Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

INS

Information Systems

INformation Systems

Loop-lifted XQuery RPC with deterministic updates

Y. Zhang, P.A. Boncz

REPORT INS-E0607 NOVEMBER 2006

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2006, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3681

Loop-lifted XQuery RPC with deterministic updates

ABSTRACT
XRPC is a minimal XQuery extension that enables distributed query execution, combining the Remote Procedure Call (RPC) paradigm with the existing concept of XQuery functions. By calling out of a for-loop to multiple destinations, and by calling functions that themselves perform XRPC calls, complex P2P communication patterns can be achieved. We further propose the use of SOAP as the protocol for XRPC, which allows seamless integration with web services and Service Oriented Architectures (SOA). XRPC is implemented in the open source MonetDB/XQuery system. We show that the technique of *loop-lifting*, that executes all expressions inside a for-loop in a single bulk operator -- pervasively applied in MonetDB/XQuery to obtain efficient relational query plans -- also benefits XRPC. Loop-lifting enables us to send *bulk* RPC requests, dramatically reducing the number of SOAP messages, and thus the performance impact of network latency. The XRPC extension is orthogonal to all XQuery language features, including the XQuery Update Facility (XUF). The XUF W3C Draft proposal does not define the order in which multiple update actions to the same node must be applied. We instead choose to make this order deterministic, and show how distributed updates can be made deterministic using a small protocol extension.

2000 Mathematics Subject Classification: 68M14 Distributed systems
1998 ACM Computing Classification System: [H.2.4]Query processing
Keywords and Phrases: Distributed Query Processing;XQuery Processing;RPC;SOAP
Note: This work was carried out under project "MultimediaN" - "AmbientDB"

Loop-Lifted XQuery RPC With Deterministic Updates

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

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

ABSTRACT

XRPC is a minimal XQuery extension that enables distributed query execution, combining the Remote Procedure Call (RPC) paradigm with the existing concept of XQuery functions. By calling out of a `for`-loop to multiple destinations, and by calling functions that themselves perform XRPC calls, complex P2P communication patterns can be achieved. We further propose the use of SOAP as the protocol for XRPC, which allows seamless integration with web services and Service Oriented Architectures (SOA).

XRPC is implemented in the open source MonetDB/XQuery system. We show that the technique of *loop-lifting*, that executes all expressions inside a `for`-loop in a single bulk operator – pervasively applied in MonetDB/XQuery to obtain efficient relational query plans – also benefits XRPC. Loop-lifting enables us to send *bulk* RPC requests, dramatically reducing the number of SOAP messages, and thus the performance impact of network latency.

The XRPC extension is orthogonal to all XQuery language features, including the XQuery Update Facility (XUF). The XUF W3C Draft proposal does not define the order in which multiple update actions to the same node must be applied. We instead choose to make this order deterministic, and show how distributed updates can be made deterministic using a small protocol extension.

1. INTRODUCTION

The contributions of this paper are three-fold: *(i)* the definition of XRPC, a small and clean language extension for XQuery that allows simple client-server RPC as well as complex P2P XQuery group communications and is based on the SOAP protocol, thus integrating seamlessly with (Web) Service Oriented Architectures, *(ii)* the idea to exploit the query translation technique of *loop-lifting* to optimize network communications, by performing many function calls in a single RPC (*Bulk RPC*). This brings the benefits of set-at-a-time database processing to the realm of RPC and by doing so dramatically reduces the impact of network latency on distributed query performance, *(iii)* the definition of *deterministic* semantics for the XQuery Update Facility [5] and a SOAP protocol extension that allows to guarantee deterministic semantics when calling user defined *updating functions* over XRPC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2006 ACM 1-59593-256-9/06/0006 ...$5.00.

XRPC Language Extension The current W3C XQuery language only provides a *data shipping* model for querying XML documents distributed on the Internet. The built-in function `fn:doc()` fetches an XML document from a remote peer to the local server, where it subsequently can be queried. The recently published W3C working draft of XQuery Update Facility (XUF) introduces the built-in function `fn:put()` for remote storage of an updated document, which again implies data shipping.

There have been various proposals to equip XQuery with *function shipping* style distributed querying abilities [19, 21, 22], and we consider our XRPC proposal an incremental development of these, with specific advantages concerning optimizability and simplicity. Considering simplicity, XRPC adds RPC to XQuery in the most direct way, adding a destination URI to the XQuery equivalent of a procedure call (i.e. function application). XRPC is optimizable in that it makes explicit the input data (parameters) of a remote query and its result type through the function signature, routinely identified during query parsing in existing XQuery systems. Also, functions can be defined in XQuery modules, and compiled separately in advance, making it easy to do query plan caching and thus accelerate distributed query processing.

On a lower level, our XRPC proposal also encompasses an – again natural – choice for SOAP (i.e. XML messages) over HTTP, as the underlying network protocol. XML is ideal for distributed environments (think of character encoding hassles, byte ordering), XQuery engines are perfectly equipped to process XML messages, and an XML-based message protocol makes it trivial to support passing values of any type from the XQuery Data Model [8].

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).

Loop-Lifted Optimizations We implemented XRPC in the open source MonetDB/XQuery system [3]. It consists of the *Pathfinder* compiler [12] that translates XQuery expressions into relational query plans, on top of the MonetDB relational execution engine. The essence of the compilation technique employed by Pathfinder is *loop-lifting*, which translates XPath/XQuery expression inside `for`-loops into single bulk relational query plans. Loop-lifting makes MonetDB/XQuery inherently different (and often faster) than those XQuery *interpreters* that tend to strictly follow the `for`-loop order syntactically suggested by a query. Bulk plans have the ability to change the order of execution when advantageous, yet we note that MonetDB/XQuery fully supports XQuery node and sequence order

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

in the final query result. Also, relational database engines are tuned to execute bulk query plans efficiently.

In case of XRPC, loop-lifting turned out to enable a SOAP protocol optimization to send requests for a sequence of (many) RPC calls in a *single* message exchange. This optimization dramatically reduces the quantity of request/response messages sent and thus the impact of network latency on query performance.

Deterministic Updates The current draft 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", then its value after the update

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

may be both `<a><b/><c/></a>` and `<a><c/><b/></a>`. In MonetDB/XQuery, we choose to implement the XUF 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><c/><b/></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.

1.1 Outline

This paper is organized as follows. In Section 2 we shortly give a definition of the XRPC language extension, including the SOAP sub-protocol it uses. Section 3 explains the concept of loop-lifting, and how this influenced the implementation and performance of XRPC in MonetDB/XQuery. In Section 4 we turn our attention to the interaction between XRPC and the XQuery Update Facility, and define a deterministic distributed update semantics. We describe an extension to our SOAP protocol and a proof that this protocol conforms to the deterministic update semantics. Finally, we discuss related work in Section 5 before outlining our conclusions and future work in Section 6.

2. THE XRPC LANGUAGE EXTENSION

We *(i)* describe the XRPC syntax extension and give examples, *(ii)* define the SOAP message format underlying XRPC, and *(iii)* define the formal semantics of XRPC calls.

2.1 XRPC syntax

The XRPC syntax for remote function application is:

$$\texttt{execute at}\ Expr\ \{\ FunApp\ (\ ParamList\ )\ \}$$

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 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 (passed by value). 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.

For a precise syntax definition, we show the rules of the XQuery 1.0 grammar that were changed:

```
PrimaryExpr ::= ...| FunctionCall | XRPCCall | ...
XRPCCall ::= "execute at" "{" ExprSing "}" "{" FunctionCall "}"
FunctionCall::= QName "(" (ExprSingle("," ExprSingle)*)?  ")"
```

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

```
<filmDB>
 <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>
</filmDB>
```

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

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

We can execute this function on 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>
{ 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:

$$\texttt{xrpc://}<host>[\,:port\,]\ \ [/[\,path\,]]$$

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>
```

More Examples. A more elaborate example demonstrates the possibility of multiple remote function calls to a peer:

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

and to make it a bit more complex, we could do multiple function calls to multiple remote peers:

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

Complex communication patterns may be programmed with XRPC, especially if recursive functions are used:

```
module namespace film="filmdb";
declare function
 film:recursiveActor($destinations as xs:string*,
                     $actor as xs:string) as node()
{
 let $cnt := fn:count($destinations)
 let $pos := ($cnt / 2) cast as xs:integer
 let $dsts1 := fn:subsequence($destinations, 1, $pos)
 let $dsts2 := fn:subsequence($destinations, $pos+1)
 let $peer1 := $destinations[1]
 let $peer2 := $destinations[$pos]
 return
  (if ($cnt > 1)
    then execute at {$peer1} {film:recursiveActor($dsts1, $actor)}
    else (),
   doc("filmDB.xml")//filmName[../actorName=$actor],
   if ($cnt > 2)
    then execute at {$peer2} {film:recursiveActor($dsts2, $actor)}
    else ())
};
```

The above executes the RPC on a set of destination peers, uniting all results, and does so by constructing an binary spanning tree of recursive RPC calls.

2.2 SOAP XRPC Message Format

SOAP (Simple Object Access Protocol) is the XML-based message format used for web services [17], and we propose the use of SOAP messages over HTTP as the network protocol underlying XRPC. 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. For the simple RPC use of SOAP over HTTP, a sub-protocol called "SOAP RPC" is in common use [14]. SOAP RPC is oriented towards binding with programming languages such as C++ and Java, and specifies parameter marshalling of a certain number of simple (atomic) data types, and also allows passing *arrays* and *structs* of such data-types. However, its supported atomic data types do not match directly those of the XQuery Data Model (XDM) [8], and the support for arrays and structs is not relevant in XRPC, where there rather is a need for supporting arbitrary-shaped XML nodes as parameters as well as sequences of heterogeneously typed items. This is the reason, why our SOAP XRPC message format, while supporting the general SOAP standard over HTTP with the purpose of RPC, implements a new parameter passing sub-format (SOAP XRPC $\neq$ SOAP RPC). The most often used form of SOAP RPC is called *rpc/encoded*, while our SOAP XRPC protocol belongs to the family of *document/literal*. It was shown in [6] that rpc/encoded in general is significantly slower than document/literal, and suffers from scalability problems when the message size increases.

XRPC Request Message. SOAP messages consist of an envelope, with a (possibly empty) 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 first example query, that looks for films with Sean Connery:

```
<?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>
```

Atomic values are represented with `xrpc:atomic-value`, and are annotated with their (simple) XML Schema Type in the `xsi:type` attribute. Thus, the heterogeneously typed sequence consisting on 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 `xrpc:element` element:[2]

```
<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`[3] defines enclosing elements for document, attribute, text, processing instruction, and comment nodes. Document nodes are represented in the SOAP message as a `xrpc:document` element that contains the serialized document root. Text, comment and processing instruction nodes are serialized textually inside the respective elements `xrpc:text`, `xrpc:comment` and `xrpc:processing-instruction`. Attribute nodes are serialized *inside* the `xrpc:attribute` element:

```
<xrpc:attribute x="y"/>
```

XRPC fully supports the XQuery Data Model, a requirement for making it an orthogonal language feature. This implies XRPC also supports passing of values of user-defined XML Schema types, including the ability to validate SOAP messages. XQuery already allows importing XML Schema files that contain such definitions. Values of user-defined types are enclosed in SOAP messages by `xrpc:element` elements, with a `xsi:type` attribute annotating their type. The XQuery system implementing XRPC should include a `xmlns` namespace definition as well as a `xsi:schemaLocation`

[2]Note that white space *inside* the `element` node does matter.

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

declaration inside the Envelope element when values of such imported element types occur in the SOAP message.

XRPC Response Messages follow the same principles, e.g.:

```
<?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:response module="filmdb" method="filmsByActor">
   <xrpc:sequence>
    <xrpc:element><filmName>The Rock</filmName></xrpc:element>
    <xrpc:element><filmName>Goldfinger</filmName></xrpc:element>
   </xrpc:sequence>
  </xrpc:response>
 </env:Body>
</env:Envelope>
```

Inside the body is now a xrpc:response element that contains the result sequence of the remote function call.

XRPC Error Message. Whenever an XRPC server discovers an error during the processing of an XRPC request, it immediately stops execution and sends back an XRPC error message, using the format of the SOAP Fault message ([17], [13]). For example, the following SOAP Fault message indicates that a required module could not be loaded (we show only the env:Fault element):

```
<env:Fault>
  <env:Code><env:Value>env:Sender</env:Value></env:Code>
  <env:Reason>
    <env:Text xml:lang="en">could not load module!</env:Text>
  </env:Reason>
</env:Fault>
```

Outlook. Our discussion of SOAP XRPC message is not fully done yet. In the next subsection, we will extend the format with support for isolation. Then in Section 3.2 we describe the *Bulk RPC* feature, that allows a single message to request multiple function calls. Finally, in Section 4.1 we describe a small extension to include tag attributes in call elements that allows to keep track of a deterministic distributed update order.

2.3 XRPC Formal Semantics

In defining the semantics of XRPC, 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 also define certain less strict isolation levels that still may be useful to certain applications.
We use the following notation and terms:

– $\mathcal{P}$ denotes a set of *peer identifiers*. We use the peer identifier p_0 to denote the *local peer*, on which a particular query is started. All other peers $p_i \in \mathcal{P}$ are *remote peers*. In practice, a peer identifier is a URI from the http protocol, that identifies a host and (optionally) a port number.

– $\mathcal{F}$ denotes a set of *XRPC function applications*. An XRPC call f is an *updating XRPC call* ($f \in \mathcal{F}_u$), if it calls an updating function; otherwise, it is a *non-updating XRPC call* ($f \in \mathcal{F}_r$). Thus $\mathcal{F} \equiv \mathcal{F}_u \cup \mathcal{F}_r$. 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 $m_f \in \mathcal{M}$.

– An *XRPC query* is an XQuery query q which contains at least one XRPC call $f \in \mathcal{F}_q$, where $\mathcal{F}_q$ denotes the set of all function calls performed during execution of q. We call a query in which only one, non-nested XRPC call appears a *simple XRPC query*. An XRPC query q is an *updating XRPC query*, if it contains at least one update command or a call to an updating (XRPC) function.

– Each query operates in a *dynamic context*. The XQuery 1.0 Formal Semantics [7] defines that each expression is normalized to a *core* expression, which then is defined by a semantic judgement $dynEnv \vdash Expr \Rightarrow val$ The semantic judgement specifies that in the dynamic context *dynEnv*, the expression *Expr* evaluates to the value *val*, where *val* is an instance of the XQuery Data Model (XDM). 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 [7] corresponds to *db* used here. To indicate a context at a particular peer p, we write $db@p$.

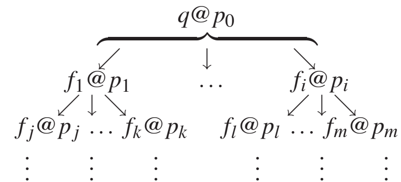
Basic read-only XRPC is defined by extending the XQuery 1.0 semantic judgements with a new rule $\mathcal{R}_{\mathcal{F}_r}$:

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

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 $(m, f_r, ParamList)$ 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 the 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 exists 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 XRPCs if these happen to be present in body of f.

Repeatable Reads. The general pattern of XRPC function applications generated by a query is a *tree*, as each XRPC call may again perform more XRPC calls. This happens when a query contains multiple XRPC function applications, or when such a function application occurs inside a for-loop. In the below diagram, the arrow '→' should be read as "XRPC call":



The peers $p_0, p_1, \cdots, p_i, p_j, \cdots, p_k, \cdots, p_l, \cdots, p_m$ are not necessarily unique: some peer p_i (or in fact many such peers) may occur multiple times in this tree. When considering rule $\mathcal{R}_{\mathcal{F}_r}$, the dynamic environment $dynEnv@p_i$ containing the *current* database

Operator	Semantics
σ_a	select all rows with column $a = true$
$\pi_{a_1:b_1,\ldots,a_n:b_n}$	project columns $b_1,\ldots,b_n$ and possibly rename columns b_i to a_i (no duplicate removal)
δ	duplicate elimination
$\dot{\cup}$	disjoint union
$\bowtie_{a=b}$	equi-join
$sort_{a_1,\ldots,a_n}$	sorting
$\rho_{b:\langle a_1,\ldots,a_n\rangle/p}$	row numbering (DENSE_RANK SQL:1999)
ab	literal table

Table 1: Relational Algebra Generated By Pathfinder

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 states with a query identifier: $db_q@p_i$.

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

This specifies the database state at peer p_i to remain in the same state $db_q@p_i$ during all XRPCs belong to the same query q. 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. 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 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).

SOAP XRPC Extension: Isolation. Repeatable reads surface in our protocol by the introduction of an optional `queryID` child element to the `xrpc:request` element. It contains `host` and `timestamp` attributes that state on which host and at what point in time the query started initially, and a `timeout` attribute that specifies a local number of seconds during which to conserve the isolated database state. Note that the timeout is relative, it is a number of seconds – this mitigates problems caused by different peers having wrong system clock settings or using different time zones. When the timeout passes, the isolated database state can be discarded, freeing up system resources. However, the local XRPC handler should still remember expired `queryID`s, such that it can give errors on XRPC requests that arrive too late. The purpose of sending the `timestamp` of the originating host is to ease the administration of expired `queryID`s, as per host only the latest timestamp needs to be retained, and can be restricted to some sane time interval.

Updating XRPC calls. The XRPC language extension is fully orthogonal to all XQuery features, and thus one can also make XRPC calls to user-defined *updating functions*, as defined by the XQuery Update Facility (XUF). The XUF syntax ensures that if a user-defined function contains one updating function, it must itself be an updating function. XQuery updates (and thus updating functions) determine which nodes to change (and how), purely based on the database state before the update, and produce a *pending update list* Δ. While in reality updating queries and updating functions always return the empty sequence and the creation of Δ happens on-the-side, we let them return this Δ directly (to simplify our rules).

$$\frac{\begin{array}{c} send_{p_0\rightarrow p_i}request(m,f_w,ParamList) \\ db@p_i \vdash f_w(ParamList) \Rightarrow \Delta_f, db@p_i \\ db@p_i \vdash \texttt{commit}(db@p_i,\Delta_f) \Rightarrow db'@p_i \\ send_{p_i\rightarrow p_0}reply() \end{array}}{db@p_0 \vdash f_w(ParamList)@p_i \Rightarrow (), db@p_0} \quad (\mathcal{R}_{\mathcal{F}_u})$$

Above rule $\mathcal{R}_{\mathcal{F}_u}$ states that update functions commit the remote database state $db@p_i$ immediately. While easy to implement, this rule allows lost updates and non-atomic distributed commits to happen and does not guarantee repeatable reads.

Updates with Isolation. Peers handling multiple XRPC calls for a query q must not only keep track of the database state db_q, but also of the pending update list Δ_q. Thus, the dynamic context with updates can be represented by a tuple: $dynEnv_u \simeq (db_q, \Delta_q)$. When a peer starts to participate in query q, its pending update list $\Delta_q = \emptyset$ (i.e. empty). Each updating XRPC call received by peer p_i for query q adds update actions to the list of the remote peer:

$$\frac{\begin{array}{c} send_{p_0\rightarrow p_i}request(q,m,f_w,ParamList) \\ (db_q@p_i,\Delta_q@p_i) \vdash f_w(ParamList) \Rightarrow \Delta_{f_w},\ db_q@p_i \\ db_q@p_i \vdash \texttt{mergeUpdates}(\Delta_q@p_i,\Delta_{f_w}) \Rightarrow \Delta'_q@p_i,\ db_q@p_i \\ send_{p_i\rightarrow p_0}reply() \end{array}}{(db_q@p_0,\Delta_q@p_0) \vdash f_w(ParamList)@p_i \Rightarrow (),\ db_q@p_0} \quad (\mathcal{R'}_{\mathcal{F}_u})$$

The translation of isolated updating XRPC calls is depicted in the inference rule $\mathcal{R'}_{\mathcal{F}_u}$ above. The rule states that at p_i, the update actions that are caused locally by f are merged (with the XUF function `upd:mergeUpdates`) with the pending update list $\Delta_q@p_i$ that p_i keeps for query q. Like rule $\mathcal{R}_{\mathcal{F'}_r}$, this rule again provides for proper isolation by keeping the database state $db_q@p_i$ constant throughout the query. Note that the XQuery system needs to implement a distributed commit protocol in order to achieve atomic commit. Any distributed commit algorithm developed for relational databases (such as 2-phase commit) can be used for this ([20], [11]).

3. LOOP-LIFTED XRPC

We implemented XRPC in MonetDB/XQuery, an efficient yet purely relational XML database system [3]. It consists of the MonetDB relational database back-end, and the *Pathfinder* compiler [12], that translates XQuery into relational algebra as front-end.

The XRPC module contains an ultra-light HTTP daemon implementation [16] 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.

The stub code uses the message sender API to generate a SOAP message from actual function parameters. This process reuses the normal sequence serialization mechanism in MonetDB/XQuery. The

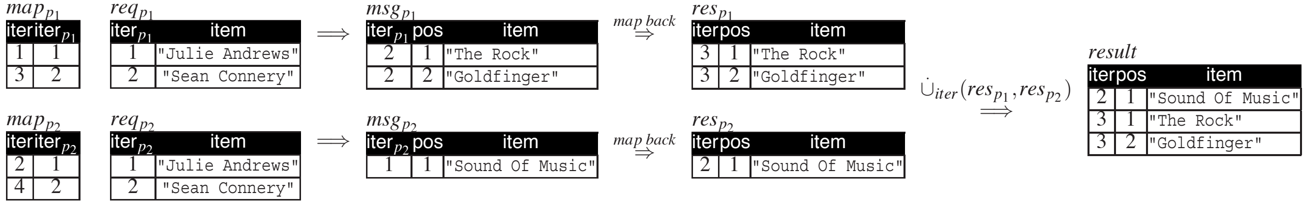


Figure 1: Relational Processing of Bulk RPC (Multiple Destinations Example)

message sender API sends the XML message using HTTP POST and waits for a result message. The result message is subsequently shredded into a relational table, the way all XML documents are shredded in MonetDB/XQuery. The stub code retrieves atomic values from the SOAP document nodes; node-typed values just refer to the nodes in the newly shredded SOAP document.

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. As MonetDB/XQuery is a relational system, XQuery values are all represented as (temporary) relational tables. 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, using the normal MonetDB/XQuery serialization mechanism onto the network socket.

Thus, as we re-used the shredding and serialization functionality already in MonetDB/XQuery, as well as an off-the-shelve open source HTTP daemon [16], implementation was limited to a small parser extension, and stub code generation.

3.1 Relational XQuery And Loop-Lifting

The *Pathfinder* compiler [12] translates XPath/XQuery expressions into bulk query plans formulated in the vanilla relational algebra, depicted in Table 1. All operators are well-known, except perhaps the row numbering operator ρ, which is similar to the SQL:1999 operator DENSE_RANK: $\rho_{b:\langle a_1,\ldots,a_n\rangle/p}(q)$ assigns each tuple in q a rank (i.e. number), which is saved in column b. The constraint for the enumeration is the implicit order or q by the columns $a_1,\ldots,a_n$. Numbers consecutively ascend from 1 in each partition defined by the optional grouping column p.

Representing sequences as tables. The evaluation of any XQuery expression yields an *ordered sequence* of $n \geq 0$ items x_i, denoted $(x_1,x_2,\ldots,x_n)$. MonetDB/XQuery is a relational system, thus sequences are represented as tables, with schema pos|item. Since relations have (unordered) set-semantics, sequence order must be explicitly maintained using a pos column. In the XQuery data model, a single item x and the singleton sequence (x) are identical. Item x is represented as a single row table containing the tuple $\langle 1,x\rangle$. The empty sequence () maps into the empty table.

pos	item
1	x_1
2	x_2
⋮	⋮
n	x_n

Loop-lifting. Each XQuery is translated bottom-up into a single relational algebra plan consisting only of the classical relational operations (select, project, join, etc); that is, the XQuery concept of nested for-loops is fully removed and a single bulk (=efficient and optimizable) execution plan is created.

The result of an XQuery at each step of bottom-up compilation is a relational plan that yields the result sequence *for each* nested iteration, all stored together. To make this possible, these intermediate tables have three columns:

```
s0 { for $x in (10,20)
     s1 { return for $y in (100,200)
          s2 { let $z := ($x,$y)
               return $z                      Q1
```

iter|pos|item, where iter is a logical iteration number, as shown in the tables below. For each scope, we keep a *loop* relation that holds all iters.[4] If we focus on the execution state in the innermost iteration body (marked as scope s_2) of $\mathcal{Q}_1$, there will be three such tables that represent the live variables $x, $y and $z respectively. As we can see from the iter columns, there are four iterations in scope s_2 (numbered from 1 to 4) and as expected, $x takes the value 10 in the first two iterations and the value 20 in the second two iterations. Similarly, $y takes the value 100 in the odd iterations and the value 200 in the even ones. Finally, $z is a sequence of two values in all four iterations (consisting of the value of $x followed by the value of $y).

loop

iter
1
2
3
4

x

iter	pos	item
1	1	10
2	1	10
3	1	20
4	1	20

y

iter	pos	item
1	1	100
2	1	200
3	1	100
4	1	200

z

iter	pos	item
1	1	10
1	2	100
2	1	10
2	2	200
3	1	20
3	2	100
4	1	20
4	2	200

3.2 Bulk RPC

Our earlier example query:

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

contains a function application inside a for-loop. Inside this loop, the variables $dst and $actor yield relational tables shown left. Thus, the value of $dst is 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.

actor

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

dst

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

XRPC SOAP Extension: Bulk RPC. The loop-lifted processing model of MonetDB/XQuery thus collects in a single table all XRPC function parameters needed by a remote function call, nested in one or more for-loops. This is exploited in XRPC SOAP by allowing *Bulk RPC*, in which a single XRPC message to the destination peer requests to perform *multiple* function calls. Each call is represented by an individual xrpc:call child element of the xrpc:request. Such a Bulk RPC also returns multiple results in the xrpc:response (one xrpc:sequence sequence for each call). From the shredded XRPC response message, it is quite straightforward to obtain the iter|pos|item table that represents an XDM result value for each iteration. Note that Bulk RPC exactly fits in the existing loop-lifted processing model of MonetDB/XQuery: without execute at, the local function translation mechanism already produced such a iter|pos|item table.

We show the xrpc:request part of the SOAP message in our Bulk RPC example, which contains two calls:

[4]The *loop* relation allows to keep track of empty sequence values, encoded by the absence of tuples in the expression representation.

$$\frac{\begin{array}{rl} \boxed{\text{iter|pos|item}}\ result & \Leftarrow \dot{\cup}_{iter}(res_p) \\ \forall p \in \delta(dst.item), \forall 1 \leq i \leq n: & \\ \boxed{\text{iter|iter}_p}\ map_p & = \pi_{iter,iter_p}(\rho_{iter_p}(\sigma_{item=p}(dst))) \\ \boxed{\text{iter}_p\text{|pos|item}}\ req^i_p & = \pi_{iter_p,pos,item}(\rho_{pos}(\bowtie_{iter=iter}(map_p, param_i))) \\ \boxed{\text{iter}_p\text{|pos|item}}\ msg_p & = f(req^1_p, \cdots, req^n_p)@p \\ \boxed{\text{iter|pos|item}}\ res_p & = \pi_{iter,pos,item}(\bowtie_{iter_p=iter_p}(msg_p, map_p)) \end{array}}{\begin{array}{c} \texttt{execute at}\ \boxed{\text{iter|pos|item}}\ \{ dst \} \\ \{ f(\boxed{\text{iter|pos|item}}\ param_1, \cdots, \boxed{\text{iter|pos|item}}\ param_n) \} \Rightarrow \boxed{\text{iter|pos|item}}\ result \end{array}}$$

Figure 2: Relational Translation of XRPC

```
<xrpc:request module="filmdb" uri="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>
```

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.

Let us now consider our other previous example:

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

actor

iter	pos	item
1	1	"JulieAndrews"
2	1	"JulieAndrews"
3	1	"SeanConnery"
4	1	"SeanConnery"

dst

iter	pos	item
1	1	"http://y.org/"
2	1	"http://z.org/"
3	1	"http://y.org/"
4	1	"http://z.org/"

We now have an inner `for`-loop with four iterations, but `$dst` takes on two *different* values, identifying peers `y.org` and `z.org`, in respective the odd and even iterations. The general rule to translate a loop-lifted XRPC call is shown in Figure 2 and Figure 1 shows the intermediate steps taken. The system establishes a list of unique peers, and for each p extracts from each parameter `iter|pos|item` those iteration (tuples) that invoke the function on p. The resulting request tables (req_p) are used to generate a Bulk RPC to p. Observe that using ρ a new $iter_p$ column is created, and a mapping table (map_p) that maps old to new iteration numbers. The mapping table is then again used to map the new iteration numbers back into old ones, and all result tables (res_p) are united with a (merge-)union on the `iter` column, to guarantee the correct order of the result.

Parallel & Out-Of-Order. The XRPC execution in Figure 1 performs two Bulk RPC calls. The first call processes both values of `$actor` on `y.org`, and then a second call performs the same task on `z.org`. It is important to observe that this order of processing is different than what is suggested by the query (i.e. first Julie Andrews on both, then Sean Connery on both). If a loop-lifted XRPC function application has multiple destination peers, in fact MonetDB/XQuery improves performance by dispatching all Bulk RPC requests *in parallel*, which makes the exact order in which peers execute the query unpredictable.[5]

[5] Note that after all parallel results are collected, the merge-union on `iter` guarantees that the result is in the correct order.

The out-of-order processing effects of loop-lifting are thus more easily explained by a single-destination (hence non-parallel) query:

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

Here, only the peer `y.org` is involved twice within the same query due to sequence construction. In the first Bulk RPC call, it will look for films by two actors with surname Connery, resp. surname Andrews in the second RPC. The intuitive order suggested by the query would be to look for actors by the name Julie first, and those named Sean second. This change of order in a peer that is visited multiple times has consequences for the semantics of updating functions called over XRPC (see Section 4).

The above is also a good example of a query that needs *isolation*, because it handles two RPC requests inside the same query. While in this particular case, those two requests could potentially be combined, this is much harder if two different functions would be executed, or downright impossible if the parameters of one depend on the outcome of the other. Certain classes of queries, such as those that contain only a single non-nested XRPC call, can be easily identified at compile time to send at most one XRPC request to each destination peer. For such queries, we can use the cheaper XRPC mechanism without `queryID` (see Section 2.3), while still guaranteeing repeatable reads.

Note that without Bulk RPC, the costly isolation mechanism would be required for any XRPC that performs more than a single XRPC call. Thanks to Bulk RPC, many queries have to send just a single message to each peer, thus not only reducing the number of network I/Os, but also lessening the overhead of isolation.

3.3 Experimental Evaluation

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)
 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. The left half of Table 2 shows the experiment where we compare performance of Bulk RPC with single RPC at-a-time, while varying 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 more synchronous RPCs.

Function Cache. XQuery Modules have the advantage that they may be pre-loaded and cached, and our choice to let XRPC use modules as the query transport mechanism also opens the possibility to reap performance profit from module pre-processing.

	Normal		Function Cache	
	`$x=1`	`$x=1000`	`$x=1`	`$x=1000`
one-at-a-time	163	68542	35	34979
bulk	172	534	35	400

Table 2: XRPC Performance On Addition Test (msec)

The feature of *prepared queries* is well-known for a RDBMS, allowing a parametrized query plan to be parsed and optimized off-line, such that an application can quickly enter actual parameters in the prepared plan and execute it. The ODBC and JDBC APIs export this functionality of relational databases using a programming language binding. MonetDB/XQuery has a mechanism for supporting prepared queries that does not need specific API support. Exploiting the fact that a prepared query is in essence a function with parameters, MonetDB/XQuery *caches* all query plans for (loop-lifted) function calls, for functions defined in XQuery Modules. Queries that just load a module and call a function in it with constant values as parameter, are detected by a pre-parser. The pre-parser then extracts the function parameters, and feeds them into a cached query plan. In MonetDB/XQuery, queries on small data sets can be accelerated ten-fold by this mechanism [3].

This same function cache mechanism is used by the SOAP XRPC request handler to handle *all* XRPC requests. This means that in MonetDB/XQuery a SOAP XRPC request usually does not need query parsing and optimization, just execution. The right half of Table 2 shows the impact of enabling the function cache: we see the processing time go down, improving both the single- and many-iteration Bulk RPC experiments. Thanks to the function cache, we can achieve a minimum RPC latency of 35 msec – which is not stellar when compared with environments like .NET ([10, 18]), but still respectable for an XML database system.

4. DETERMINISTIC XRPC UPDATES

The W3C working draft of the XUF [5] 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 working draft specifies that this ordering is *implementation-dependent*.

Definition Of Deterministic Updates. The motivation in MonetDB/XQuery to exercise our liberty to implement the XUF deterministically, is simply that order matters in XML. The solution chosen is that if the XUF working draft leaves the ordering of updates actions undetermined, we respect the order in the pending update list. The XUF 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 update must be merged with the `upd:mergeUpdates()` internal function. The XUF 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 $\mathcal{F}_u$ and $\mathcal{F}'_u$ further lead to synchronous function all semantics when updating functions are called over XRPC.

The Challenge. Now that the MonetDB/XQuery implementation of XUF cares about the update order, our challenge is to extend this care to distributed updates. In the end of Section 3.2, 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 XUF 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:

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

Our deterministic XUF 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 to re-order the pending update list at commit time such that the correct update order is followed.

4.1 Order-Correct Update Tags

We start by characterizing 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 $\langle p,a\rangle$ combinations in the required order. Then, we define an additional third $\mathcal{T}$ column for the DPUT that holds a *tag*, and explain how these tag values are constructed. We show that this $\mathcal{T}$ 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 $\mathcal{T}$ 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 [5]:

– insert expressions $\in$ {`upd:insertBefore`, `upd:insertAfter`, `upd:insertIntoAsFirst`, `upd:insertIntoAsLast`, `upd:insertInto`, `upd:insertAttributes`}.
– delete expressions $\in$ {`upd:delete`}.
– rename expressions $\in$ {`upd:rename`}.
– replace expressions $\in$ {`upd:replace-Node`, `upd:replaceValue`, `upd:replaceElementContent`}.

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

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 Δ an additional peer column $\mathcal{P}$. The resulting table $\mathcal{P}\mathcal{A}$ we call the Distributed Pending Update Table (DPUT). We should stress that this is a conceptual table only, we do not propose to materialize such a table in any way.

In Section 2.3 we described that when an updating XRPC query is started with isolation (i.e. following the semantics defined by $\mathcal{F}'_u$), each peer p keeps an isolated environment $\langle db_q@p, \Delta_q@p\rangle$ around. Each XRPC function application $f_i(Params)@p$ causes a sub-list $fid_i@p$ of pending update actions (just denoted Δ_f in rules $\mathcal{F}_u$ and $\mathcal{F}'_u$) that is merged into the overall list $\Delta_q@p$. We stress that $fid_i@p$ is just a conceptual list (not an implementation data structure) that represent the update actions caused at peer p by a single function call. The local list of pending updates at query site p_0 is denoted $qid@p_0$ here.

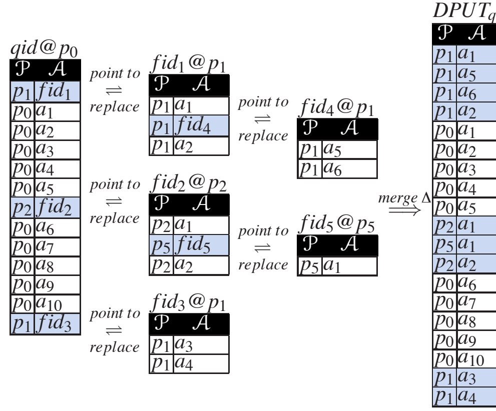


Figure 3: The conceptual Distributed Pending Update Table

COROLLARY 1. *Iteratively substituting each* $\langle p_x, fid_y \rangle$ *in* $qid@p_0$ *by sub-list* $fid_y@p_x$*, yields the DPUT in required order.*

Figure 3 shows $qid@p_0$ and all $fid_i@p_j$ caused by a single query, and the DPUT derived from those (the right-most table). In the lists, values a_i indicate single update actions, while the values fid_i points to another pending update sub-list, that represents all update actions caused by the called function fid_i at the peer in column $\mathcal{P}$. The iterative substitution of the sub-lists 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 XUF 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:

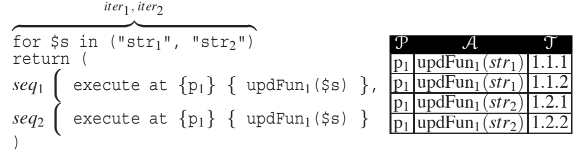
```
body ::= UpdateAction |
         "for" ... "return" body |
         body ("," body)*
```

A body can contain an expression in one of the three types, *(i)* an update action (possibly a 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 $\mathcal{T}$ of the DPUT are concatenations of numbers, separated by a dot. We initialize $t_{prefix} = 1$ for executions done locally on behalf of the initiating query. The query body mimicks the parse tree of the query, which is then "executed" recursively as follows (starting with b=root and $t_b = \emptyset$) to generate all tags:

- if b is a sequence constructor, we process all sequence expressions $s_1, \cdots, s_n$ while assigning $t_{s_i} = t_b.i$.
- if b is a for-loop with iterations $1 \le i \le n$, 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 initialization $t_{prefix} = tag$.

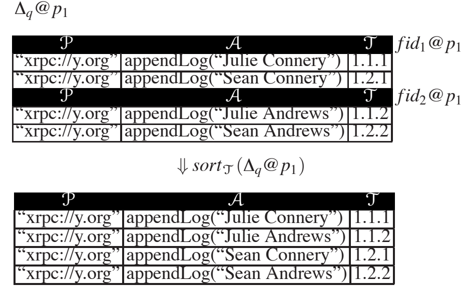
Figure 4 shows how the tags are constructed from the body of our example update query. The initial t_{prefix} is 1. The for-loop with two iterations introduces the second number, 1 for the first iteration, and 2 for the second. Inside the loop body we find a sequence constructor, introducing a third number in the tag. Inside this sequence constructor, the update actions are found and tagged.



```
                     iter1, iter2
for $s in ("str1", "str2")
return (
seq1 { execute at {p1} { updFun1($s) },
seq2 { execute at {p1} { updFun1($s) }
)
```

$\mathcal{P}$	$\mathcal{A}$	$\mathcal{T}$
p1	updFun1(str1)	1.1.1
p1	updFun1(str1)	1.1.2
p1	updFun1(str2)	1.2.1
p1	updFun1(str2)	1.2.2

Figure 4: The Body of the Example Query and its DPUT



$\Delta_q@p_1$

$\mathcal{P}$	$\mathcal{A}$	$\mathcal{T}$	
"xrpc://y.org"	appendLog("Julie Connery")	1.1.1	$fid_1@p_1$
"xrpc://y.org"	appendLog("Sean Connery")	1.2.1	
$\mathcal{P}$	$\mathcal{A}$	$\mathcal{T}$	$fid_2@p_1$
"xrpc://y.org"	appendLog("Julie Andrews")	1.1.2	
"xrpc://y.org"	appendLog("Sean Andrews")	1.2.2	

$\Downarrow sort_{\mathcal{T}}(\Delta_q@p_1)$

$\mathcal{P}$	$\mathcal{A}$	$\mathcal{T}$
"xrpc://y.org"	appendLog("Julie Connery")	1.1.1
"xrpc://y.org"	appendLog("Julie Andrews")	1.1.2
"xrpc://y.org"	appendLog("Sean Connery")	1.2.1
"xrpc://y.org"	appendLog("Sean Andrews")	1.2.2

Figure 5: The pending update list $\Delta_q@p1$ was created by two XRPC calls executed after each other. Sorting those at commit time on $\mathcal{T}$ achieves deterministic update order.

Note that the tag construction algorithm respects the for-loop and sequence construction order just like XUF 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 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, then 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 [12]), 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. $t_{prefix}.iter_{\{1,2\}}.seq_1$):

```
<xrpc:request module="filmdb" uri="http://x.org/film.xq"
                              method="appendLog">
 <queryID host="x.org" timestamp="32414232" timeout="180"/>
 <xrpc:call 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 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 5 shows the pending update list $\Delta_q@p_1$ at peer p_1 (`y.org`) including the extra column $\mathcal{T}$. 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 $\mathcal{F}'_u$ (note that the XRPC Request above includes the `queryID` element). Sorting the pending update list on $\mathcal{T}$ achieves the desired deterministic update order.

5. RELATED WORK

Among others, our work is closely related to XQueryD [21]. The syntax of XRPC is based on that of XQueryD, but in a more restricted form, namely, instead of executing an arbitrary query at a remote peer, XRPC only allows execution of predefined module functions. 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, since the binding of the parameters of an XRPC function application is done by the compiler as if it was a normal function application. Unlike XRPC, XQueryD does not explicitly address its underlying communication protocol.

In Active XML ([2], [1]), calls to service functions are embedded in XML documents. The evaluation of a service call `sc` results in an XML fragment, which is inserted into the original XML document as a sibling of the service call. Similar to nested XRPC calls, the resulting XML fragment of a service call can again contain service calls. AXML differs with XRPC mostly in the execution model. AXML uses a lazy evaluation model, which means that a service call will not be evaluated unless the data is required, while XPRC put the emphasis on bulk operation to speed up the evaluation of RPC calls. AXML defines that the AXML documents are dynamic documents, which means that evaluating the same service call multiple times can possibly return different results. This is another major difference between AXML and XRPC, because XRPC uses the Repeatable-Read isolation level.

Galax Yoo-Hoo! [19] 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. Yoo-Hoo! differs with XRPC mainly in the implementation of the stub. In the former system, the generated stub is a user-defined function in XQuery. The stub builds the SOAP request message using XQuery's element construction, which is then passed to a new function supporting SOAP calls. The parameters are copied into the SOAP message and this approach results in one more intermediate step of element reconstruction, comparing with XRPC. Bulk operation is not supported by Yoo-Hoo!. Another major difference is that the SOAP RPC implementation of Galax Yoo-Hoo [19] is based on an equivalent of the `http_post` method, but supports only *one* destination URI for each imported web services module, namely, the one to which the web service is bound.

DXQ [22] is a specification of a Distributed XML-Query (DXQ) network. A DXQ-Network consists of one or more XDQs (XML-Query Distributor) and XDPs (XML-Document Provider). A DXQ-client can send XQuery queries to an XDQ for distributed execution. The XDQ is responsible for disseminating a request to all registered XDPs, gathering results from the XDPs and sending all results back to the client. The main part of DXQ is an HTTP-like protocol for the communication between the components in a DXQ-network. DXQ is thus more a communication system than an XQuery execution system, in the sense that it does not specify how the XQuery queries are processed. Instead of inventing a new communication protocol, XRPC chooses SOAP as underlying transport protocol because of its interoperability and its seamless integration with the (Web) Service Oriented Architectures. Another benefit of using SOAP is that applications can even directly communicate with XRPC servers by generating XRPC request messages in SOAP format without running an XRPC client.

In the area of distributed query processing and transactions, much prior research has been done. There have been several surveys on these topics, such as [15], [23] and parts of the book of [20]. Distributed XRPC updates with isolation (rule $\mathcal{F}'_u$) need a distributed commit protocol, for which any of the Two-Phase commit protocol [20], the Paxos Commit algorithm [11] and the distributed Sagas [9] could be used.

6. CONCLUSION

In this paper, we presented XRPC, a minimal XQuery extension that enables distributed query execution.[6] We first gave a formal definition of the syntax and the semantics of XRPC, including the semantics of distributed update queries, that follow from the use of XQuery Updating Functions over XRPC. Part of our proposal is the SOAP XRPC message protocol, that allows to seamlessly integrate our approach with web service architectures. SOAP XRPC supports the concept of Bulk RPC; the execution of multiple function calls in a single message exchange. We have shown that the loop-lifting technique, pervasively applied in our MonetDB/XQuery system for the translation of XQuery expressions to relational algebra can easily generate such Bulk RPC Requests. A small experimental section demonstrated that Bulk RPC strongly improves performance of queries that execute XRPC calls inside `for`-loops. Finally, we defined a deterministic update semantics for the XQuery Update Facility, and showed how the SOAP XRPC can be extended to guarantee deterministic order in distributed update scenarios.

Short term future work includes extensive performance testing, such that we can assess the scalability when large XML payloads are involved, and an increasing number of peers. On the longer term, we regard the implementation of XRPC in MonetDB/ XQuery a major step towards our AmbientDB [4] architecture, that aims to create versatile P2P data management technology.

7. REFERENCES

[1] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic xml documents with distribution and replication. In *SIGMOD Conf.*, pages 527–538, 2003.

[2] S. Abiteboul, I. Manolescu, and E. Taropa. A framework for distributed xml data management. In *Intl. Conf. on Extending Database Technology*, pages 1049–1058, 2006.

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

[4] P. Boncz and C. Treijtel. AmbientDB: relational query processing in a P2P network. In *International Workshop on Databases, Information Systems, and P2P Computing (DBISP2P)*, (LNCS/LNAI), Springer-Verlag, Berlin, Germany, September 2003.

[6] XRPC will be available in the next open-source release of MonetDB/XQuery, scheduled for December 2006.

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

[6] F. Cohen. Discover SOAP encoding's impact on web service performance, March 2003. http://www-128.ibm.com/developerworks/webservices/library/ws-soapenc.

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

[8] 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, July 2006. http://www.w3.org/TR/2006/CR-xpath-datamodel-20060711.

[9] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD Conf.*, pages 249–259, New York, NY, USA, 1987. ACM Press.

[10] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. van Engelen, and M. J. Lewis. Toward Characterizing the Performance of SOAP Toolkits. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 365–372, Washington, DC, USA, 2004. IEEE Computer Society.

[11] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1):133–160, 2006.

[12] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *VLDB*, pages 252–263, September 2004.

[13] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation 24 June 2003, June 2003. http://www.w3.org/TR/2003/REC-soap12-part1-20030624.

[14] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation 24 June 2003, June 2003. http://www.w3.org/TR/2003/REC-soap12-part2-20030624.

[15] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[16] S. Lyubka. SHTTPD: Simple HTTPD. http://shttpd.sourceforge.net.

[17] N. Mitra. SOAP Version 1.2 Part 0: Primer. W3C Recommendation 24 June 2003, June 2003.

[18] A. Ng, S. Chen, and P. Greenfield. An Evaluation of Contemporary Commercial SOAP Implementation. In *AWSA*, pages 64–71, April 2004.

[19] N. Onose and J. Siméon. XQuery at Your Web Service. In *WWW*, pages 603–611, 2004.

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

[21] C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *IIWeb*, pages 116–121, September 2004.

[22] C. Thiemann, M. Schlenker, and T. Severiens. Proposed Specification of a Distributed XML-Query Network. *CoRR*, cs.DC/0309022, 2003.

[23] C. Yu and C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, 1984.