes the new message sender API.

XRPC call processing re-uses the standard XML shredding and serialization provided by MonetDB/XQuery: first from the actual parameters a XML message is serialized into a HTTP post request, sent out using the message sender API. The return message is again shredded and the resulting relational tables are used as the result of the XRPC call. The request handler, on the other side, behaves similarly. It listens for SOAP requests and shreds incoming messages into a temporary relational table, from which the parameter values are extracted. The module function specified in the SOAP request is then executed locally with these parameter tables, producing a result table. The request handler then builds a response message in which this result table is serialized into XML onto the network socket.

Experiments. We conducted some simple and preliminary experiments to evaluate the performance of SOAP XRPC in MonetDB/XQuery. The test setup consisted of two 2GHz Athlon64 Linux machines connected on 100Mb/s ethernet. We defined a module with a trivial user defined function, that adds two integer parameters:

```
module namespace test="test";
declare function add($a as xs:integer, $b as xs:integer) as xs:integer
{ return $a + $b };
```

For each measurement, we executed the following function hundred times (the average elapsed time is reported):

```
import module namespace test="test" at "http://x.org/test.xq";
for $i in (1 to $x)                                                    (Q3)
  return execute at {"xrpc://y.org"} {test:add(20,22)}
```

While in MonetDB/XQuery loop-lifting of XRPC calls (i.e. Bulk RPC) is the default, we also implemented a single RPC at-a-time mechanism for comparison. Figure 1 shows the experiment where we compare performance of Bulk RPC with single RPC at-a-time, while varying

	$x=1	$x=1000
one-at-a-time	35	34979
bulk	35	400

Fig. 1. XRPC Performance (msec)

the number of loop iterations $x. It shows that performance is identical at $x=1, such that we can conclude that the overhead of Bulk RPC is small. At $x=1000, there is an enormous difference, mostly caused by network communication cost. This is easily explained as the single RPC at-a-time experiment involves performing 1000 times synchronous RPCs instead of 1.

3 P2P XQuery Semantics

MonetDB/XQuery* provides generic XQuery functionality, and its distributed querying and update facilities can be used in widely varying environments. First, we show how the mechanism described so far, can be useful in LAN environments with a limited number of nodes. When considering WAN applications with potentially thousands or more participating peers (such as StreetTiVo), we propose to use Distributed Hash Table (DHT) data structures under the hood of the system.

In the following, we will show how these widely varying application areas can be addressed by the fn:doc() and fn:put() built-in functions plus our XRPC execute at language construct.

3.1 Simple Scenarios

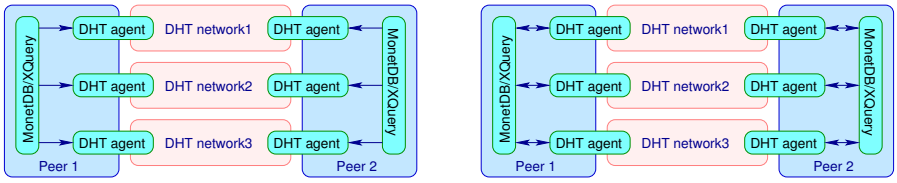
The XQuery 1.0 language [3] specifies a *data shipping* model to deal with remote XML documents. The built-in function fn:doc() retrieves an XML document stored in the peer identified by an xs:anyURI. In the XQuery Update Facility [11], a new built-in function fn:put() has been added, which stores an XML document node (or an XML element node) to the location specified by an xs:anyURI. Our XRPC extension for the XQuery language enables a *query shipping* model to query and manipulate remote XML documents. Given our choice for SOAP over HTTP as the network protocol for XRPC, it is interesting to note that the execute at construct, when combined with fn:doc() and fn:put(), provides an implementation of HTTP-based data shipping, as shown by the following rewriting rules:

$$\begin{array}{c} \texttt{fn:put}(\$node,\ \texttt{xrpc://}host/localname) \\ == \\ StatEnv.baseURI \Leftarrow \emptyset \\ \texttt{execute at } \{\ \texttt{xrpc://}host\ \}\ \{\ \texttt{fn:put}(\$node,\ localname)\ \} \end{array} \qquad (\mathcal{R}_{put_1})$$

$$\begin{array}{c} \texttt{fn:doc}(\texttt{xrpc://}host/localname) \\ == \\ StatEnv.baseURI \Leftarrow \emptyset \\ \texttt{execute at } \{\ \texttt{xrpc://}host\ \}\ \{\ \texttt{fn:doc}(localname)\ \} \end{array} \qquad (\mathcal{R}_{doc_1})$$

Thus, an XQuery system with XRPC can implement the HTTP protocol in fn:doc(), fn:put() internally by using XRPC to execute those requests remotely with the local part of the URI (and an empty "base-URI", from the static environment [4]).

Consumer Electronics Use Cases. Our AmbientDB research project originates from the desire in the consumer electronics domain to provide a common data management layer, that manages meta data found on mobile and stationary consumer electronics



(a) Loose DHT/DBMS Coupling (b) Tight DHT/DBMS Coupling

Fig. 2. MonetDB/XQuery* with multiple DHT connections

devices in and around the home (mobile phones, music players, PDAs, digital video recorders, PCs, etc). The fact that some of these devices are mobile and not always connected, and that manufacturers would prefer to provide functionality without the strict need to install a central server, squarely puts this area in the P2P domain, albeit in a LAN and a relatively small scale. Some of the applications found here (e.g. music meta data browsing, intelligent media synchronization and playlist statistics collection, analysis and recommendation) can already benefit from the functionalities of a system like MonetDB/XQuery* . More information about the envisioned AmbientDB applications can be found in [9, 13]. Note that in a LAN, peer discovery can still be handled by the application (using application-level protocols like UPnP [1]).

3.2 Loose DHT Coupling

A Distributed Hash Table [24, 5] 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 amount of peers in the network). A number of P2P database prototypes have already used DHTs [10, 16–18, 21]. An important design question is how a DHT should 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 destination 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 P2P XDBMS to participate in multiple (logical) DHTs simultaneously, as shown in Figure 2(a). The third part (key) is used to store and retrieve values in the DHT. The simplest architecture to couple a DHT network with a DBMS is to just use the DHT API, the `put(key,value)` and `get(key):value` functions, to implement the XQuery data shipping functions `fn:put()` and `fn:doc()`, as shown in rules $\mathcal{R}_{put_2}$ and $\mathcal{R}_{doc_2}$:

$$\frac{\begin{array}{c} p_i = dht\_hash_{dht\_id}(key) \\ dht\_send_{p_0 \to p_i} request(\text{“}\texttt{put}\text{”},\ (key, \$node)) \\ dht\_store@p_i \vdash \texttt{put}(\$node)@p_i \Rightarrow dht\_store'@p_i \\ dht\_send_{p_i \to p_0} response() \end{array}}{db@p_0 \vdash \texttt{fn:put}(\$node,\ \texttt{dht://}dht\_id/key) \Rightarrow db@p_0} \quad (\mathcal{R}_{put_2})$$

$$\frac{\begin{array}{c} p_i = dht\_hash_{dht\_id}(key) \\ dht\_send_{p_0 \to p_i} request(\text{“}\texttt{get}\text{”}, (key)) \\ dht\_store@p_i \vdash \texttt{get}(dht\_id/key) \Rightarrow \$node,\ dht\_store@p_i \\ dht\_send_{p_i \to p_0} response(\$node) \end{array}}{db@p_0 \vdash \texttt{fn:doc}(\texttt{dht://}dht\_id/key) \Rightarrow \$node,\ db@p_0} \quad (\mathcal{R}_{doc_2})$$

That is, we simply use the DHT to store XML documents as string values. The rules indicate that at the remote peer p_i, only the peer's DHT storage is involved, hence, peer p_i does not even have to have a running MonetDB/XQuery* instance. Note that the XQuery function `fn:doc` is a read-only function, since the document retrieved by using this function is stored as a transient document.

In this architecture, we can run the DHT as a separate process called the Local DHT Agent (LDA). Each LDA is a process that is connected to one DHT *dht_id* (see Figure 2(a)). This process runs separately from the database server, such that we can use the DHT software without any modifications.

The `execute at` can be "simulated" as follows:

$$\frac{\begin{array}{c} StatEnv.baseURI \Leftarrow \texttt{dht://}dht\_id/prefix \\ db@p_0 \vdash f_r(ParamList) \Rightarrow val,\ db@p_0 \end{array}}{db@p_0 \vdash f_r(ParamList)@\texttt{dht://}dht\_id/prefix \Rightarrow val,\ db@p_0} \quad (\mathcal{R}_{xrpc_2})$$

Rule $\mathcal{R}_{xrpc_2}$ in fact just evaluates the function locally, by getting all documents with a relative URI name from the DHT. This is achieved by setting the baseURI in the static environment to `dht://`*dht_id*/*prefix*. If the function body thus contains any `fn:doc()`, `fn:put()` on some relative URI *localname*, the rules $\mathcal{R}_{put_2}$ and $\mathcal{R}_{doc_2}$ specify that the document should be stored/retrieved into/from `dht://`*dht_id*/*prefix localname*. One should note that the *prefix* may be empty.

While this approach allows zero-effort coupling of DHT technology with DBMS technology, we consider it nothing more than a workaround. Rule $\mathcal{R}_{xrpc_2}$ substitutes function shipping by data shipping, defeating the purpose of XRPC. In case of updates, we would need to modify the rule to store the modified documents using `put` back in the DHT, but such a two-step update is hard to be made atomic.

3.3 Tight DHT Coupling

In a tight coupling scenario, rather than keep XML as string blobs inside the DHT (in RAM), each DHT peer actually uses its local XDBMS to store the documents (see Figure 2(b)). To realize this, we need to extend the DHT API with a single new method:

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

This new method allows the *request* in the below rule to be routed through the DHT (*dht_send*), to achieve the following semantics for XRPC calls to a "`dht://`" URI:

$$\frac{\begin{array}{c} p_i = dht\_hash_{dht\_id}(key) \\ dht\_send_{p_0 \to p_i} request(q, m, f_r, ParamList) \\ db@_q p_i \vdash f_r(ParamList)@p_i \Rightarrow val,\ db_q@p_i \\ dht\_send_{p_i \to p_0} response(val) \end{array}}{db_q@p_0 \vdash f_r(ParamList)@\texttt{dht://}dht\_id/key \Rightarrow val,\ db_q@p_0} \quad (\mathcal{R}_{xrpc_3})$$

This rule states that the DHT *dht_id* routes an XRPC request using the normal DHT routing mechanism towards the peer p_i responsible for *key*. When the Local DHT Agent

(LDA) in p_i receives such a request, it performs an XRPC to the MonetDB/XQuery* instance on the same peer p_i. This XRPC executed at remote location p_i from the LDA into MonetDB/XQuery* (it may use either semantic $\mathcal{R}_{\mathcal{F}_r}$ or $\mathcal{R}'_{\mathcal{F}_r}$). The response is then transported back via the DHT towards the query originator p_0.

In this scenario, we can support `fn:doc()` and `fn:put()` by combining rule $\mathcal{R}_{xrpc_3}$ with $\mathcal{R}_{doc_1}$ and $\mathcal{R}_{put_1}$. That is, use an XRPC request routed via the DHT to do a remote execution of `fn:doc()`, `fn:put()` on the relative URI *localname*.

In the tight coupling, we have to extend the DHT implementation. A positive side-effect of this is that the DBMS gets access to to information internal to the P2P network. This information (e.g. peer resources, connectivity) can be exploited in query optimization. Also, bulk XRPC requests routed over the DHT may be optimized (similar to Bulk RPC), by combining requests that follow the same route as long as possible in single network messages.

StreetTiVo Use Cases. We show how two uses cases in the StreetTiVo application can be implemented as XQuery module functions, which then can be executed using XRPC and tightly coupled DHT semantics.

(i) Collaborator Discovery. In StreetTiVo, compute-intensive video analysis work is distributed to all peers p_i that record the same TV program. Every TV program's video data is divided into segments of e.g. 30 seconds. Each peer p_i analyses one segment at a time. If a peer p_0 has finished retrieving the meta data of a segment s_i, it sends the meta data to those peers that record the same TV program, so that the other peers do not have to analyse the same segment themselves. Peer p_0 then tries to find out if there are more segments need to be analysed, if yes, the peer p_0 claims the segment s_j and analyse it. These steps are repeated until all segments of a TV program are analysed.

In this scenario, peer p_0 needs to know which other peers record the same program (its collaborators). This can be implemented as the following. Assume that every TV program has a unique identifier `progid`, then we creates for each recorded TV program $prog_i$ an XML document with the name "`<progid>.xml`" to keep a list of `pid`-s of peers that record this program. Store the document "`progid.xml`" in a peer in the DHT network and later retrieve it can be done by the calls:

```
fn:put($node, "dht://dht_id/progid.xml"),  and
fn:doc("dht://dht_id/progid.xml").
```

If a peer is going to record the TV program `progid`, it should add its peer ID to "`progid.xml`". This can be typically done by an XRPC call like:

```
execute at {"dht://dht_id/progid.xml"} {xrpc:addPID($pid)}
```

(ii) Distributed Keyword Retrieval. One of the features provided by StreetTiVo is Automatic Speech Recognition (ASR) on TV programs. The resulting text is stored as meta data for the recording, and can be searched using text retrieval functionality in MonetDB/XQuery* . Each peer should have access to a local XQuery module with the user-defined function `searchKeywords` that gets two parameters. The first parameter `pid` identifies the program in which text fragment should be searched. The second parameter `keywords` is a list of search keywords.

Image this scenario: a StreetTiVo user wants to search in the today's newscast for text fragments that were about the earthquake in Hawaii, but he/she did not record the newscast. Then the search request needs to be send to other StreetTiVo peers that

have recorded the newscast. Assume the newscast's program ID is "newscast123", this scenario can be implemented by the following pseudo-code:

```
let $peers := execute at { "dht://dht_id/newscast123" } { xrpc:getPIDs() }
for $p in $peers
  execute at { $p } { xrpc:searchKeywords("newscast123", ("earthquake", "hawaii")) }
```

3.4 Research Questions

The mission of the AmbientDB project at large is:

> ***build a large-scale Peer-to-Peer middleware XML database management system, which hides the heterogeneity of underlying database systems and communications networks, and provides a uniform programming interface to ease the development of the ambient intelligent applications.***

The final objective to unite data from heterogeneous sources recognizes the importance of schema and data integration in P2P data management – issues not addressed here. The AmbientDB project is being conducted together with Philips Research and Tech. Univ. Twente, and this data integration aspect is being pursued there with special attention for the probabilistic nature of the result of such data/schema integration [28].

At CWI, we focus on the query and update processing research questions, such as:

- *Which information and API should a DHT expose to a query processing engine such that it can effectively optimize P2P queries? How can query optimization be pursued effectively without having any global statistics? How can we balance query optimization effort with query execution effort in a P2P setting?*
- *How can it be guaranteed that the distributed execution of a query will terminate in a finite time. When it is the right moment to decide a query has produced enough result? How can an error be handled properly in MonetDB/XQuery* ? When should an execution be aborted due to errors?*
- *Which consistency constraints can be relaxed for different kinds of applications? What are the trade-offs between location transparency and efficiency?*

4 Related Work

P2P networks active topic in networking research, especially Distributed Hash Tables, such as Chord [26]. For practical use, the systems Bamboo [23] and P-Grid [5] currently seem to be the most usable.

[14] is one of the first papers that discuss database management issues in a P2P environment. PIER [16] 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 [18] provides an (RDF-like) triple storage on top of P-Grid. XPeer [25] 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. [21] is a proposal for a XML-based database system on top of a DHT, which is also named XPeer. 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, several proposals are close to our work ([22, 7, 6, 20, 27]). The syntax of XRPC is based on that of XQueryD [22], but in a more restricted form. The XQueryD approach requires a runtime rewriter to scan the XQuery expressions in the `execute` statement for variables and substitute the variables with the current runtime values. Such a rewriter is not needed in XRPC, and by relying on function resolution in modules, query transportation is simplified and queries may benefit from pre-processing. In Active XML ([7, 6]), calls to service functions are embedded in XML documents. Galax Yoo-Hoo! [20] 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 (although messages must be manipulated explicitly with element construction). DXQ [27] is a specification of a Distributed XML-Query network.

5 Conclusion

In this paper, we discussed work on MonetDB/XQuery* that aims to create powerful P2P XML database technology that preserves the full XQuery language (+XUF), extending it only with a single new construct called XRPC. [6] We first gave a formal definition of the syntax and the semantics of XRPC, and showed how it can be implemented efficiently using set-at-a-time processing (Bulk RPC). A small experimental section demonstrated that Bulk RPC strongly improves performance of queries that execute XRPC calls inside `for`-loops. We then described how Distributed Hash Tables (DHTs) can be integrated without further XQuery extensions, by adding support for a new `dht://` protocol in URIs. We discussed the semantics of two ways of coupling (loose and tight) a DHT with an XDBMS, of which the latter is more powerful. This was shown by elaborating in some detail how this functionality can be used in the StreetTiVo collaborative video indexing application.

Our next step is to implement these couplings in MonetDB/XQuery* using the Bamboo DHT [24], and perform experiments in environments like PlanetLab. Especially the tight coupling will open up a playing field for a number of query optimization techniques that exploit the P2P network characteristics. This is only one of the many research questions in AmbientDB, of which we provided a non-exhaustive overview.

References

1. UPnP: Universal Plug and Play. http://www.upnp.org.
2. SOAP Version 1.2 Part 0: Primer. W3C Recommendation 24 June, 2003. http://www.w3.org/TR/2003/REC-soap12-part0-20030624.
3. XQuery 1.0: An XML Query Language. W3C Candidate Recommendation 8 June, 2006. http://www.w3.org/TR/2006/CR-xquery-20060608.
4. 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.

[6] The next open-source release of MonetDB/XQuery including XRPC (download from `http://monetdb.cwi.nl/XQuery`) is coming soon.

5. K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *CooplS '01*, London, UK, 2001. Springer-Verlag.
6. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic xml documents with distribution and replication. In *SIGMOD Conf.*, 2003.
7. S. Abiteboul, I. Manolescu, and E. Taropa. A Framework for Distributed XML Data Management. In *Intl. Conf. on Extending Database Technology*, 2006.
8. P. Boncz, T. Grust, M. vKeulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a RDBMS. In *SIGMOD*, 2006.
9. P. A. Boncz. AmbientDB: P2P Database Technology for Ambient Intelligent Multimedia Applications. *ERCIM News*, (55), 2003.
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. 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.
12. M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Candidate Recommendation 11 July, 2006. http://www.w3.org/TR/2006/CR-xpath-datamodel-20060711.
13. W. Fontijn and P. A. Boncz. AmbientDB: P2P Data Management Middleware for Ambient Intelligence. In *PERWARE*, Orlando, FL, USA, 2004.
14. 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*, Santa Barbara, CA, USA, 2001.
15. T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *VLDB*, 2004.
16. 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.
17. R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.
18. 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.
19. S. Lyubka. SHTTPD: Simple HTTPD. http://shttpd.sourceforge.net.
20. N. Onose and J. Siméon. XQuery at Your Web Service. In *WWW*, 2004.
21. W. Rao, H. Song, and F. Ma. Querying XML Data over DHT System Using XPeer. In *Grid and Cooperative Computing - GCC 2004*, 2004.
22. C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *IIWeb*, 2004.
23. S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *USENIX WORLDS'05*, 2005.
24. S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference, General Track*, 2004.
25. C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. XPeer: A Self-Organizing XML P2P Database System. In *EDBT Workshops*, 2004.
26. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, 2001.
27. C. Thiemann, M. Schlenker, and T. Severiens. Proposed Specification of a Distributed XML-Query Network. *CoRR*, cs.DC/0309022, 2003.
28. M. van Keulen, A. de Keijzer, and W. Alink. A probabilistic XML approach to data integration. In *Proceedings of the International Conference on Data Engineering (ICDE), 5-8 April 2005, Tokyo Japan*, pages 459–470. IEEE Computer Society, April 2005.
29. Y. Zhang and P. A. Boncz. Loop-Lifted XQuery RPC with Deterministic Updates. Technical Report INS-E0607, CWI, Amsterdam, The Netherlands, 2006.