XRPC: efficient distributed query processing on heterogeneous XQuery engines
Zhang, Y.

Citation for published version (APA):
Zhang, Y. (2010). XRPC: efficient distributed query processing on heterogeneous XQuery engines

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Distributed XQuery With XRPC

4.1 Introduction

In this section, we discuss various uses of XRPC for distributed XQuery processing on heterogeneous XQuery engines.

First, we show that XRPC is not system-specific: every XQuery data source can service XRPC calls using a simple wrapper. Since XQuery is a pure functional language, we can leverage techniques developed for functional query decomposition to rewrite data shipping queries into XRPC-based function shipping queries. Powerful distributed database techniques (such as semi-join optimisations) map directly onto Bulk RPC, opening up interesting future work opportunities. We demonstrate this with experiments in which MonetDB/XQuery and Saxon work together over XRPC.

Second, we turn our attention to the interaction between XRPC and XQUF. We first define a deterministic distributed update semantics and show that a small extension to the SOAP XRPC protocol enables the protocol to conform to the deterministic update semantics. We then describe how the industry standard Web Service Atomic Transaction[55] can be adapted to support atomic distributed commits of XQUF queries on heterogeneous XQuery engines.

While XRPC already allows XQuery engines to perform P2P queries, it still misses a number of vital P2P functionalities (robust connectivity, peer and resource discovery, approximate query/transaction processing). In the final part of this chapter, we present preliminary work on MonetDB/XQuery*, in which we integrate existing XDBMS and P2P structures to provide P2P data management facilities.

XRPC as Target Language One of the design goals of XRPC is – besides it being directly useful as an explicit instrument to write distributed queries – to have it serve as the target language for a distributed XQuery optimiser that takes queries without XRPC as input (thus data shipping only), and produces decomposed queries as output that use XRPC for function shipping. Our choice to make distributed execution explicit in terms of remote functions and their dependencies (parameters), aligns well with XQuery being a pure functional language. Query decomposition techniques [105] can thus be applied to decompose the full query (function) into sub-queries (again functions), that each can in theory be executed on any of the participating sites.

Automatic query decomposition techniques are discussed in the next chapter. In this chapter, we limit ourselves to showing how some well-known distributed query execution strategies, such as the distributed semi-join strategy, can be elegantly expressed in XRPC.
4.1. INTRODUCTION

To demonstrate the performance opportunities of XRPC, as well as its interoperability, we provide some initial performance experiments with one peer running MonetDB/XQuery, and another running Saxon.

The implementation of XRPC in the open-source XML database system MonetDB/XQuery (http://monetdb.cwi.nl) already allows query writers to experiment with distributed query processing strategies, but we show using Saxon that even without XRPC being integrated into other XQuery systems, we can achieve our goal of cross-system distributed querying using an XRPC wrapper (Section 4.2). This wrapper is a SOAP service handler which generates an XQuery query that uses the incoming SOAP request message as an input, iterates over all function call requests in it, applying the local XQuery function on the supplied parameters, and uses element construction to produce a SOAP response message that is sent back by the wrapper.

One should note that the capabilities of such an XRPC wrapper outclass that of the well-known wrapper architecture applied in federated database systems [114]. Not only can this architecture do without a centralised integrating engine (XRPC allows for true P2P query processing), we will also show that the possibility to submit sets of requests (that can each have sequence-typed correlated parameters) allows query writers to, e.g., express the well-known distributed query processing strategy of semi-join reduction by simply passing a key parameter to the remotely called function.

Deterministic Updates  The W3C Candidate Recommendation proposal for the XQuery Update Facility leaves it undetermined how to handle multiple updates to the same node. For example, if we have an XML document ⟨a⟩ named “a.xml”, then its value after executing the update expression

```
for $n in (<b/>,<c/>) return insert node $n as first into doc("a.xml")
```

can be either ⟨a⟩⟨b/⟩⟨c/⟩⟨/a⟩ or ⟨a⟩⟨c/⟩⟨b/⟩⟨/a⟩. Arguably, this semantics does not match the transactional semantics in databases very well. In MonetDB/XQuery, we thus choose to implement the XQUF deterministically, by respecting the for-loop order (respectively the sequence construction order), in which the multiple update statements occur in the query (in the above case yielding ⟨a⟩⟨b/⟩⟨c/⟩⟨/a⟩).

The question we address here is how to achieve deterministic semantics in distributed updates using our loop-lifted RPC technique. Note that XRPC is fully orthogonal to XQuery, thus it is allowed to call user-defined updating functions over XRPC. Updating functions can contain for-loops and sequence constructors, which might again make (multiple) other XRPC updating function calls to other peers. Thus, distributed update queries generally involve a group of peers and within a single query the same peer may even be involved multiple times, potentially through different function call sequences. Our loop-lifting approach to XRPC (“bulk” RPC requests) changes the order in which RPC function calls are evaluated. This means that the order in which updates must be applied may differ from the order in which the XRPC function calls were received. To address this issue, we formulate an extension to our bulk SOAP XRPC protocol that allows keeping track of deterministic update order, while conserving the performance advantages of loop-lifted RPC.

Distributed Transactions  During a single XRPC query, it may happen that multiple read-only XRPC requests are sent to the same site. In the repeatable read isolation level we define, each request from the same query is guaranteed to see the same database state. XRPC queries may themselves also update the databases by invoking XQUF “updating functions”
import module namespace func = "functions" at "http://example.org/functions.xq";
declare namespace env = "http://www.w3.org/2003/05/soap-envelope";
declare namespace xrpc = "http://monetdb.cwi.nl/XQuery";
⟨env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
 xmlns:xrpc="http://monetdb.cwi.nl/XQuery"
 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:response xrpc:module="functions" xrpc:method="getPerson"⟩
{for $call in doc("/tmp/request_nnn.xml")//xrpc:call
let $param1 := n2s($call/xrpc:sequence[1])
let $param2 := n2s($call/xrpc:sequence[2])
return s2n(func:getPerson($param1, $param2))}
⟨/xrpc:response⟩
⟨/env:Body⟩
⟨/env:Envelope⟩

Table 4.1: XQuery generated for the getPerson() XRPC request

over XRPC. Note that XQUF queries only perform side-effecting actions after all query execution has finished, such that during query execution the database state is constant, and updating queries behave much like read-only queries. Obviously, atomically committing a distributed transaction requires a protocol like two-Phase Commit (2PC). We decided not to add 2PC to the XRPC network protocol, but rather rely on the recent industry standard Web Services Atomic Transaction (WS-AtomicTransaction) [55, 54] that provides exactly this feature for distributed web-service transactions.

Integrating XQuery and P2P Our approach to equip XRPC with P2P facilities is to integrate services offered by diverse P2P network structures, such as the Distributed Hash Tables (DHTs), into existing XDBMS. In MonetDB/XQuery⋆, we propose different ways of integration that avoid any further intrusion into the XQuery language and semantics. We also show how the proposed approaches, similarly to Bulk RPC, will lead to further query optimisation opportunities where the XDBMS interacts with the underlying P2P network. XRPC and MonetDB/XQuery⋆ are adopted by StreetTiVo, a P2P collaborative video analysis and metadata distribution application. We discuss the architecture of StreetTiVo in Chapter 7 and show how XRPC and MonetDB/XQuery⋆ enable quick development of complex P2P application such as StreetTiVo.

4.2 Cross-System Distributed XQuery

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 in Figure 4.1. 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 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 its underlying XQuery engine to handle
calls with normal XRPC-incapable systems, but obviously does not allow making outgoing XRPC calls from them.

We illustrate how such an XRPC wrapper works by an example. The following function returns the person node from an XMark document ($doc) whose $id attribute matches a given $pid:

```xquery
declare function getPerson($doc as xs:string, $pid as xs:string) as node()? {
    zero-or-one(doc($doc)//person@id=$pid);
}
```

Table 4.1 shows the query generated by an XRPC wrapper to handle the getPerson() request.

The XRPC protocol includes information about the arity of the function (as well as its return type), so it is easy to generate the right number of param parameters in the call. The brunt of the work is done by the n2s() and s2n() marshalling functions, introduced in Section 3.4.1. These functions can be implemented purely in XQuery.

The n2s() function, used here to process all parameters, converts a SOAP XRPC element into an item sequence, where each item has the right type. This is done by going over all children of the xrpc:sequence using a series of if..then XQuery statements that select on the xsi:type attribute found in the xrpc:atomic-value nodes. In case of xrpc:element nodes with an xsi:type, XQuery validation is performed. The s2n() function is used here only to convert the function return value into a correct SOAP XRPC node. It iterates over the input item sequence, and for each item uses an XQuery typeswitch() to generate the right SOAP node. If the return type is a sequence of nodes that have a schema type (this information is supplied in the SOAP request) we insert the correct xsi:type attribute in it.

**Saxon Experiments** Using the wrapper, we can run a number of experiments on the Saxon XSLT/XQuery processor [109] (Saxon-B 8.7). The results are shown in Table 4.2. Like the experiments in Section 3.5.3, we put the execute at inside a for-loop with a varying number of iterations ($x$) to study the performance impact of Bulk RPC. By 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. For the echoVoid experiment, we see that Bulk RPC again allows amortising XRPC latency really well: instead of 1000 times the latency, with a 1000 times more work, total latency increases just over a factor of 2. As the execution time still is increased by a factor of 30, the low impact is due to other amortised latencies, in parsing the XML request document, compiling the query, etc.

We also show the results of the getPerson() example above. 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(), that 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

\[
\begin{array}{|c|c|c|c|}
\hline
\text{call} & \text{total} & \text{compile} & \text{treebuild} & \text{exec} \\
\hline
\text{echoVoid $x=1$} & 275 & 178 & 4.6 & 92 \\
\text{echoVoid $x=1000$} & 590 & 178 & 86 & 325 \\
\hline
\text{getPerson $x=1$} & 4276 & 185 & 1956 & 2134 \\
\text{getPerson $x=1000$} & 8167 & 185 & 1973 & 6010 \\
\hline
\end{array}
\]

Table 4.2: Saxon latency via the XRPC Wrapper (msec)
that performance remains linear in the size of the XMark document, just like it was in the single call selection.

4.3 Distributed XQuery Optimisation

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 (hence, only data shipping) and produces a decomposed query as output that uses XRPC for function shipping. In this section, we show how some well-known distributed query execution strategies, such as distributed semi-join, can be elegantly expressed in XRPC. We also outline several future work issues in the area of automatic query distribution techniques using functional decomposition.

Let us assume a distributed XDBMS system 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 (open/closed) auctions in “auctions.xml”.

for $p$ in doc("persons.xml")//person, $ca$ in doc("xrpc://B/auctions.xml")//closed_auction
where $p/@id = $ca/buyer/@person
return <result>{$p,$ca/annotation}</result> (Q4-1)

The above query is executed at peer \( p_a \). For each person and for every item this person has bought, query Q7 returns the person node and the annotation node of the bought item in a new result node. For the moment, assume that \( fn:\text{doc}() \) is invoked with a compile-time known constant URI from our \( xrpc:// \) URI name scheme, indicating that the peer is known to support XRPC.

**Predicate Pushdown** A first heuristic optimisation is to push predicates that depend only on a single \( fn:\text{doc}() \) application into data source \( p \). Thus, instead of transferring the whole document “auctions.xml” from \( p_b \) to \( p_a \), we define a function to return all closed_auction nodes and execute this function on \( p_b \):

```xml
module namespace b = "functions_b";
declare function b:Q_B1() as node()*
doc("auctions.xml")//closed_auction;
import module namespace b="functions_b" at "http://example.org/b.xq";
for $p$ in doc("persons.xml")//person, $ca$ in execute at {"B"} {b:Q_B1()},
where $p/@id = $ca/buyer/@person
return <result>{$p,$ca/annotation}</result>
```

**Rewritten query Q4-1-1**

This heuristic rewrite can simply be triggered by the presence of \( fn:\text{doc}() \). The required analysis to determine how much of the XQuery (Core) expression is dependent on that \( fn:\text{doc}() \) alone, and therefore can be pushed, is highly similar to the analysis method developed for XML projection [125].

**Advanced Pushdown** We could push expressions that depend on a \( fn:\text{doc}() \) application even if that function application has a non-constant URL argument, and even could depend on a for-loop variable. That is, using the helper functions:

```xml
declare function xrpc:host ($url as xs:string) as xs:string
declare function xrpc:path ($url as xs:string) as xs:string
```

where by default \( host() \) returns “localhost” and \( path() \) returns its argument – except for \( xrpc:// \) URLs, where they would separate the URL in a host prefix and path suffix – we could rewrite calls to \( fn:\text{doc}($url) \) into:

```xml
execute at {xrpc:host($url)} {fn:doc(xrpc:path($url))}
```
However, this approach does require a refinement of the work in [125]. One must bear in mind that any of the rewrites discussed here should only be made by an automatic rewriter if it can establish that the call-by-value semantics of XRPC will not compromise the semantics of the query. This at least involves a check whether nodes that come from pushed expressions are only navigated downwards, and also involves checking against node identity tests and order-dependent (e.g., order by) processing of node sequences that stem from multiple fn:doc() calls pushed to different sources.

**Execution Relocation** The possibilities of query rewriting do not stop at push-down of fn:doc('xrpc://...')-dependent expressions. Even if a query depends on a set \( P \) of XRPC peers that contribute documents, one could decide to select one peer \( p_i \) from \( P \) and put all execution on \( p_i \). We call this mechanism *Execution Relocation*. For example, it might be beneficial to relocate all execution on \( p_b \), if “auctions.xml” is much larger than “persons.xml”:

```xquery
module namespace b = "functions_b";
declare function b:Q_B2() as node()*
  {for $p in doc("xrpc://A/persons.xml")//person,
   $ca in doc("auctions.xml")//closed_auction
   where $p/@id = $ca/buyer/@person
   return <result>{$p, $ca/annotation}</result>};
```

Then peer \( p_a \) needs only to call this function to get the results:

```xquery
import module namespace b="functions_b" at "http://example.org/b.xq";
execute at {"B"} {b:Q_B2()}
```

**Distributed Semi-Join** The classical distributed semi-join strategy [25, 178] 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 can be passed in a loop to a function executed at \( p_b \) that returns those closed auctions with buyers having that \( @id \):

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

```xquery
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)}
return if(empty($ca)) then () else <result>{$p, $ca/annotation}</result>
```

Rewritten query Q4-1-3

This shows that federating data sources with XRPC (even via the XRPC Wrapper) is more powerful than the “wrapper-architecture” [114] used in federated database systems. Such wrappers typically lack the possibility to push table-valued parameters into data sources, which is required for the semi-join optimisations. It is worth pointing out that the loop-lifted implementation of XRPC is essential for the efficiency of the distributed query plans discussed in this section. The XRPC calls in the inner for-loop of the rewritten query \( Q7-1 \) and \( Q7-3 \) require only one message exchange between \( p_a \) and \( p_b \). Without the loop-lifted implementation, the network can easily get flooded by the huge amount of messages.

**Saxon and MonetDB/XQuery Joined by XRPC** To demonstrate the interoperability, expressiveness and performance potential of XRPC we run query \( Q7 \) on two peers using all four mentioned strategies. On peer \( p_a \) (the local peer), we run MonetDB/XQuery with the document “persons.xml” (1.1MB, 250 person nodes); on peer \( p_b \) the Saxon XSLT/XQuery processor with the document “auctions.xml” (50MB, 4875 closed_auction nodes). There are 6 matches between the person nodes and the closed_auction nodes.
Table 4.3: Execution time (msecs) of query Q4-1 distributed on MonetDB/XQuery and Saxon (Saxon time includes network).

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Total Time</th>
<th>MonetDB Time</th>
<th>Saxon Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>data shipping</td>
<td>28122</td>
<td>16457</td>
<td>11665</td>
</tr>
<tr>
<td>predicate push-down</td>
<td>25799</td>
<td>2961</td>
<td>22838</td>
</tr>
<tr>
<td>execution relocation</td>
<td>53184</td>
<td>69</td>
<td>53115</td>
</tr>
<tr>
<td>distributed semi-join</td>
<td>10278</td>
<td>118</td>
<td>10160</td>
</tr>
</tbody>
</table>

All communication between MonetDB/XQuery and Saxon happens via XRPC. The XRPC wrapper described in Section 4.2 is used to generate the XQuery query from an XRPC request message.

The measured execution times are shown in Table 4.3. In the column “MonetDB Time” are execution times on peer \( p_a \) and in the column “Saxon Time” are 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 (where it takes longer than on MonetDB). The “distributed semi-join” is the strategy that incurs least data shipping, and is most efficient in this case.

### 4.4 Deterministic Distributed Updates

The W3C Candidate Recommendation of the XQUF [58] does not determine the ordering among newly inserted nodes if those nodes are inserted into the same target node using the same kind of insert expression (into or as first/last or into before/after). The Candidate Recommendation specifies that this ordering is implementation-dependent.

**Definition of Deterministic Updates** The motivation in MonetDB/XQuery to exercise our liberty to implement the XQUF deterministically, is simply that order matters in XML. The solution chosen is that if the XQUF working draft leaves the ordering of updates actions undetermined, we respect the order in the pending update list. The XQUF working draft specifies how this list is built up incrementally. For two XQuery language constructs, namely for-loops and sequence construction, the working draft states that two pending updates must be merged with the `upd:mergeUpdates()` internal function. The XQUF leaves the working of this function unspecified, and our solution is to implement it with *concatenation*. Thus, each new pending update sublist (second parameter of `upd:mergeUpdates()`) is appended to the existing list (its first parameter). Note that this definition of update order is “intuitive” in that it respects the for-loop iteration order, as well as sequence construction order. Our rules \( f_u \) and \( f'_u \) further lead to synchronous function call semantics when updating functions are called over XRPC.

**The Challenge** Now that the MonetDB/XQuery implementation of XQUF cares about the update order, our challenge is to extend this deterministic update semantics to distributed updates. In the end of Section 3.5.2, we showed an example query that executed two Bulk RPCs on the same peer, and discussed how our loop-lifting technique causes the function to
be evaluated out of the intuitive order (this intuitive order is also followed by the XQUF to build the pending update list). The below query is the updating equivalent of that previous example, now using a hypothetical updating function `appendLog`, that appends entries to a log:

```xml
import module namespace film="filmdb" at "http://x.example.org/film.xq";
for $name in ("Julie", "Sean")
let $connery := concat($name, " ", "Connery")
let $andrews := concat($name, " ", "Andrews")
return (execute at {"xrpc://y.example.org"} {film:appendLog($connery)},
execute at {"xrpc://y.example.org"} {film:appendLog($andrews)})
```

Our deterministic XQUF requires us to write first two Julie entries in the log, followed by two Sean entries. The loop-lifting, however, will process the two Connery invocations first, followed by the two Andrews. In this section, we describe an extension to the SOAP XRPC message format that allows re-ordering the pending update list at commit time such that the correct update order is followed.

### 4.4.1 Order-Correct Update Tags

We start by characterising the update actions $a$ on behalf of query $q$ that may be found in the pending update lists $\Delta_q @ p$ at the various peers $p$. Second, we define a conceptual Distributed Pending Update Table (DPUT), that holds all $(p, a)$ combinations in the required order. Then, we define an additional third $T$ column for the DPUT that holds a tag, and explain how these tag values are constructed. We show that this $T$ column will always appear in sorted order, given that the DPUT contains the required output order. From this, we can then conclude that if each peer orders its local $\Delta_q @ p$ on $T$ just before commit, it will apply the update actions in the correct order. As a last step we show how the tags are constructed during query execution and passed between peers using a small (and final) extension to the SOAP XRPC message protocol.

**Update Actions** There are four groups of updating primitives described in [58]:

- **insert expressions** = `{upd:insertInto, upd:insertIntoAsFirst, upd:insertIntoAsLast, upd:insertBefore, upd:insertAfter, upd:insertAttributes};
- **delete expressions** = `{upd:delete};
- **rename expressions** = `{upd:rename};
- **replace expressions** = `{upd:replaceNode, upd:replaceValue, upd:replaceElementContent}.

For our purposes here, we abstract from these different groups and consider them as single update actions, denoted $A$. We denote $A$ the set of all update actions. Composite update actions, denoted $A_c$, are calls to an updating function, which itself can perform one or more update actions $\in A$. We have $A \equiv A_s \cup A_c$.

**Distributed Pending Update Table** Imagine that all update actions caused in a distributed update query are put in the correct deterministic update order, and attach to this global list $\Delta$ an additional peer column $P$. The resulting table $\Delta P$ we call the Distributed Pending Update Table (DPUT). We should stress that this is a conceptual table only, we do not propose to materialise such a table in any way.

In Section 3.4 we described that when an updating XRPC query is started with isolation (i.e., following the semantics defined by $F_u'$), each peer $p$ keeps an isolated environment
Corollary 4.4.1. Iteratively substituting each \(<p_x, f_{i_1}@p_x>\) in \(qid@p_0\) by sublist \(f_{i_2}@p_x\), yields the DPUT in required order.

Figure 4.2 shows \(qid@p_0\) and all \(f_{i_2}@p_x\) caused by a single query, and the DPUT derived from those (the right-most table). In the lists, values \(i\) point to another pending update sublist, that represents all update actions caused by the called function \(f_{i_2}\) at the peer in column \(p\). The iterative substitution of the sublists in DPUT achieves the required \(synchronous\) semantics for remote function calls, as it inserts all update actions (recursively) caused by a function call in the DPUT at the point where the remote function was applied.

**Body and Tags** The XQUF restricts the locations in a query where update actions can be done. We abstract from the full XQuery syntax using the body concept, to define these places. body refers to the body of an updating XRPC query or the body of an updating XRPC function. The body grammar is shown below:

\[
\text{body ::= UpdateAction | "for" ... "return" body | body ("," body)}\]

A body can contain an expression in one of the three types, (i) an update action (possibly an XRPC updating function), (ii) a for expression which in turn contains a body in its return clause, or (iii) a sequence of one or more bodys.

The tags in column \(T\) of the DPUT are concatenations of numbers, separated by a dot. We initialise \(t_{\text{prefix}} = 1\) for executions done locally on behalf of the initiating query. The query body mimics the parse tree of the query, which is then “executed” recursively as follows (starting with \(b=\text{root}\) and \(t_b = 0\)) to generate all tags:

- if \(b\) is a sequence constructor, we process all sequence expressions \(s_1, \ldots, s_n\) while assigning \(t_{s_i} = t_{b,i}\). 

![Figure 4.2: The conceptual Distributed Pending Update Table](image-url)
### 4.4. Deterministic Distributed Updates

#### Figure 4.3: The body of the example query and its DPUT

```plaintext
for $s$ in ("str1", "str2")
    return{
        execute at {$p_1$} {updFun_1($s)}
    }
```

<table>
<thead>
<tr>
<th>$\Delta_q@p_1$</th>
<th>$\mathcal{P}$</th>
<th>$\mathcal{A}$</th>
<th>$\mathcal{T}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;xrpc://y.example.org&quot; appendLog(&quot;Julie Connery&quot;)</td>
<td>$f_{id_1}@p_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;xrpc://y.example.org&quot; appendLog(&quot;Sean Connery&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;xrpc://y.example.org&quot; appendLog(&quot;Julie Andrews&quot;)</td>
<td>$f_{id_2}@p_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;xrpc://y.example.org&quot; appendLog(&quot;Sean Andrews&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.4: The pending update list $\Delta_q@p_1$ was created by two XRPC calls executed after each other. Sorting those at commit time on $\mathcal{T}$ achieves deterministic update order.**

- if $b$ is a for-loop with iterations $1 \leq i \leq n$, we process each iteration of the body $f$ with $t_f = t_b.i$.
- if $b$ is an updating action, we put $tag = t_{prefix}t_b$ in column $\mathcal{T}$ for all update actions it inserts in the pending update list.
- if $b$ is an updating XRPC function, we also insert $tag$ as an attribute of the xrpc:call in the XRPC request. The updating function body is executed remotely with initialisation $t_{prefix} = tag$.

Figure 4.3 shows how the tags are constructed from the body of our example update query. The initial $t_{prefix}$ is 1. The for-loop with two iterations introduces the second number, 1 for the first iteration, and 2 for the second. Inside the loop body we find a sequence constructor, introducing a third number in the tag. Inside this sequence constructor, the update actions are found and tagged.

Note that the tag construction algorithm respects the for-loop and sequence construction order just like XQUF pending update list construction. Also, the tags generated by remote function applications are prefixed by the current tag and therefore must be bigger than all previous and smaller than all following locally generated tags, which mimics synchronous XRPC semantics. Therefore:

**Corollary 4.4.2.** Column $\mathcal{T}$ in DPUT is ordered by definition.

One should remember that the DPUT is only a concept used to define the required order, and there is no single place where we can afford to bring together the entire merged pending update list – each peer only has local information. But, if we could attach the correct tag values to the (partial) pending update lists $\Delta_q@p$ at each peer $p$ in a $\mathcal{T}$ column, we can achieve correct update order by (stable) sorting the $\Delta_q@p$ on $\mathcal{T}$ locally at each peer at commit time.

**XRPC SOAP Extension: Tag Attributes** The tags are only constructed on demand, just before executing a Bulk RPC request. In local execution, the iter columns maintained by
MonetDB/XQuery for loop-lifting correspond with the iteration numbers in the tags. Thus by obtaining all `iter` numbers from the current scope through to the root level (by joining with so-called `map` relations [88]), the tags can be constructed whenever an update action needs to be executed. For sequence construction, these numbers are available in the Pathfinder XQuery Core parse tree, and can be inserted in the generated query plan. The tags are always prefixed by `t_prefix`, stored as a loop-lifted expression. The reconstructed tags are included as attributes in the `xrpc:call` elements in the Bulk SOAP XRPC request message. The remote peer uses this tag then as prefix for generating further tag numbers, as described before (i.e., as the loop-lifted `t_prefix` expression).

Below we show the first XRPC request message triggered by the RPC call in our example query, which leads to tags 1.1.1 and 1.2.1 (i.e., `t_prefix.iter{1,2}.seq1`):

```xml
<xrpc:request xrpc:module="filmdb" xrpc:method="appendLog" xrpc:arity="1" xrpc:location="http://x.example.org/film.qx" xrpc:updCall="false">
  <xrpc:queryID xrpc:host="x.example.org" xrpc:timestamp="32414232" xrpc:timeout="180"/>
  <xrpc:call xrpc:tag="1.1.1"> <!-- first call -->
    <xrpc:sequence>
      <xrpc:atomic-value xsi:type="xs:string">Julie Connery</xrpc:atomic-value>
    </xrpc:sequence>
  </xrpc:call>
  <xrpc:call xrpc:tag="1.2.1"> <!-- second call -->
    <xrpc:sequence>
      <xrpc:atomic-value xsi:type="xs:string">Sean Connery</xrpc:atomic-value>
    </xrpc:sequence>
  </xrpc:call>
</xrpc:request>
```

The second function application leads to a similar XRPC request (logging actors with surname Andrews this time), with call tags 1.1.2 and 1.2.2 (not shown). Figure 4.4 shows the pending update list \( \Delta_q \) at peer \( p_1 \) ("y.example.org") including the extra column \( T \). It depicts the situation at commit time. Both both XRPC requests have been executed successfully, and produced pending update sublists \( fid_1 @ p_1 \) and \( fid_2 @ p_1 \), which were concatenated in \( \Delta_q @ p_1 \) as both executed with isolation semantics \( F'_u \) (note that the XRPC Request above includes the `queryID` element). Sorting the pending update list on \( T \) achieves the desired deterministic update order.

### 4.5 Distributed XRPC Transactions

XRPC allows XQUF [58] expressions to be executed on remote peers, by means of XRPC calls to updating functions, thus providing distributed transaction functionality. For such distributed updating queries, XRPC provides two different isolation levels, *no isolation* and *repeatable reads*, to meet the needs of different kinds of applications. The latter level provides *repeatable reads* for all XRPC requests to the same peer made in a single query and uses a distributed 2-Phase Commit (2PC) protocol to ensure atomic commit. The semantics of these levels have already been formally defined in Section 3.4.2, including necessary extensions to the basic SOAP XRPC protocol to support them. Each XRPC query can specify the desired isolation level using the XQuery `declare option` feature to set `xrpc:isolation` to `none` or `repeatable`. Here, we briefly explain how repeatable reads with atomic commit are supported for updating XRPC queries on different XQuery engines.

To ensure *repeatable reads*, during the execution of an updating XRPC query \( q_{up} \), each participating peer maintains the same database state (i.e., all persistently stored XML documents) for the query. This can be done using systems that either use (lock-based) serialisation,
snapshot isolation, or multi-version concurrency control. The XRPC update requests generated by a query are not applied immediately to the database state used by that query while it runs. Rather, these requests are collected, in correspondence with the XQUF formal definition of a pending update list, that grows while the query runs. When the update query decides to commit, all peers in the transaction effectuate all updates in this list.

To provide atomic distributed commit, we have chosen to use the SOAP-based 2PC industry standard WS-AtomicTransaction[55], which defines an API with functions such as Prepare() and Commit(). It is embedded in the WS-Coordinator framework [54] that allows registering a collection of peers that participate in a distributed transaction, and subsequently run a transaction protocol (in this case WS-AtomicTransaction) on those peers. In XRPC, peer $p_q$ that starts the query $q$ is the one that registers the participating peers at the WS Coordinator service and initiates the Prepare and Commit phases. For this registration task, it thus needs to know a full list of peers that participated in the transaction. Due to nested XRPC calls (i.e., remote functions calling in turn other remote functions), the query originator may not be aware of all peers involved and therefore we extended the SOAP XRPC protocol to piggyback a list of unique participating peers in their response messages.

To provide updating XRPC queries with repeatable reads and atomic commit, XRPC systems must implement these web service 2PC interfaces and offer them over the same HTTP SOAP server that runs XRPC (this is the case in MonetDB/XQuery).

### 4.5.1 Heterogeneous Distributed 2PC

To enable heterogeneous distributed transactions, that is, performing XQUF updates on multiple peers that run different XQuery engines, the WS-AtomicTransaction 2PC interfaces are implemented in our XRPC Wrapper. This involves extending the XRPC Wrapper with concurrency control, XRPC message logging, and recovery functionality, that is used on top of the transactional capabilities of the underlying XQuery engine.

To provide the repeatable reads isolation level, the underlying XQuery engine must provide repeatable read consistency or better, and support multi-query transactions (with explicit start-transaction and commit/abort commands). For read-only queries under repeatable reads,
the XRPC Wrapper keeps a separate client connection open to the XQuery engine in which all XQuery requests with the same query ID are executed. This connection is kept open for the timeout period as specified in the XRPC requests. The XRPC Wrapper also keeps a log of recently expired query IDs (and an in-memory hash-table for fast lookups) such that it can properly generate error message for late requests. Note that query IDs contain a global timestamp, on which a reasonable maximum timeout can be enforced, so the size of the hash table should remain limited.

Updating queries can generate XRPC requests to both normal (read-only) XQuery functions as well as updating functions as defined by the XQUF, and are processed as follows.

1. When an XRPC request is received:
   a) check the query ID $id$ carried by the request to see if a connection $C_{id}$ for this query has already been created, and if not, create a new one, starting a new transaction (as mentioned, an error is generated for expired IDs). Also, a new subdirectory $D_{id}$ is created in the logging directory of the XRPC Wrapper;
   b) if the called function is a read-only function, execute it using the underlying XQuery engine and send its result back to the caller\(^1\). If the execution fails, add the query ID to the expired query log and remove $D_{id}$;
   c) otherwise, save the XRPC request message to the logging subdirectory $D_{id}$ and send a response message to the caller to indicate success without actually executing the updating function (this is possible, as updating XQuery functions do not return a result). The rationale is that in order to provide repeatable reads, we must execute all updates together, at the end of the transaction; otherwise their effects would be visible for subsequent requests belonging to the same transaction.

2. When a Prepare request with ID $id$ is received, then:
   a) if ID is expired, send Aborted to the coordinator;
   b) otherwise, if there are no request messages saved in the logging directory $D_{id}$, send ReadOnly to the coordinator, and then remove the logging directory $D_{id}$\(^2\);
   c) otherwise, construct a single query containing all updating requests that have been saved so far (by using XQuery sequence construction). Execute the query in connection $C_{id}$, without committing the transaction yet. If this update query fails, add the query ID to the expired query log and remove $D_{id}$. Finally, send the decision Committed or Aborted to the coordinator (depending on the update success).

3. When a Rollback or a Commit request with ID $id$ is received:
   a) if the request is Commit, log a “committing message” to $D_{id}$, and commit the transaction in $C_{id}$; The XRPC Wrapper should cease operation if committing in $C_{id}$ fails, and then try to restart the underlying XQuery engine and/or itself, entering recovery mode;
   b) add the query ID to the expired query log and remove $D_{id}$.

\(^1\)Note that, to reduce possible communication time with the coordinators needed by the recover procedure, each message should be logged before it is sent.

\(^2\)Upon receipt of a ReadOnly notification, the coordinator knows that the participant votes to commit the transaction and has forgotten the transaction.
Thus, the XRPC Wrapper plays the game of declaring a distributed transaction committed, before actually committing in the underlying XQuery engine, relying on its own logging to do so at the global commit point.

Recovery is done every time the XRPC Wrapper starts, before it accepting any new XRPC requests. During recovery, the logging directory is scanned for unfinished transactions, i.e., for subdirectories containing messages of unfinished transactions. For each subdirectory, if no final decision can be deduced from the logs (message logs and expired query ID log), it is requested from the coordinator. Transactions that should be committed are then re-executed (Step 3).

The worst possible case is finding a “committing” message. As it may happen that the underlying XQuery engine committed but the XRPC Wrapper crashed before removing the $D_{id}$ directory, re-trying the commit runs the risk of executing its updates twice. This risk can be mitigated by inspecting the log of the underlying XQuery engine (if accessible).

4.6 MonetDB/XQuery*

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 $\text{fn:doc()}$ and $\text{fn:put()}$ built-in functions plus our XRPC $\text{execute at}$ language construct.

4.6.1 Simple Scenarios

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 $\text{execute at}$ construct, when combined with $\text{fn:doc()}$ and $\text{fn:put()}$, provides an implementation of HTTP-based data shipping, as shown by the following rewriting rules:

\[
\text{StatEnv.baseURI } \Leftarrow \emptyset
\]
\[
\text{execute at } \{"xrpc://host"\} fn:put($\text{node}$, "\text{name}\}) \quad (R_{\text{put}})
\]
\[
\text{fn:put($\text{node}$, "xrpc://host/localname")}
\]
\[
\text{StatEnv.baseURI } \Leftarrow \emptyset
\]
\[
\text{execute at } \{"xrpc://host"\} fn:doc("localname") \quad (R_{\text{doc}})
\]
\[
\text{fn:doc($\text{node}$, "xrpc://host/localname")}
\]

Thus, an XQuery system with XRPC can implement the HTTP protocol in $\text{fn:doc()}$, $\text{fn:put()}$ 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 [67]).

4.6.2 Loose DHT Coupling

A Distributed Hash Table [147, 2] 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 [42, 43, 100, 101, 108, 142]. 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 \( \text{fn:doc()} \), \( \text{fn:put()} \) 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 DBMS to participate in multiple (logical) DHTs simultaneously, as shown in Figure 4.6(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 \( \text{put}(key,value) \) and \( \text{get}(key):value \) functions, to implement the XQuery data shipping functions \( \text{fn:put()} \) and \( \text{fn:doc()} \), as shown in rules \( \mathcal{R}_{\text{put}_2} \) and \( \mathcal{R}_{\text{doc}_2} \):

\[
\begin{array}{l}
\begin{align*}
p_1 &= \text{dht\_hash}_{\text{dht\_id}}(key) \\
\text{dht\_send}_{p_0\to p_i}\text{r}est\text{send}\text{r}equest("put", (key,\$node)) & \Rightarrow dht\_store_{p_i}\text{r}est\text{send}\text{r}equest() \\
\text{dht\_send}_{p_i\to p_0}\text{r}est\text{send}\text{r}equest() & \Rightarrow dht\_store_{p_i}\text{r}est\text{send}\text{r}equest() \\
\end{align*}
\end{array}
\] (\( \mathcal{R}_{\text{put}_2} \))

\[
\begin{array}{l}
\begin{align*}
p_1 &= \text{dht\_hash}_{\text{dht\_id}}(key) \\
\text{dht\_send}_{p_0\to p_i}\text{r}est\text{send}\text{r}equest("get", (key)) & \Rightarrow \text{dht\_store}_{p_i}\ \text{r}est\text{send}\text{r}equest() \\
\text{dht\_send}_{p_i\to p_0}\text{r}est\text{send}\text{r}equest(\$node) & \Rightarrow \text{dht\_store}_{p_i}\ \text{r}est\text{send}\text{r}equest(\$node) \\
\end{align*}
\end{array}
\] (\( \mathcal{R}_{\text{doc}_2} \))

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 \( \text{fn:doc}() \) 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 \( \text{dht\_id} \) (see Figure 4.6(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:

\[
\begin{align*}
\text{StatEnv\_baseURI} &\leftarrow \text{dht://dht\_id}/\text{prefix} \\
\text{db}@p_0 \vdash f_i(\text{ParamList}) \Rightarrow \text{val}, \text{db}@p_0 & \quad (\mathcal{R}_{\text{rpc}_2})
\end{align*}
\]

Rule \( \mathcal{R}_{\text{rpc}_2} \) 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 $R_{put2}$ and $R_{doc2}$ 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 $R_{xrpc2}$ 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.

4.6.3 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 4.6(b)). To realise this, we need to extend the DHT API with a single new method:

\[
\text{xrpc}(\text{key}, \text{q}, \text{m}, f_r(\text{ParamList})) : \text{item()}^* \tag{R_{xrpc3}}
\]

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:

\[
\begin{align*}
  p_i &= \text{dht_hash}_{dht}_id(\text{key}) \\
  \text{dht_send}_{p_0 \rightarrow p_i}(\text{q}, \text{m}, f_r, \text{ParamList}) \\
  \text{db}@q_{p_i} &\vdash f_r(\text{ParamList})@p_i \Rightarrow \text{val}, \text{db}@p_i \\
  \text{dht_send}_{p_i \rightarrow p_0}(\text{response}(\text{val})) \\
  \text{db}@p_0 &\vdash f_r(\text{ParamList})@\text{dht://dht_id/key} \Rightarrow \text{val}, \text{db}@p_0
\end{align*}
\]

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 $R_f$ or $R'_f$). 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 $R_{xrpc3}$ with $R_{doc1}$ and $R_{put1}$. 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 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.

4.7 Conclusion

In this chapter, we have discussed various aspects of using XRPC in distributed XQuery processing. First we show that XRPC can be easily adopted by different XQuery engines, such that complex P2P communication patterns can be programmed using XRPC. To enhance adoption of XRPC, we described a XRPC wrapper that allows any XQuery data source to handle
XRPC calls. During our Saxon experiments, we also saw that Bulk RPC enables set-oriented optimisations, such that Bulk RPC execution of a selection function can be handled using a join strategy.

Then, to better match the transaction semantics in databased, we define a deterministic update semantics for XQUF queries, and showed how the SOAP XRPC can be extended to guarantee deterministic order in distributed update scenarios. To provide atomic distributed commit, we have chosen to use the SOAP-based 2PC industry standard Web Services Atomic Transaction [55].

Finally, we discussed work on MonetDB/XQuery that aims to create powerful P2P XML database technology that preserves the full XQuery language (+XQUF), extending it only with a single new construct, i.e., XRPC. We 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. In Chapter 7, we will show how this functionality can be used in the StreetTiVo collaborative video indexing application. Our next step in this area is to implement these couplings in MonetDB/XQuery using the Bamboo DHT [147], and perform experiments in environments like PlanetLab. Especially the tight coupling will open up a playing field for a number of query optimisation techniques that exploit the P2P network characteristics.

\[^3\text{XRPC and the XRPC wrapper are available in the open-source XDBMS MonetDB/XQuery (http://monetdb.cwi.nl).}\]