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.
The XRPC Language Extension

In this chapter, we introduce the XRPC language extension. We start with a discussion of the design criteria that the XRPC extension must satisfy. Then, we give the definition of XRPC syntax in Section 3.2, and the SOAP XRPC message format in Section 3.3. In Section 3.4, we spend considerable time in rigorously defining the formal semantics of XRPC. Finally, in Section 3.5, we outline the initial implementation of XRPC in MonetDB/XQuery, including the correspondence of Bulk RPC with the loop-lifting technique applied by the pathfinder compiler.

3.1 Design Considerations

The XRPC language extension must satisfy the following design criteria:

- The extension must be orthogonal to all XQuery features, including XQUF.
- The extension must support all XDM data types.
- The extension must be unambiguous, i.e., if a query containing our extension is run on systems that do not support this extension, the query must not produce unexpected results.
- The extension must allow functions in the same module to be executed both locally and remotely, and it must also allow functions from different modules to be executed remotely.
- The extension should be clean, i.e., only require minimal changes to the XQuery standard.
- The extension should be well-defined and have easy to understand semantics.
- The extension should provide potentials for efficiency and scalability.
- The extension should be easy to support by heterogeneous XQuery engines.

Besides the existing proposals [74, 134, 144, 75], we have considered several alternatives, which all satisfy the first two design criteria:

1. Use a special namespace prefix, e.g., rpc, to indicate that all functions from an imported module, bound to this special namespace prefix, will be executed remotely (for short, we call modules that contain functions that will be executed at a remote peer as RPC modules):

   ```
   import module namespace rpc = "rpc-functions"
   at "http://example.org/foo.xq", "http://example.org/bar.xq";
   rpc:foo("foo.example.org", rpc:bar("bar.example.org", 42))
   ```
This approach is easy to understand. However, its semantics is ambiguous because namespace prefixes do not have significant meaning in XQuery. Query writers could accidentally use the special namespace prefix, causing queries to return unexpected results on different XQuery systems. Moreover, functions in the imported modules could only be run either locally or remotely, as the XQuery standard [38] does not allow the same module to be loaded twice in a query: “It is a static error [err:XQST0047] if more than one module import in a Prolog specifies the same target namespace.”.

This approach allows multiple different modules to be imported by listing multiple module location URLs in the at-hint. The drawback of this approach is that multiple modules are loaded under the same namespace, which is restrictive and can lead to clashes. To overcome this problem, one could, instead of using one special namespace prefix, use namespace prefixes that start with a predefined string, e.g., "rpc-", to import RPC modules:

```xquery
import module namespace rpc-foo = "rpc-functions" at "http://example.org/foo.xq";
import module namespace rpc-bar = "rpc-functions" at "http://example.org/bar.xq";
rpc-foo:foo("foo.example.org", rpc-bar:bar("bar.example.org", 42))
```

However, this alternative is even less clean: it is even more likely that query writers would accidentally use a namespace rpc-∗ that causes queries to behave differently on systems with our extension.

2. Use a special namespace, e.g., “http://www.w3.org/TR/soap/”, to indicate RPC modules:

```xquery
import module namespace rpc = "http://www.w3.org/TR/soap/" at "http://example.org/foo.xq";
import module namespace bar = "http://example.org/bar" at "http://example.org/bar.xq";
rpc:foo("foo.example.org", bar:bar(42))
```

Comparing with the first approach, this approach is much cleaner by assigning special semantics to a namespace, which has significant meaning in XQuery. However, this approach allows only one module with the special target namespace to be imported in a query, which is a major limitation. Similar as for the first approach, this problem might be alleviated by defining a special namespace prefix, e.g., “http://www.w3.org/TR/soap/”, and all modules, whose target namespaces start with this prefix, are recognised as RPC modules:

```xquery
import module namespace foo = "http://www.w3.org/TR/soap/foo" at "http://example.org/foo.xq";
import module namespace bar = "http://www.w3.org/TR/soap/bar" at "http://example.org/bar.xq";
foo:foo("foo.example.org", bar:bar("bar.example.org", 42))
```

On the other hand, this workaround intensifies a next problem: modules declared using this special SOAP namespace (as a prefix) cannot be used as normal, local modules, unless the module definitions are duplicated using a different target namespace (the same situation exists in the first approach). This is bad for software re-use and maintenance.

3. Use the at-hint to indicate an RPC module:

```xquery
import module namespace foo = "http://example.org/foo.xq" at "http://www.w3.org/TR/soap/";
import module namespace bar = "http://example.org/bar.xq" at "http://www.w3.org/TR/soap/";
foo:foo("foo.example.org", bar:bar("bar.example.org", 42))
```

Comparing with previous approaches and their alternatives, this approach is more favorable. First of all, the semantics of this approach is completely legal under the XQuery specification, because XQuery 1.0 [38] has specified that “The URILiterals that follow
the at keyword are optional location hints, and can be interpreted or disregarded in an implementation-defined way.". Moreover, this approach allows multiple RPC modules to be imported using different namespace prefixes bindings, avoiding introducing unnecessary clashes. Nevertheless, modules still cannot be imported both as local modules and as RPC modules.

The major disadvantage of this approach is the at-hint, whose semantics is very counter-intuitive. The original intention of the XQuery specification is to use the at-hint to specify the physical location of the module, while in this approach, the at-hint contains a logical namespace. Additionally, this approach forces modules to have the same target namespaces as their physical location, defeating the purpose of target namespaces that should be logical identifiers of modules. A workaround of the latter disadvantage would be to include the special namespace "http://www.w3.org/TR/soap/" in the at-hint as an additional URL, whose semantics is different than other URLs in the at-hint:

```
import module namespace foo = "http://example.org/foo" at "http://example.org/foo.xq", "http://www.w3.org/TR/soap/";
import module namespace bar = "http://example.org/bar" at "http://example.org/bar.xq", "http://www.w3.org/TR/soap/";
foo:foo("foo.example.org", bar:bar("bar.example.org", 42))
```

but this mix of at-hint containing both physical and logical hints makes the design even more messy.

4. Extend the XQuery language with a new module importing feature:

```
import rpc-module namespace foo = "http://example.org/foo" at "http://example.org/foo.xq";
import rpc-module namespace bar = "http://example.org/bar" at "http://example.org/bar.xq";
import module namespace foo-loc = "http://example.org/foo" at "http://example.org/foo.xq";
foo:foo("foo.example.org", bar:bar("bar.example.org", 42)),
foo-loc:foo(bar:bar("bar.example.org", 42))
```

With such a language extension, the semantics of the imported modules are clear. Modules can be imported both as local modules and as RPC modules. However, this approach is a language extension, while one of our design criteria is to limit the change to the XQuery standard to a minimum.

All four approaches discussed above suffer from a common problem: they do not have an elegant, flexible way to specify the destination of the remote peer on which a function is to be executed. The signature of each function is implicitly extended with an additional leading string parameter to hold the URL of the destination peer. Such a design could be considered unclean. In the next two approaches, more attention is paid on where and how the destination URLs should be specified.

5. Put RPC calls in extension expressions and specify in the pragmas of the extension expressions which functions should be executed remotely and on which peers:

```
import module namespace foo = "http://example.org/foo" at "http://example.org/foo.xq";
import module namespace bar = "http://example.org/bar" at "http://example.org/bar.xq";
declare namespace rpc = "http://www.w3c.org/TR/SOAP";
(# rpc:rpc-call [foo:foo, "foo1.example.org", "foo2.example.org"] #)
| foo:foo(bar:bar(42),
  (# rpc:rpc-call [bar:bar, "bar1.example.org", "bar2.example.org"] #)
  (bar:bar(24))) } + 10
```
As shown in the example above, we identify in a pragma the exact function that should be executed remotely, followed by a list of URLs of the remote peers. In this way, modules only need to be imported once, but can be used both as local and as RPC modules. We can specify per function, instead of per module, whether it should be executed remotely or not. At the syntax level, this approach is much more flexible than the previous approaches, since pragmas are allowed everywhere where a path expression would be allowed. Thus, certain sub-expressions of a query could live within, while other sub-expressions could exist outside the scope of a pragma. Another advantage of using a pragma is that the semantics of the extension expressions is unambiguous, since it has been specified by the XQuery standard that: “An extension expression is an expression whose semantics are implementation-defined. Typically a particular extension will be recognised by some implementations and not by others.” This makes pragma an ideal place to define additional language features.

However, pragmas are often considered to be difficult to understand, and have not been generally adopted. From the above example, it can already be seen that allowing pragmas everywhere in a query will quickly make a query unreadable. Query writers are also required to have a very good understanding of the exact semantics of pragmas, since they apply to the whole expression enclosed in the curly braces behind them. For instance, in the above example, the two calls to the function \texttt{bar:bar()} should be executed on different peers (i.e., the first one is a local function call while the second one is an RPC call), this requires that the second pragma is only specified for the second \texttt{bar:bar()}; otherwise the query will have very different semantics, e.g.:

```
(foo:foo(bar:bar(42), bar:bar(24)) + 10)
```

6. Extend the XQuery language with a new function application syntax, i.e., \texttt{QName(...)}\@(URILiteral (',', URILiteral)*):

```
import module namespace foo = "http://example.org/foo" at "http://example.org/foo.xq";
import module namespace bar = "http://example.org/bar" at "http://example.org/bar.xq";
foo:foo(bar:bar(42)@("bar.example.org", "foo2.example.org"), bar:bar(24))@("foo1.example.org", "foo2.example.org")
```

The pros and cons of this approach are similar to that of the pragma approach. It allows modules to be imported once and used both as local and as RPC modules. The syntax is very flexible, as destinations can be specified for each function, and remote function calls can be made everywhere in a query where a function call is allowed. The semantics is easy to understand and is less error-prone, as the list of destinations only applies to its associated function.

Our final choice is to introduce a language extension by adding one new statement to XQuery to allow remote function application. At the syntax level, our language extension is inspired by that of [144]. By comparing different alternatives of adding a query shipping feature to XQuery, we conclude that a language extension results in a cleaner and more flexible design than extending existing XQuery features with new semantics. We consider one new statement to be a minimal change to the XQuery standard. Later in this chapter, we discuss
3.2 XRPC Syntax

Remote function applications take the XQuery syntax:

```
"execute" "at" {ExprSingle} {FunApp(ParamList)}
```

where ExprSingle is an XQuery xs:string expression that specifies the URI of the peer on which FunApp is to be executed. The function to be applied can be a built-in or a user-defined function. For user-defined functions, we currently restrict ourselves to functions defined in an XQuery Module. A small (future) extension to the network protocol would also allow functions defined inside the query to be executed over XRPC. 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 an import module statement, the module URI may be supplemented by a so-called at-hint, which also is a URI. For a precise syntax definition, Table 3.1 shows the rules of the XQuery 1.0 grammar that were changed.

The current choice to allow functions defined in XQuery modules is due to efficiency and security reasons. 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. The feature of prepared queries is well-known for an RDBMS, allowing a parametrised query plan to be parsed and optimised off-line, such that an application can quickly enter actual parameters in the prepared plan and execute it. 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 [41]. For security reasons, by allowing only modules, it is trivial to specify which
modules are allowed to be executed or not. XRPC can be easily extended to support free form queries, with some extra work on preserving the efficiency and security issues.

**The XRPC URI Scheme** We also introduce a new URI scheme, named xrpc to indicate that the remote peer specified in an xrpc URL is able to process XRPC requests. The generic form of such URIs is:

\[
\text{xrpc://}(\text{host})[::\text{port}][/\text{path}]
\]

The “xrpc://” indicates the network protocol. The second part “(host)[::port]” identifies a remote peer. The third part “[/path]” is an optional local path at the remote peer.

The xrpc URI scheme is accepted in the destination URI of execute at. Moreover, we have extended the built-in functions fn:put() and fn:doc() to accept the xrpc URI scheme in their $uri parameters. Given a URL xrpc://P/\mathcal{D}, fn:put() stores the XML tree rooted at its $node parameter on the remote peer P as document \mathcal{D}, which possibly overwrites the existing \mathcal{D}. With fn:doc(), \mathcal{D} could then be retrieved (over HTTP) from (the XRPC server on) peer P. As we will see in Section 5.7, this extension enables supporting updates on remote documents identified by xrpc:// URIs.

**Examples** 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:

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

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

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

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

```xml
import module namespace f="films" at "http://x.example.org/film.xq";
<film>
  execute at {"xrpc://y.example.org"} {f:filmsByActor("Sean Connery")}
</film> (Q3-1)
```

This example yields: 

\[
<\text{name}>\text{The Rock}\langle/\text{name}\rangle<\text{name}>\text{Goldfinger}\langle/\text{name}\rangle<\text{name}>\text{Green Card}\langle/\text{name}\rangle.
\]

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

```xml
import module namespace f="films" at "http://x.example.org/film.xq";
<film>
  for $actor in ("Julie Andrews", "Sean Connery")
  let $dst := "xrpc://y.example.org"
  return execute at {$dst} {f:filmsByActor($actor)}
</film> (Q3-2)
```

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

```xml
import module namespace f="films" at "http://x.example.org/film.xq";
<film>
  for $actor in ("Julie Andrews", "Sean Connery")
  for $dst in ("xrpc://y.example.org", "xrpc://z.example.org")
  return execute at {$dst} {f:filmsByActor($actor)}
</film> (Q3-3)
```
Complex communication patterns may be programmed with XRPC, especially if recursive functions are used. The query below executes the RPC on a set of destination peers, uniting all results, and does so by constructing a binary spanning tree of recursive RPC calls.

```xml
module namespace film="films";
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 ($cnt.gt.1) then execute at {$peer1} {film:recursiveActor($dsts1, $actor)} else (),
     doc("filmDB.xml")//name[../actor=$actor],
     if ($cnt.gt.2) then execute at {$peer2} {film:recursiveActor($dsts2, $actor)} else ())
}; (Q3-4)
```

### 3.3 SOAP XRPC Message Format

The Simple Object Access Protocol (SOAP) is the XML-based message format used for web services [128, 89, 90], and we propose the use of SOAP messages over HTTP as the network protocol underlying XRPC. SOAP web service interactions usually follow an 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 subprotocol called “SOAP RPC” is in common use [90]. 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) [71], 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 subformat (SOAP XRPC ≠ SOAP RPC).

#### 3.3.1 XRPC Request Messages

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, an at-hint location, a function name method and its arity. The module definition must be accessible (via an HTTP connection) for the remote peer at the location given by the at-hint. In this way, we can rely on the XQuery facility to import the module from an arbitrary URL. 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 XRPC request message for the first example query that looks for films with Sean Connery:

```xml
<?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>
```
30

3.3. SOAP XRPC MESSAGE FORMAT

<env:Body>
</env:Envelope>

Atomic Values Atomic values are represented with atomic-value elements, and are annotated with their (simple) XML Schema Type in the xsi:type attribute. Thus, the heterogeneously typed sequence consisting of an integer 2, a double 3.1 and a string “abc” would become:

<Node Typed XML Elements XML nodes are passed by value in an (element) element:

Similarly, the XML Schema “XRPC.xsd” defines enclosing elements for document, text, attribute, processing-instruction, and comment nodes. A document node is represented in the SOAP message as a (document) element that contains the serialised document root. The text, comment and processing-instruction nodes are serialised textually inside the respective elements (text), (comment) and (processing-instruction). An attribute node is serialised inside an (attribute) element, for example, the attribute node x="y" is serialised as: {xrpc:attribute x="y"/}.

User-Defined Types XRPC fully supports the XDM, a requirement for making it an orthogonal language feature. This implies that 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 named types are enclosed in SOAP messages by (element) elements, with an xsi:type attribute annotating their type. The XQuery system implementing XRPC should include an xsi:schemaLocation declaration as well as an xmlns namespace definition inside the (Envelope) element when values of such imported element types occur in the SOAP message. If a parameter has an anonymous user-defined schema type, its type information is lost. However, this can be avoided exploiting a future protocol extension (discussed in Section 5.5) by including the lowest ancestor-or-self element with a named schema type in the SOAP messages.

3.3.2 XRPC Response Messages

XRPC response messages follow the same principles. Inside the body is now an XRPC response element that contains the result sequence of the remote function call:
### 3.3.3 XRPC Error Message

If 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 ([128], [89]). Thus, any error will cause a run-time error at the site that originated the query. Updating queries with 2PC enabled behave similarly, since update effects will only be applied if a query succeeds. If 2PC is not enabled, a failed updating query might already have applied changes somewhere. The exact semantics of updating queries is discussed in Section 3.4.2. As an example, the following SOAP Fault message indicates that a required module could not be loaded:

```xml
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
  xmlns:env:Fault>
</env:Fault>
</env:Envelope>
```

**Remarks** Our discussion of SOAP XRPC message is not fully done yet. In the next section, we will extend the format with support for isolation and updates. Then, in Section 3.5.2 we describe the Bulk RPC feature, that allows a single message to request multiple function calls. Finally, in Section 4.4.1 we describe a small extension to include tag attributes in call elements that allows keeping track of a deterministic distributed update order. The XML Schema Definition of the XRPC SOAP messages is given in Appendix A.

### 3.4 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 serialisability 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.
Notations We use the following notation and terms in this section:

- \( \mathcal{P} \) 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 xrpc protocol that contains a host and (optionally) a port number.

- \( \mathcal{F} \) denotes a set of XRPC function applications. An XRPC call \( f^{p_0 \rightarrow p_j} \) that triggered from \( p_i \) that causes function \( f \) to be executed at \( p_j \) is an updating XRPC call \( (f^{p_0 \rightarrow p_j}) \in \mathcal{F}_u \), if it calls an updating function; otherwise, it is a non-updating XRPC call \( (f^{p_0 \rightarrow p_j}) \in \mathcal{F}_r \). If the evaluation of an XRPC call \( f^{p_0 \rightarrow p_j} \) requires evaluation of other XRPC call(s) at \( p_j \), we term \( f^{p_0 \rightarrow p_j} \) 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 \( f^{p_0 \rightarrow p_j} \) 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 \( f^{p_0 \rightarrow p_j} \in \mathcal{F}_q \), 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 [67] defines that each expression is normalised to a core expression, which then is defined by a semantic judgment \( \text{dynEnv} \vdash \text{Expr} \Rightarrow \text{val} \). The semantic judgment specifies that in the dynamic context \( \text{dynEnv} \), the expression \( \text{Expr} \) evaluates to the value \( \text{val} \), where \( \text{val} \) is an instance of the XQuery Data Model (XDM). For now, we simplify the dynamic environment to a database state \( \text{db} \) (i.e., the documents and their contents stored in the XML database): \( \text{dynEnv} \simeq \text{db} \). The \( \text{dynEnv.docValue} \) from the XQuery Formal Semantics [67] corresponds to \( \text{db} \) used here. To indicate a context at a particular peer \( p \), we write \( \text{db}^p \).

- When considering that a database may be changed by updates, we can view it as a function over time \( t \) as \( \text{db}^p(t) \). In our formal rules, the default assumption on database states is that they stay equal over time, unless otherwise stated. When the time context \( t \) is clear, the shorthand notation \( \text{db}^p \) is used to refer to the current database state.

### 3.4.1 Read-Only XRPC Semantics

**Basic Read-Only XRPC** The semantics of executing a read-only function \( f^{p_0 \rightarrow p_x} \) \((f \in \mathcal{F}_r)\) is defined by extending the XQuery 1.0 semantic judgments with a new rule:\(^1\)

\[
\begin{align*}
\text{db}^{p_0}(t_0) \vdash (\text{call})\{s2n(v_1), \ldots , s2n(v_n)\} / (\text{call}) \Rightarrow \text{call}; \\
\text{send}^{p_0 \rightarrow p_x} \text{request}(m, f, \text{call}); t_0 \geq t_0 \\
\text{db}^{p_0}(t_0) \vdash s2n(f, n2s(\text{call} / [1]), \ldots , n2s(\text{call} / [n])); \Rightarrow \text{res}; \\
\text{send}^{p_0 \rightarrow p_x} \text{reply}(\text{res}); \\
\text{db}^{p_0}(t_0) \vdash n2s(\text{res}) \Rightarrow \text{vres}; \\
\text{db}^{p_0}(t_0) \vdash f^{p_0 \rightarrow p_x}(v_1, \ldots , v_n) \Rightarrow \text{vres};
\end{align*}
\]

\((\mathcal{R}_f)\)

This rule \( \mathcal{R}_f \) states that execution at \( p_0 \) of the read-only XRPC call \( f^{p_0 \rightarrow p_x}(v_1, \ldots , v_n) \) in the dynamic context \( \text{db}^{p_0}(t_0) \) (without further assumption on \( t_0 \)) starts with constructing a \( \text{call} \) element that contains the SOAP representation of all parameters \( v_i \). This XML representation,

\(^1\)In our rules, we use the ‘;’ sign to suggest an order in the evaluation of the statements.
described in the previous Section 3.3, is created by the sequence-to-node marshalling function \textit{s2n()}}, discussed below. Then, the request \((m, f, call)\) is sent to peer \(p_x\). Here, \(m\) is the module URI (plus at-hint) in which function \(f\) is defined. The function \(f\) is then evaluated as a normal local function in the dynamic context of the remote peer \(db^{p_x}(t_x)\), where we only assume \(t_x \geq t_0\). The parameters of \(f\), are obtained by using the inverse node-to-sequence marshalling function \textit{n2s()} to produce the result node \(res\). This result \(res\) is sent back to peer \(p_0\), which finally converts \(res\) into the result sequence \(v_{res}\).

This definition inductively relies on the XQuery Formal Semantics to evaluate \(f\) locally at \(p_x\), and thus may trigger the evaluation of additional XRPCs if these happen to be present in the body of \(f\). Also, this definition covers execution of XRPC calls in the current database state \(db^{p_0}\), which we need for our basic purpose of defining the semantics of XRPC queries (in which case \(t_0\) is the current time point). Finally, this XRPC rule does not produce a new current local database state \(db^{p_0}\), nor any new remote database state \(db^{p_x}\) (i.e., it defines read-only semantics).

**Parameter Marshalling** The SOAP representation of a sequence \$seq is created in a new \(\langle\text{sequence}\rangle\) node by the function:

\[
\text{declare function s2n($seq as item*) as node()}
\]

The inverse transformation (from \(\langle\text{sequence}\rangle\) representation to real item sequence) is provided by:

\[
\text{declare function n2s($n as node[]) as item*}
\]

For example, we get \(\langle\text{"abc", 42}\rangle\) from calling:

\[
\text{n2s(<xrpc:sequence>}
  \text{<xrpc:atomic-value xsi:type="xs:string">abc</xrpc:atomic-value>}
  \text{<xrpc:atomic-value xsi:type="xs:integer">42</xrpc:atomic-value>}
\text{</xrpc:sequence>}
\]

An important characteristic of the function \textit{n2s()} is that it guarantees that for node-typed parameters (i.e., those represented as \(\langle\text{element}\rangle\), \(\langle\text{text}\rangle\), \(\langle\text{document}\rangle\), \(\langle\text{attribute}\rangle\), \(\langle\text{comment}\rangle\) and \(\langle\text{processing-instruction}\rangle\)) an XDM node of the correct type is returned as a separate XML fragment. This guarantees that evaluating the upwards and horizontal XPath axes on such nodes will return empty results. It may be tempting to return element nodes under the identity found in the message (i.e., \$request/xrpc:call/xrpc:sequence[i]/xrpc:element/*), but this would allow a query to navigate to e.g. the SOAP envelope element, or the other function parameters.

One should note that \textit{n2s()} and \textit{s2n()} are internal functions only that do not need to be exposed to XRPC users, and in fact do not need to exist in reality, as each XRPC system implementation may have its own internal (efficient) mechanisms to process SOAP messages. In case of MonetDB/XQuery, beyond shredding the SOAP request and response messages, we do not spend any effort in \textit{n2s()} nor \textit{s2n()} on element construction to retrieve node values of the correct type, as our implementation directly chops up the shredded XML message in separate XML fragments per function parameter, and modifies node types internally (as the SOAP messages are invisible to the user, their integrity can be compromised at will by the system). It is possible, though, to implement \textit{n2s()} and \textit{s2n()} purely in XQuery, as we will show when we discuss the XRPC wrapper, that allows arbitrary XQuery processors to participate in distributed XRPC queries in Section 4.2.
A final detailed remark on parameter marshalling is that XRPC requires the caller to perform parameter up-casting. The rationale is that such casting is already part of the standard function application code generated by any XQuery system, thus it is easy to do at the caller for XRPC calls, and it makes it easier to implement XRPC handlers that have no or limited XQuery capabilities (e.g. wrapped outside web services as in [134]).

Pass-By-Value An important choice implied by making $n2s()$ and $s2n()$ explicit in our Formal Semantics is to enforce by-value parameter passing in XRPC. If nodes are passed as parameters of an XRPC call, they will be serialised into a SOAP message, shipped to the remote side, and there new nodes will be constructed virtually of the correct type with equal-valued contents, but with different node identifiers. This can lead to a number of semantic differences between local and remote function application. We already mentioned that XPath navigation from node parameters over non-downwards axes (e.g. parent, following) will always produce empty results on the remote side. More subtly, if a function is invoked over XRPC with two nodes as parameters that have a descendant-or-self relationship, XRPC parameter marshalling will destroy this relationship at the remote side\(^2\). Finally, the XQuery Formal Semantics specifies that some consistent order should be enforced over nodes from different documents, but our semantics will not respect this order on their copies when shipped over XRPC.

The rationale behind this by-value choice is that a by-reference semantics would lead to complications when the upwards or sideways XPath axes are invoked on node parameters (or results) of XRPC calls. Correctly supporting that would either lead to the need to ship the full XML data fragment for all node parameters upfront (defeating the purpose of function shipping) or cause implicit communication when navigating beyond the descendants of such nodes. Obviously, call-by-value semantics complicate life when XRPC is used as the target language for automatic query distribution (as opposed to explicit XRPC query processing, where we can assume the query writer to be aware of the call-by-value semantics). In that case, the query optimiser has the task to make sure by-value parameter passing does not affect query semantics. The simplest solution is to refrain from function shipping in problematic cases, but more sophisticated solutions may be found for some query patterns.

\(^2\)In Section 5.5, we discuss a future XRPC protocol extension that allows node parameters to be referred to using an xrpc:fragid and an xrpc:nodeid attribute that together identify a node serialised earlier in a special (fragment) section of an XRPC message. This alternative node representation can be used for nodes that are a descendant-or-self of another parameter that is fully serialised in the SOAP message. The $s2n()$ function would then be altered to return nodes from the XML fragment that corresponds with that fully serialised parameter. This change of semantics ensures that ancestor/descendant relationships among parameters at the calling peer are preserved at the remote XRPC peer. This indirect addressing is useful for compressing the SOAP message. Moreover, if applied maximally, the resulting pass-by-fragment result/parameter passing, allows an distributed XRPC rewriter to relocate parts of certain query predicates that do depend on node identity (i.e., node-valued join conditions whose predicates only contain descendant/ancestor XPath steps).
Nested XRPC Calls 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 Figure 3.1, the arrow ‘→’ should be read as “XRPC call”. The peers \( p_0, p_1, \ldots, p_l \) are not necessarily unique: some peer \( p_l \) (or in fact many such peers) may occur multiple times in this tree. When considering rule \( \mathcal{R}_\mathcal{F} \), the dynamic environment \( \text{dynEnv}^{db} \) containing the current database state \( db^{p_l} \) may thus be seen multiple times during query evaluation. In between those multiple function evaluations, other transactions may update the database and change \( db^{p_l} \). Thus, those different XRPC calls to the same remote peer \( p_l \) 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.

Repeatable Read XQuery users can control per query which semantics is used by using the XQuery declare option feature, setting \text{xrpc:isolation} either to “none” (rule \( \mathcal{R}_\mathcal{F} \)) or “repeatable”, defined by rule \( \mathcal{R}'_\mathcal{F} \):

\[
\begin{align*}
\text{db}^{p_0}(t_{q_0}^{p_0}) & \vdash \{ \text{call} \} \{ \text{call} \} \Rightarrow \text{call}; \\
\text{send}^{p_{m-1}}_{p_m} & \text{request}(q, m, f_r, \text{call}) ; \\
\text{db}^{p_0}(t_{q_0}^{p_0}) & \vdash \text{call}(f_r, (n2s(\text{call}/[1]), \ldots, n2s(\text{call}/[n]))) \Rightarrow \text{res}; \\
\text{send}^{p_x}_{p_{m-1}} & \text{reply}(q, \text{res}) ; \\
\text{db}^{p_0}(t_{q_0}^{p_0}) & \vdash n2s(\text{res}) \Rightarrow v_{\text{res}} ; \\
\text{db}^{p_0}(t_{q_0}^{p_0}) & \vdash f_r \Rightarrow v_{\text{res}} ; \\
\text{db}^{p_0}(t_{q_0}^{p_0}) & \vdash f_r \Rightarrow v_{\text{res}} .
\end{align*}
\]

The above rule \( \mathcal{R}'_\mathcal{F} \) specifies that for evaluating XRPC calls on behalf of query \( q \), peer \( p_x \) always uses the same database state \( \text{db}^{p_x}(t_{q_x}^{p_x}) \). Time \( t_{q_x}^{p_x} \) is typically the time that the first XRPC request of query \( q \) reached \( p_x \); but we place no specific restriction on it. Observe that a unique query identifier \( q \) is now passed as an extra parameter in the XRPC request so that a peer can recognise which XRPC calls belong to the same query and it can associate an isolated database state with it.

Clearly, XRPC with repeatable reads requires more resources to implement, as some database isolation mechanism (of choice) will have to be applied to retain \( \text{db}^{p_x}(t_{q_x}^{p_x}) \) across calls. The transaction mechanism of MonetDB/XQuery, for instance, uses snapshot isolation\[32\] based on shadow paging, which keeps copies of modified pages around. Systems that provide the isolation levels serialisable or repeatable reads (obviously) can also provide this semantics.

A quite common reason why a peer is called multiple times in the same query and why the need for repeatable reads arises, is when an XRPC call appears inside a for-loop. In Section 3.5.2 we describe how Bulk RPC helps avoid these costly isolation measures in case of simple XRPC queries (i.e., those that contain only one non-nested function application).

Other Isolation Levels If we would suppose that all peers involved in \( q \) support the isolation level snapshot isolation, and all would use the same timestamp \( t_q \) as the one in which the original query executes, i.e., \( t_{q_0}^{p_0} = \cdots = t_{q_x}^{p_x} = \cdots = t_{q_m}^{p_m} = t_q \), we could obtain the isolation level distributed snapshot isolation. Just using a globally consistent query timestamp is actually not enough for that, extra effort is needed to enforce distributed commits to happen at the same time point (one way to do that is to block or abort incoming reads while a node is in the prepared state – this is called the pessimistic approach in [158]). For this to be meaningful in practice, however, we would have to have a representation of \( t \) values (until now, this
3.4. XRPC FORMAL SEMANTICS

is left opaque) that allows a full ordering, thus enabling us to define a “happened before” query/transaction order \( t_{q_1} \ll t_{q_2} \).

However, as XRPC is also intended for use in P2P settings, we make no assumptions on a centralised distributed transaction coordinator that could give out unique and monotonically increasing \( t \) numbers. In absence of that, one could think of \( t \) numbers generated by Lamport Clocks [116], but while this method guarantees that a transaction that depends on a previous one (“happened before”) has a smaller Lamport clock value, the reverse inference cannot be made (i.e., meaningfully enforcing a transaction order depending on such \( t \)-s) unless all peers participate in all queries (which again is not a reasonable assumption in P2P). Of course, we can think of \( t \) as being “exact” (UTC) time, but as we do not want to assume either that all participating peers possess (synchronised!) Strontium grade precision clock hardware, this is only a theoretical notion. For this reason, we leave the maximum XRPC isolation currently at the repeatable read level, though finding a distributed isolation level useful in P2P is on our future work agenda.

**SOAP XRPC Extension: Isolation** XRPC uses repeatable reads semantics for requests that have the optional queryID child element in the xrpc:request element. The queryID in the SOAP message contains host and timestamp attributes that state on which host and at what UTC 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 big clock synchronisation differences. When the timeout passes, the isolated database state can be discarded, freeing up system resources. However, the local XRPC handler should still remember expired queryIDs, 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 queryIDs, as per host only the latest timestamp needs to be retained, and can be restricted to some sane time interval.

A timeout mechanism is inevitable, even if XRPC would use a 2PC-like coordination protocol to signal the finishing of a query (for updates, XRPC actually uses a 2PC protocol via Web Services Atomic Transaction [55]), because such a coordination protocol also needs a timeout to conclude that remote hosts are no longer responding. Automatically computing a good timeout value requires a cost model that takes into account the query, data-distribution, network, and peer characteristics – a task we leave for our future work on automatic query distribution. Therefore, the timeout to use is specified in the query using declare option xrpc:timeout (sec), so users and applications can set them according to their needs.

3.4.2 XRPC Update Semantics

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 (XQUF). The XQUF 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 \( \Delta \). Only after query execution has finished, are all updates in the pending update list to be applied and committed. This concept is quite similar to IO monads, used in functional languages like Haskell, that cleanly separate functional execution from any side-effecting actions.
CHAPTER 3. THE XRPC LANGUAGE EXTENSION

Basic Updating XRPC The semantics of executing a single updating function \( f^{p_0\rightarrow p_x} (f \in f_u) \), is defined by extending the XQuery 1.0 semantic judgments with a new rule:

\[
\begin{align*}
\text{db}^{p_0}(t_0) &\vdash (\text{call})\{\text{s2n}(v_1), \ldots, \text{s2n}(v_n)\}(/\text{call}) \Rightarrow \text{call}; \\
\text{send}^{p_0\rightarrow p_x}\text{request}(m, f_u, \text{call}); \quad t_x \geq t_0 \\
\text{db}^{p_x}(t_x) &\vdash f_u(\text{n2s}(\text{call} /^*\text{[1]}), \ldots, \text{n2s}(\text{call} /^*\text{[n]})) \Rightarrow \Delta; \\
\text{db}^{p_x}(t_x) &\vdash \text{applyUpdates}(\Delta) \Rightarrow \text{db}^{p_x}'; \\
\text{send}^{p_x \rightarrow p_0}\text{reply}() \\
\hline
\text{db}^{p_0}(t_0) &\vdash f^{p_0\rightarrow p_x}(v_1, \ldots, v_n) \Rightarrow () \quad (R_{x})
\end{align*}
\]

The above rule \( R_{x} \) states that update functions apply the pending update list \( \Delta \) immediately, producing a new current remote database state \( \text{db}^{p_x} \). For this purpose, we use the internal function \( \text{applyUpdates}() \) defined in the XQUF [58] that carries through all changes in a pending update list. Note that this rule executes an updating call between \( p_0 \) and \( p_x \) in database states from \( t_0 \) resp. \( t_x \) with no other assumptions than \( t_x \geq t_0 \). Typically, an implementation may choose to use \( \text{db}^{p_x} \), i.e. the latest database state to handle each XRPC request.

Remote execution of an XQUF updating function causes no new \( \text{db}^{p_0} \) state directly (it returns an empty pending update list), but does yield a new \( \text{db}^{p_x} \). This is a simplification, because \( f_u() \) itself may perform XRPC calls that modify database states of other peers involved in \( q \) – and potentially even \( \text{db}^{p_0} \) itself. While the local query \( q \) at \( p_0 \) always operates in \( \text{db}^{p_0}(t_0) \), if it performs multiple XRPC calls to the same peer \( p_x \), these calls will thus potentially see different states \( \text{db}^{p_x}(t_1) \), \( \text{db}^{p_x}(t_2) \), ..., which may even include the updates caused by the previous XRPC calls made for \( q \). Therefore, while easy to implement, this semantics does not guarantee repeatable reads, even allows lost updates at the same peer between multiple calls performed on behalf of the same query, and will cause non-atomic distributed commits to happen if XRPC execution is aborted halfway due to an error.

Atomic Updates with Isolation We now define an improved XRPC isolation level that provides repeatable reads as well as atomic distributed commit. Recall that the effects of XQUF updates are invisible until query execution finishes; only then is \( \text{applyUpdates}() \) invoked on the pending update list. In the previous rule \( R_{x} \), updates were visible directly after handling each individual XRPC request. The new rule \( R'_{x} \), given below, corresponds more closely to the intent of the XQUF in that no side effects of query \( q \) are visible at any involved peer \( p_x \) until the query commits.

The repeatable read isolation implies that peers defer applying pending update lists created by individual XRPC calls made on behalf of the same query \( q \) until the point that \( q \) actually commits. Thus, peers \( p_x \) must not only keep track of the database state \( \text{db}^{p_x}(t^{p_x}_q) \), but also of a collection of pending update lists \( \Delta^{p_x}_q = \cup_{q \in \{1, \ldots, U^{p_x}_q\}} \Delta^{p_x}_q(i) \), where \( U^{p_x}_q \) is the number of updating XRPC calls \( p_x \) has handled so far for \( q \).

\[
\begin{align*}
\text{db}^{p_0}(t^{p_0}_q), \Delta^{p_0}_q &\vdash (\text{call})\{\text{s2n}(v_1), \ldots, \text{s2n}(v_n)\}(/\text{call}) \Rightarrow \text{call}; \\
\text{send}^{p_0\rightarrow p_x}\text{request}(q, m, f_u, \text{call}); \\
\text{db}^{p_x}(t^{p_x}_q), \Delta^{p_x}_q &\vdash f_u(\text{n2s}(\text{call} /^*\text{[1]}), \ldots, \text{n2s}(\text{call} /^*\text{[n]})) \Rightarrow \Delta^{p_x}_q(U^{p_x}_q); \\
\text{send}^{p_x \rightarrow p_0}\text{reply}() \\
\hline
\text{db}^{p_0}(t^{p_0}_q), \Delta^{p_0}_q &\vdash f^{p_0\rightarrow p_x}(v_1, \ldots, v_n) \Rightarrow () \quad (R'_{x})
\end{align*}
\]

The translation of isolated updating XRPC calls is depicted in the inference rule \( R'_{x} \) above. Like rule \( R_{x} \), this rule again provides proper isolation by keeping the database state \( \text{db}^{p_x}(t^{p_x}_q) \) constant throughout the query. The execution of a function \( f_u() \) at \( p_x \) causes a new pending update list to be created that becomes part of the collection \( \Delta^{p_x}_q \).
Obviously, atomically committing a distributed transaction requires a protocol like 2PC or one of its more advanced derivatives [135, 85]. We decided not to add 2PC to the XRPC network protocol, but rather rely on the recent industry standard Web Services Atomic Transaction [55] that provides exactly this feature for distributed web-service transactions. The Web Services Atomic Transaction [55] standard provides a fairly vanilla SOAP-based 2PC interface with e.g. Prepare() and Commit() functions. It is embedded in the Web Services Coordinator framework [54] that allows registering a collection of peers that participate in a distributed transaction, and subsequently run a transaction protocol on those peers (in this case WS-AtomicTransaction). Thus, in order to support updates with this isolation level, XRPC systems must implement support for these web service interfaces, and offer them over the same HTTP SOAP server that runs XRPC.

To implement proper 2PC, the Prepare() function brings \( q \) in the prepared state. It may raise an error, if a conflicting transaction has reached this state already. Else, it logs the union of the pending update lists (\( \Delta^p_q \)) to stable storage, ensuring \( q \) can commit later:

\[
\begin{align*}
\text{send}_{p_0 \rightarrow p_x}.\text{request}(q, \text{Prepare}); \\
\text{db}^{p_x}((t^p_{x, q}), \Delta^p_q) \vdash \log(\Delta^p_q) \Rightarrow r; \\
\text{send}_{p_x \rightarrow p_0}.\text{reply}(r) \\
\text{db}^{p_0}((t^{p_0}_{, q}), \Delta^p_0) \vdash \text{Prepare}^{p_0 \rightarrow p_x}() \Rightarrow r
\end{align*}
\]

Commit() carries through the updates, creating a new database state:

\[
\begin{align*}
\text{send}_{p_0 \rightarrow p_x}.\text{request}(q, \text{Commit}); \\
\text{db}^{p_x}((t^p_{x, q}), \Delta^p_q) \vdash \text{applyUpdates}(\Delta^p_q) \Rightarrow \text{db}^{p_x} \\
\text{db}^{p_0}((t^{p_0}_{, q}), \Delta^p_0) \vdash \text{Commit}^{p_0 \rightarrow p_x}() \Rightarrow \text{db}^{p_0}
\end{align*}
\]

More SOAP XRPC Extensions 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 participate in the transaction. Due to nested XRPC calls, it may not be aware of all peers and therefore we extended the SOAP XRPC protocol to piggyback a list of all unique participating peers in the response message.

Finally, the XQUF specifies that when the same node is updated twice in the same query, the order in which the different update actions on that node are applied is non-deterministic! This means that we can simply union all individual \( \Delta^p_q(i) \) pending update lists (one for each XRPC call handled in \( p_x \) for \( q \)) to get a full update list \( \Delta^p_q \) without worrying about preserving some proper order on the update actions. In Section 4.4, we define a deterministic update order for XQUF and devise a way to enforce it over XRPC using a small XRPC protocol extension, despite the out-of-order execution effects of Bulk RPC that will be observed at the end of Section 3.5.1.

3.5 Loop-lifted Implementation of XRPC

We have implemented XRPC in open-source MonetDB/XQuery, an efficient yet purely relational XDBMS [41]. It consists of the MonetDB relational database back-end, and the Pathfinder compiler [88], that translates XQuery into relational algebra, as front-end. The essence of the compilation technique employed by Pathfinder is loop-lifting [88], which translates XPath/XQuery expressions inside for-loops into single bulk relational query plans that process all iterations of the loop independently of each other. Loop-lifting makes MonetDB/XQuery inherently different (and often faster) than those XQuery interpreters that tend
Operator | Semantics
---|---
\(\sigma_a\) | select all rows with column \(a = \text{true}\)
\(\pi_{a_1:b_1,\ldots,a_n:b_n}\) | project columns \(b_1,\ldots,b_n\) and possibly rename columns \(b_j\) to \(a_i\) (no duplicate removal)
\(\delta\) | duplicate elimination
\(\cup\) | disjoint union
\(\Delta_{x=y}\) | equi-join
\(\rho_{b:⟨a_1,\ldots,a_n⟩/p}\) | row numbering (DENSE_RANK SQL:1999)
\[4\] | literal table

Table 3.2: Relational algebra generated by Pathfinder

to strictly follow the `for`-loop order syntactically suggested by a query. In case of Pathfinder, with its loop-lifted approach to XQuery translation, it was trivial to generate Bulk RPC requests for any XRPC call found in an XQuery. Hence, an XRPC call nested in a `for`-loop taken many times leads to only a single Bulk XRPC request/response, which invokes the function for all iterations of the loop in bulk. This optimisation dramatically reduces the number of request/response messages sent and thus the impact of the network latency on query performance.

The XRPC module contains an ultra-light HTTP daemon implementation [122] 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 serialisation mechanism in MonetDB/XQuery. The 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 serialised into XML, using the normal MonetDB/XQuery serialisation mechanism onto the network socket. As we re-used the shredding and serialisation functionality already in MonetDB/XQuery, as well as an off-the-shelf open source HTTP daemon [122], implementation was limited to a small parser extension, and stub code generation.

### 3.5.1 Relational XQuery and Loop-Lifting

The *Pathfinder* compiler [88] translates XPath/XQuery expressions into bulk query plans formulated in the vanilla relational algebra, depicted in Table 3.2. All operators are well-known, except perhaps the row numbering operator \(\rho\), which is similar to the SQL:1999 operator DENSE_RANK: \(\rho_{b:⟨a_1,\ldots,a_n⟩/p}(q)\) 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 of \(q\) by the columns \(a_1,\ldots,a_n\). Numbers ascend consecutively from 1 in each partition defined by the optional grouping column \(p\).
3.5. LOOP-LIFTED IMPLEMENTATION OF XRPC

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.

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 optimisable) 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: `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 `iter`s. Figure 3.2 shows an example query (Q3-5) and its loop-lifted relational representation. If we focus on the execution state in the innermost iteration body (marked as scope `s` 2) of (Q3-5), 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 (having the value of \$x\) concatenated with \$y\).

3.5.2 Bulk RPC

Our earlier example query (Q3-2) (repeated above) contains a function application inside a for-loop. Inside this loop, the variables \$dst\) and \$actor\) yield relational tables shown on the right. 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.

---

3The loop relation allows keeping track of empty sequence values, encoded by the absence of tuples in the expression representation.
SOAP XRPC 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 SOAP XRPC by allowing Bulk RPC, in which a single XRPC message to the destination peer requests it 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 straightforward to obtain the iter|pos|item table that represents an XDM result value for each iteration. Note that Bulk RPC fits well with the existing loop-lifted processing model of MonetDB/XQuery: without execute at, the local function translation mechanism already produced such an iter|pos|item table.

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

```xml
<xrpc:request xrpc:module="films" xrpc:method="filmsByActor" xrpc:arity="1" xrpc:location="http://x.example.org/film.xq" xrpc:updCall="false">
  <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 (Q3-3). We now have an inner for-loop with four iterations, but $dst$ takes on two different values, identifying peers “y.example.org” and “z.example.org”, in respectively the odd and even iterations. The general rule to translate a loop-lifted XRPC call is shown in Figure 3.4, and Figure 3.3 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 $\rho$ 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 3.3 performs two Bulk RPC calls. The first call processes both values of \(\$actor\) on “y.example.org”. Then a second call performs the same task on “z.example.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, MonetDB/XQuery improves performance by dispatching all Bulk RPC requests in parallel, which makes the exact order in which peers execute the query unpredictable. After all parallel results are united, the mapping of temporary \(iter_p\) numbers into \(iters\) guarantees that the final result is produced in the correct order.

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

```sql
import module namespace f="films" 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"} {f:filmsByActor($connery)},
execute at {"xrpc://y.example.org"} {f:filmsByActor($andrews)} )
```

Here, only the peer “y.example.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, and in the second RPC for actors with the surname Andrews. Note that the intuitive order suggested by the query would be to look for actors by the name Julie first, and those named Sean second.

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 3.4), 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 amount of network I/O, but also reducing the overhead of isolation.
Table 3.3: XRPC performance (msec): loop-lifted vs. one-at-a-time; no function cache vs. with function cache.

### 3.5.3 Performance Evaluation

We conducted some experiments to evaluate the performance of XRPC in MonetDB/XQuery. The test setup consisted of two 2GHz Athlon64 Linux machines connected on 1Gb/s Ethernet.

**Efficiency of Loop-Lifting**

To study the effect of loop-lifting, we defined an `echoVoid` function and called it over XRPC while varying the number of iterations:

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

While in MonetDB/XQuery loop-lifting of XRPC calls (i.e., Bulk RPC) is the default, we also implemented a one-at-a-time RPC mechanism for comparison. The left half of Table 3.3 (the “No Function Cache” column) 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, caused by (i) serialisation/deserialisation of the request/response messages, (ii) network communication cost and (iii) overhead of function call (1000 calls instead of 1 call). This is easily explained as the one-at-a-time RPC experiment involves performing 1000 times more synchronous RPCs.

**Throughput**

We also carried out bandwidth experiments (details omitted for space) that scaled request and response payloads. Here we observed throughput of 8MB/s (large requests) and 14 MB/s (large responses), which correspond roughly with resp. the document shredding and serialisation speed of MonetDB/XQuery [41]. Thus, like other SOAP-based messaging [84], XRPC data throughput on a fast local 1Gb network is CPU-bound rather than network-bound (though in a WAN it is likely to be the other way round).

**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.

The feature of prepared queries is well-known for RDBMS. It allows a parameterised query plan to be parsed and optimised 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...
3.6 Conclusion

In this chapter, we introduced XRPC, a minimal XQuery extension that enables distributed query execution with a focus on efficiency and interoperability. We first gave a formal definition of the syntax and the semantics of XRPC, including the semantics of distributed updates, that follow from the use of XQUF updating functions over XRPC. This includes the definition of two isolation levels for read-only and updating XRPC queries. Since interoperability is a major goal, the XRPC proposal also comprises a message protocol, which we chose to base on SOAP. Such a SOAP protocol has the additional advantage of seamless integration with web services and AJAX-based GUIs.

Our experiences in MonetDB/XQuery suggest that adding XRPC to existing XML database systems is easy; as shredding, serialisation and HTTP functionality are usually already present, the work is limited to a small parser extension and stub code generation. The SOAP XRPC protocol supports the concept of Bulk RPC, the execution of multiple function calls in a single message exchange. This amortises network and parsing latencies, and can make XRPC a quite efficient communication mechanism. 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. In the next section, we will show in our Saxon experiments that Bulk RPC enables set-oriented optimisations such that Bulk RPC execution of a selection function can be handled using a join strategy.

### Table 3.4: XRPC bandwidth for serialising XML documents

<table>
<thead>
<tr>
<th>XMark document size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth (MB/sec)</td>
</tr>
<tr>
<td>1MB</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Document Serving To examine the performance of XRPC in serialising XML documents, we used an HTTP client (wget) that retrieves a number of XMark documents of increasing size. Such requests are automatically handled as a call to fn:doc() over XRPC. Table 3.4 shows that XRPC achieves a bandwidth of 14MB/sec on average. Only for small documents (< 10MB) the bandwidth is lower, due to fixed start-up cost.

on small data sets can be accelerated ten-fold by this mechanism [41]. Note, that the function cache is not a query cache: queries are executed always on the latest data, and the performance improvement stems solely from the fact that query translation and optimisation is avoided.

This same function cache mechanism is used by the XRPC request handler. This means that in MonetDB/XQuery an XRPC request usually does not need query parsing and optimisation, just execution. The right half of Table 3.3 (the “With Function Cache” column) shows the impact of enabling the function cache: we see the processing time go down by 130ms (XQuery module translation time), improving both the single- and many-iteration Bulk RPC experiments. Thanks to the function cache, MonetDB/XQuery can achieve a minimum RPC latency of 3 msec – which is identical to that of commercial-strength software like .NET ([84, 130]).