Batches: Unifying RPC, WS and Database access

William R. Cook
University of Texas at Austin
with
Eli Tilevich, Yang Jiao, Virginia Tech
Ali Ibrahim, Ben Wiedermann, UT Austin
Typical Approach to Distribution/DB

1. **Design** a programming language
   Finalize specification, then...

2. Implement distribution/queries as **library**
   - RPC library, stub generator
   - SQL library
   - Web Service library, wrapper generator
Typical Approach to Distribution/DB

1. **Design** a programming language
   Finalize specification, then...

2. Implement distribution/queries as **library**
   - RPC library, stub generator
   - SQL library
   - Web Service library, wrapper generator

3. **Rejected and ignored** by community
   - CORBA, DCOM, RMI are complete disaster
Using a Mail Service

```java
int limit = 500;
for (Message m : mailer.Messages)
    if (m.Size > limit) {
        print(m.Subject + " : " + m.Sender.Name);
        m.delete();
    }
```
Using a Mail Service

```java
int limit = 500;
for (Message m : mailer.Messages)
    if (m.Size > limit)
        print(m.Subject + " : " + m.Sender.Name);
        m.delete();
```

Works great if mailer is a local object, but is terrible if mailer is remote.
## Goals: Why not Have it All?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low latency</td>
<td><strong>One</strong> round trip</td>
</tr>
<tr>
<td>Stateless</td>
<td><strong>No</strong> Proxies</td>
</tr>
<tr>
<td>Platform independent</td>
<td><strong>No</strong> Serialization</td>
</tr>
<tr>
<td>Clean Server APIs</td>
<td><strong>No</strong> Data Transfer Object</td>
</tr>
<tr>
<td></td>
<td><strong>No</strong> Server Facade</td>
</tr>
<tr>
<td></td>
<td><strong>No</strong> Superclass/type</td>
</tr>
<tr>
<td></td>
<td>(e.g. java.rmi.Remote)</td>
</tr>
</tbody>
</table>

**A Note on Distributed Computing**
Using a Mail Service

```
int limit = 500;
for ( Message m : mailer.Messages )
    if ( m.Size > limit ) {
        print( m.Subject + " : " + m.Sender.Name);
        m.delete();
    }
```

Separate local and remote code!
```java
int limit = 500;

Remote<Mailer> connection = ....;

batch (Mailer mailer : connection) {
    for (Message m : mailer.Messages) {
        if (m.Size > limit) {
            print(m.Subject + " : " + m.Sender.Name);
            m.delete();
        }
    }
}

Remote<Mailer> connection = ....;
Forest in = new Forest("A", limit);
Forest mailer = connection.execute(script, in);
for (m : mailer.getIteration("m"))
    print(m.getString("B") + " : " + m.getString("C"));
```
Batch Pattern

Execution model: Batch Command Pattern
1. Client *sends script* to the server
   (Creates Remote Façade on the fly)
2. Server *executes* the script
3. Server *returns results in bulk* (name, size)
   (Creates Data Transfer Objects on the fly)
4. Client *runs the local code* (print statements)
Batch Script to SQL

```java
for (m : ROOT.Messages) {
    if (m.Size > In("A")) {
        out("B", m.Subject);
        out("C", m.Sender.Name);
        m.delete();
    }
}
```

**SQL:**

```sql
SELECT m.Subject as B, u.Name as C
FROM Message m INNER JOIN User u
ON m.Sender = User.ID
WHERE m.Size > ?
```

**DELETE:**

```sql
DELETE FROM Message
WHERE m.Size > ?
```

*Always constant number of queries*
Batch Script
(Subset of JavaScript)

s ::= <literal> variables, constants
    | e.x fields
    | e.m(e, ..., e) method call
    | e = e assignment
    | e ⊕ e | !e | e ? e : e primitive operators
    | if (e) { e } [else { e }] conditionals
    | for (x in e) { e } loops
    | var x = e; e binding
    | OUTPUT(label, e) outputs
    | INPUT(label) inputs
    | function(x) { e } functions
⊕ = + - * / % < <= == => > & & || ;
Batch Providers

Forwarder:

TCP Send A

TCP Receive B

Execute A

SQL Gen A

DB

Execution:
Batch Providers

**Forwarder:**
- TCP Send A
- TCP Receive B
- XML, JSON, ...
  (no fetish about transport)

**Execution:**
- Execute A
- SQL Gen A
- DB

568 LOC in Java

Execution:

- Execute
- SQL Gen A
- DB

TCP Send A

TCP Receive B

B

A

XML, JSON, ...

(no fetish about transport)
Combining Batch Providers

Traditional RPC:

Database Connection:
The Hard Part

Add Batch Statement to your favorite language
JavaScript, Python, ML, F#, Scala, etc
(More difficult in dynamic languages)

Batch (x) {...} → LangToScript → Script + Other → Partition(x)
I = Local1
O = Execute(Remote, I)
Local2(O)

Local1 → ScriptToLang → Local1
Remote (no other)
Local2

This is the part that needs to be written
Batch Summary

Client

Batch statement: compiles to Local/Remote/Local
Works in any language (e.g. Java, Python, JavaScript)
Completely cross-language and cross-platform

Server

Small engine to execute scripts
Call only public methods/fields (safe as RPC)
Stateless, no remote pointers (aka proxies)

Communication

*Forests* (trees) of primitive values (no serialization)
Efficient and portable
Batch = One Round Trip

Clean, simple performance model

Some batches would require more round trips

```java
batch (..) {
    if (AskUser("Delete " + msg.Subject + "?"))
        msg.delete();
}
```

Pattern of execution

**OK:** Local → Remote → Local

**Error:** Remote → Local → Remote

Can't just mark everything as a batch!
What about Object Serialization?

Batch only transfers primitive values, not objects
But they work with any object, not just *remotable* ones

Send a local set to the server?

```java
java.util.Set<String> local = ... ;
batch ( mail : server ) {
    mail.sendMessage( local, subject, body);
    // compiler error sending local to remote method
}
```
Serialization by Public Interfaces

```java
java.util.Set<String> local = ... ;
batch ( mail : server ) {
    service.Set recipients = mail.makeSet();
    for (String addr : local )
        recipients.add( addr );
    mail.sendMessage( recipients, subject, body);
}
```

Sends list of addresses with the batch
Constructors set on server and populates it
Works between different languages
Interprocedural Batches

Reusable serialization function

```java
@Batch
service.Set send(Mail server, local.Set<String> local) {
    service.Set remote = server.makeSet();
    for (String addr : local)
        remote.add(addr);
    return remote;
}
```

Main program

```java
batch ( mail : server ) {
    remote.Set recipients = send( localNames );
}
Exceptions

Server Exceptions
- Terminate the batch
- Return exception in forest
- Exception is raised in client at same point as on server

Client Exceptions
- Be careful!
- Batch has already been fully processed on server
- Client may terminate without handling all results locally
Transactions and Partial Failure

Batches are not necessarily transactional
But they do facilitate transactions
Server can execute transactionally

Batches reduce the chances for partial failure
Fewer round trips
Server operations are sent in groups
Order of Execution Preserved

All local and remote code runs in correct order

```java
batch ( remote : service ) {
    print( remote.updateA( local.getA() ));  // getA, print
    print( remote.updateB( local.getB() ));  // getB, print
}
```

Partitions to:

- `input.put(“X”, local.getA() );`  // getA
- `input.put(“Y”, local.getB() );`  // getB

.... execute updates on server

- `print( result.get(“A”) );`  // print
- `print( result.get(“B”) );`  // print

Compiler Error!
Lambdas

```java
for (InfoSchema db : connection)
    for (Group<Category, Product> g :
            db.Products.groupBy(Product. byCategory))
        print("Category={0}\t ProductCount={1}\t",
                g.Key.CategoryName,
                g.Items.count());

Fun<Product, Category> byCategory =
    new Fun<Product, Category>() {
        public Category apply(Product p) {
            return p.Category;
        }
    };
```
Available Now...

Batch Java
100% compatible with Java 1.7 (OpenJDK)
Transport: XML, JSON, easy to add more
Available now (alpha)

Batch statement as “for”
for (RootInterface r : serviceConenction) { ... }

Full SQL generation and ORM
Select/Insert/Delete/Update, aggregate, group, sorting