XRPC: Distributed XQuery and Update Processing with Heterogeneous XQuery Engines

Ying Zhang and Peter Boncz
Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands
Y.Zhang@cwi.nl, P.Boncz@cwi.nl

ABSTRACT

We demonstrate XRPC, a minimal XQuery extension that enables distributed querying between heterogeneous XQuery engines. The XRPC language extension enhances the existing concept of XQuery functions with the Remote Procedure Call (RPC) paradigm. XRPC is orthogonal to all XQuery features, including the XQuery Update Facility (XQUF). Note that executing XQUF updating functions over XRPC leads to the phenomenon of distributed transactions. XRPC achieves heterogeneity by an open SOAP-based network protocol, that can be implemented by any engine, and an XRPC Wrapper that allows even XRPC-oblivious XQuery engines to handle XRPC requests efficiently. XRPC is fully implemented in the open-source MonetDB/XQuery engine, and is demonstrated here to co-operate with Saxon, Galax and X-Hive through the XRPC wrapper.

This demonstration will focus on the following features of XRPC: *(i)* glue-less interaction between AJAX style web-based applications with XQuery databases thanks to the SOAP-based nature of the XRPC network protocol, *(ii)* the efficiency of XRPC communication also for voluminous inter-server communication thanks to the *Bulk RPC* feature that optimizes network communication and exposes set-at-a-time opportunities to the underlying XQuery engines, *(iii)* the interoperability between *different* XQuery engines that can handle both distributed transactions (both read-only requests and updates) *(iv)* support and performance trade-offs of two different isolation levels for distributed transactions among different XQuery engines.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing, Distributed databases; H.2.3 [**Languages**]: Query Languages

General Terms

Design, Experimentation, Languages, Performance

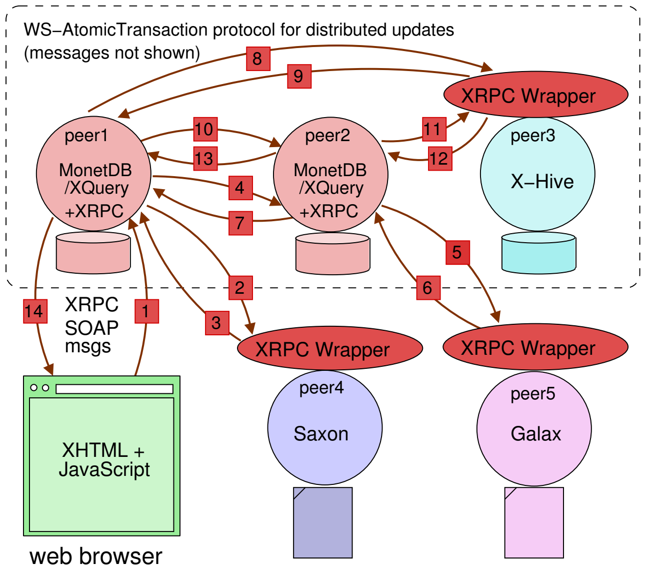


Figure 1: browser-initiated XRPC query, visiting peers 2,3 twice; updating peers 1-3 with atomic commit (dashed box area).

1. THE XRPC LANGUAGE EXTENSION

The XQuery 1.0 language [1] only provides a *data shipping* model for querying XML documents distributed over 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 recent W3C working draft of XQuery Update Facility (XQUF) [5] introduces a built-in function fn:put() for remote storage of XML documents, which again implies data shipping.

For distributed queries, a *function shipping* approach significantly enhances optimization possibilities. Therefore, we designed and implemented XRPC [8], a minimal XQuery extension that enables distributed XQuery processing with different XQuery engines, by naturally extending the existing concept of XQuery functions with the RPC paradigm. XRPC was carefully designed to be simple and easy to implement in existing engines on the one hand, yet on the other hand to fully respect the full semantics of the existing XQuery language, including such diverse feature areas as XML typing, but also XQUF *updating functions*. As the goal of XRPC is to allow different XQuery engines to co-operate, its network protocol is explicitly part of the proposal. This protocol is based on SOAP messages over HTTP, which also means that XRPC database servers can seamlessly integrate in Service Oriented Architectures (SOA). In our demo, we showcase "glue-less" access of web applications to MonetDB/XQuery, using XHTML pages that contain embedded JavaScript XRPC SOAP calls (i.e. AJAX). An example scenario is shown in Figure 1.

The XRPC Syntax. Remote function applications in XRPC take the XQuery syntax: execute at {Expr}{FunApp (ParamList)}, where *ExprSingle* is an XQuery xs:string expression that specifies the URI of the peer on which the function *FunApp* is to be executed. Hence, the destination of an XRPC call is not hard coded, instead, it can be calculated by any XQuery *ExprSingle*.

Examples. As a running example, we assume a set of XQuery database systems (peers) that each store a movie database in an XML document filmDB.xml with contents similar to:

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

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

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

With XRPC, we can execute this function on a remote peer, e.g. x.example.org, to get a sequence of films in which Sean Connery plays in the film database stored on the remote peer.

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

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

The XRPC URI scheme. We introduce a new URI scheme, accepted in the destination URI of execute at. The generic form of such a URI is: xrpc://<host>[:port][/[path]], where, xrpc:// indicates the network protocol, <host>[:port] indicates the remote peer, and [/[path]] is an optional local path at the remote peer. This URI scheme can be used to identify a remote peer that is running an XQuery engine and is able to handle XRPC requests. An xrpc:// URI can *also* be used inside a fn:doc() to retrieve XML documents from the XML database on a remote host, instead of from the file system of the host. That is, the query doc("xrpc://example.org/doc.xml") results in an HTTP request for getting the document "doc.xml" to the XRPC handler on host example.org. When xrpc:// URIs are present in a query, an XQuery optimizer could possibly split a query in subexpressions (as XQuery functions), and decide to execute some of these functions on the remote peer using XRPC [1].

[1]This demo uses distribution through explicit "execute at" statements; automatic XQuery distribution is ongoing work.

2. THE SOAP XRPC MESSAGE FORMAT

Our XRPC proposal also encompasses a SOAP [7] based network protocol, the SOAP XRPC protocol, which specifies the format of XRPC request and response messages. We use a literal SOAP format, with XML serialized according to the W3C standard, and full support for validation.

In an XRPC request message, the actual parameters of each single function call are enclosed by a call element. The SOAP XRPC protocol allows multiple call elements in a single request element so that multiple iterations of XRPC calls to the same function can be sent together (i.e. *Bulk RPC*, see below).

Since a parameter in XQuery can be a heterogeneously typed sequence, we enclose each parameter in a sequence element and annotate the type of each value. Atomic values are represented with atomic-value, and are annotated with their (simple) XML Schema Type in the xsi:type attribute, e.g. the sequence *(3.1, "abc")* would become:

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

XML nodes are passed by value in an element element.

```
<xrpc:sequence>
  <xrpc:element><name>The Rock</name></xrpc:element>
</xrpc:sequence>
```

Enclosing elements for other nodes types (e.g. document, text, attribute) are defined in the XML schema XRPC.xsd[2].

Bulk RPC. XRPC calls can be included in arbitrarily nested for-loops, thus, a naive implementation that makes a single RPC call at a time would easily generate a large amount of messages, resulting in high network latency and bandwidth usage. A crucial feature of the SOAP XRPC protocol is *bulk RPC*, that allows to compute multiple applications of the same function (with different parameters) in a *single* request/response network interaction, by including multiple call elements in a single XRPC request message. Moreover, bulk RPC also exposes set-at-at-time opportunities to the remote XQuery engine: a function that performs a selection can, when invoked with a bulk RPC, be handled with a join strategy.

We have implemented XRPC in the open-source relational XQuery DBMS MonetDB/XQuery [2] based on the *Pathfinder* compiler [6]. In our implementation, we generate bulk RPC requests for any XRPC call found in an XQuery. All iterations of applications of the same function in a for-loop are partitioned by their destinations. For each unique destination, a single bulk XRPC request is generated containing parameter values of all iterations for this destination, and then, all request messages are sent in *parallel* to their destinations. Each destination only sends back one response message containing results of all iterations.

3. THE XRPC WRAPPER

XRPC is not MonetDB/XQuery-specific: every XQuery engine that implements the SOAP XRPC protocol should be able to participate distributed XRPC queries. However,

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

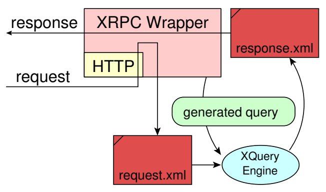
```
import module namespace f="funs" at "http://example.org/funs.xq";

declare namespace env="http://www.w3.org/2003/05/soap-envelope";
declare namespace xrpc="http://monetdb.cwi.nl/XQuery";

<env:Envelope ...>
  <env:Body>
    <xrpc:response xrpc:module="funs" xrpc:method="getPerson"> {
      for $call in doc("/tmp/requestXXX.xml")//xrpc:call
      let $param1 := n2s($call/xrpc:sequence[1])
      let $param2 := n2s($call/xrpc:sequence[2])
      return s2n(f:getPerson($param1, $param2))
    }</xrpc:response>
  </env:Body>
</env:Envelope>
```

Table 1: XQuery generated for the getPerson() XRPC request

cross-system distributed querying can be achieved even without XRPC integrated into an XQuery processing engine. What is needed is a simple *XRPC Wrapper* that can be run on top of an XQuery system. The XRPC Wrapper is a SOAP service handler, whose architecture is shown to the right. It stores an incoming XRPC request message in a temporary file, generates an XQuery for the request, and executes it using an XQuery processor. The generated query is crafted to compute the result of a bulk XRPC by calling the requested function on the parameters found in the message, and to generate the SOAP response message in XML using element construction. Note that this XRPC Wrapper only allows to *handle* calls using normal XRPC-incapable XQuery systems, it can not make outgoing XRPC calls from them.



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

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

Table 1 shows the query generated by XRPC Wrapper to handle a getPerson request. The XRPC protocol includes information about the arity of the function (as well as its return type), so it is easy to generate the right amount of *param* parameters in the call. The brunt of the work is done by two marshaling functions:

```
declare function n2s($n as node()) as item*
declare function s2n($seq as item*) as node()
```

The n2s() function converts an XRPC sequence node into an item sequence, where each item gets the right type. It traverses over all children of a sequence node using a series of if..then XQuery statements that select on the xsi:type attribute found in the atomic-value nodes. The s2n() function converts the function return value into a correct XRPC sequence node. It iterates over the input item sequence, and for each item it uses an XQuery typeswitch to generate the right SOAP node.

In this demo, we will show the XRPC Wrapper in action with different XQuery engines, where Galax, Saxon and X-Hive co-operate with XQuery queries initiated by MonetDB/XQuery instances.

4. DISTRIBUTED XRPC TRANSACTIONS

XRPC allows XQUF [5] expressions to be executed on remote peers, by means of XRPC calls to updating functions, thus providing distributed transaction functionality. For such distributed updating queries, XRPC provides two different isolation levels, *no isolation* and *repeatable reads*, to meet the needs of different kinds of applications. The latter level provides *repeatable reads* for all XRPC requests to the same peer made in a single query and uses a distributed 2-Phase Commit (2PC) protocol to ensure atomic commit. The semantics of these levels is formally defined in [8], including necessary extensions to the basic SOAP XRPC protocol to support them. Each XRPC query can specify the desired isolation level using the XQuery declare option feature to set xrpc:isolation to none or repeatable. Here, we briefly explain how repeatable reads with atomic commit is supported for updating XRPC queries on different XQuery engines.

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

To provide atomic distributed commit, we have chosen to use the SOAP-based 2PC industry standard WS-AtomicTransaction [4], which defines an API with functions such as Prepare() and Commit(). It is embedded in the WS-Coordinator framework [3] that allows to register a collection of peers that participate in a distributed transaction, and subsequently run a transaction protocol (in this case WS-AtomicTransaction) on those peers. In XRPC, peer p_q that starts the query q is the one that registers the participating peers at the WS Coordinator service and initiates the Prepare and Commit phases. For this registration task, it thus needs to know a full list of peers that participated in the transaction. Due to nested XRPC calls (i.e. remote functions calling in turn other remote functions), the query originator may not be aware of all peers involved and therefore we extended the SOAP XRPC protocol to piggyback a list of unique participating peers in their response messages.

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

4.1 Heterogeneous Distributed 2PC

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

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

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

1. When an XRPC request is received:
   - (a) check the query ID *id* carried by the request to see if a connection C_{id} for this query has already been created, and if not, create a new one, starting a new transaction (as mentioned, an error is generated for expired ID-s). Also, a new subdirectory D_{id} is created in the logging directory of the XRPC Wrapper;
   - (b) if the called function is a read-only function, execute it using the underlying XQuery engine and send its result back to the caller[3]. If the execution fails, add the query ID to the expired query log and remove D_{id}.
   - (c) otherwise, save the XRPC request message to the logging subdirectory D_{id} and send a response message to the caller to indicate success *without* actually executing the updating function (this is possible, as updating XQuery functions do not return a result). The rationale is that in order to provide repeatable reads, we must execute all updates together, at the end of the transaction; otherwise their effects would be visible for subsequent requests belonging to the same transaction.
2. When a Prepare request with ID *id* is received, then:
   - (a) if ID is expired, send Aborted to the coordinator;
   - (b) otherwise, if there are no request messages saved in the logging directory D_{id}, send ReadOnly to the coordinator, and then remove the logging directory D_{id}[4].
   - (c) otherwise, construct a *single* query containing *all* updating requests that have been saved so far (by using XQuery sequence construction). Execute the query in connection C_{id}, *without* committing the transaction yet. If this update query fails, add the query ID to the expired query log and remove D_{id}. Finally, send the decision Committed or Aborted to the coordinator (depending on the update success).
3. When a Rollback or a Commit request with ID *id* is received:
   - (a) If the request is Commit, log a "committing message" to D_{id}, and commit the transaction in C_{id}; The XRPC Wrapper should cease operation if committing in C_{id} fails, and then try to restart the underlying XQuery engine and/or itself, entering recovery mode.
   - (b) Add the query ID to the expired query log and remove D_{id}.

[3]Note that, to reduce possible communication time with the coordinators needed by the recover prodedure, each message should be logged before it is sent.

[4]Upon receipt of a ReadOnly notification, the coordinator knows that the participant votes to commit the transaction and had forgotten the transaction.

Thus, the XRPC Wrapper plays the game of declaring a distributed transaction committed, before actually committing in the underlying XQuery engine, relying on its own logging to do so at the global commit point.

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

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

5. DEMONSTRATION OUTLINE

This demonstration aims at showing all main features of XRPC:

- The demo GUI is an AJAX style web interface, created without any server-side logic, that itself demonstrates the use of XRPC from within XHTML pages using its JavaScript API.
- All basic XRPC functionality, including XRPC calls on multiple remote peers and nested XRPC calls, and calls to updating functions will be shown.
- We demonstrate efficiency improvement of *Bulk RPC* by comparing bulk XRPC calls with a one-at-a-time RPC mechanism, showing both the benefit in network performance, as well as the benefit of join-plans achievable by bulk RPC, over repeated selection plans caused by one-at-a-time RPC.
- To showcase the interoperability provided by XRPC, we run distributed XQueries on different XQuery engines (e.g. MonetDB/XQuery, Saxon, Galax and X-Hive). On MonetDB/XQuery, we run the integrated XRPC client and server. The other XQuery engines run with the XRPC Wrapper on top.
- We run queries under the *no isolation* and *repeatable reads* isolation levels on MonetDB/XQuery and X-Hive, showcasing the semantic (and performance) differences,

using a concurrent update load. For this purpose, we will construct non-trivial interaction patterns where peers must handle *multiple XRPC calls* originating from the same query.

- We perform distributed transactions formed by executing XQUF updating functions on a set of peers running MonetDB/XQuery. Here we will show two different consistency levels, one that performs updates immediately, at the cost of consistency, and one that provides repeatable reads and atomic commit.

- The demo then proceeds to run the same distributed transactions on a set of *heterogeneous* peers, i.e. some peers run MonetDB/XQuery and other peers run X-Hive.

As an added bonus – depending on development progress – some of the ongoing work on automatically distributing XQueries using XRPC may be shown.

6. REFERENCES

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

[2] 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.

[3] L. F. Cabrera, G. Copeland, M. Feingold, R. W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey. Web Services Coordination (WS-Coordination), August 2005. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[4] L. F. Cabrera, G. Copeland, M. Feingold, R. W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, T. Storey, and S. Thatte. Web Services Atomic Transaction (WS-AtomicTransaction), August 2005. ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf.

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

[6] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *VLDB*, 2004.

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

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