Implementation of CRDTs with $\delta$-mutators

Carlos Baquero

(Joint work with Paulo Almeida, Ali Shoker)

Presented at SyncFree M24 Meeting, September 2015
Why Deltas?

- State-based CRDTs
  - Simple middleware, Gossip
  - Complex/Big state, with UIDs and concurrency info
- Replicas evolve by mutations, inflations in a lattice
  - $X' = m(X)$
  - $X' = X \sqcup m(X)$
  - Updating other replicas $\Rightarrow$ shipping the big $X'$
- $\delta$—mutations
  - $\delta = m^\delta(X)$
  - $X' = X \sqcup \delta$
  - Updating other replicas $\Rightarrow$ shipping $\delta$, hoping $\delta \ll X'$
- $\delta$'s can be merged in transit. Usually applied in causal order
Why C++?

- Existing libraries: Python, Java, Erlang, Akka, (later Elixir)
- Strongly typed approach, no pointers and casts used.
- Good starting point from the Standard Template Library
- Efficiency...author already familiar with the language

GitHub

https://github.com/CBaquero/delta-enabled-crdts
Delta Enabled CRDTs

---

**delta-enabled-crdts**

Reference implementations of state-based CRDTs that offer deltas for all mutations.

## Datatypes

Current datatypes are:

- **GSet**: A grow only set
- **2PSet**: A two phase set that supports removing an element for ever
- **Pair**: A pair of CRDTs, first and second.
- **GCounter**: A grow only counter
- **PNCounter**: A counter supporting increment and decrement
- **LexCounter**: A counter supporting increment and decrement (Cassandra inspired)
- **DotKernel**: (Auxiliary datatype for building causal based datatypes)
- **CCounter**: A (causal) counter for map embedding (Optimization over Riak EMCounter)
- **AWORSet**: A add-wins optimized observed-remove set that allows adds and removes
- **RWORSet**: A remove-wins optimized observed-remove set that allows adds and removes
- **MVRRegister**: An optimized multi-value register (new unpublished datatype)
- **EWFlag**: Flag with enable/disable. Enable wins (Riak Flag inspired)
- **DWFFlag**: Flag with enable/disable. Disable wins (Riak Flag inspired)
- **ORMap**: Map of keys to CRDTs. (spec in common with the Riak Map)
- **RWLWWSet**: Last-writer-wins set with remove wins bias (SoundCloud inspired)
- **LWWReg**: Last-writer-wins register
A **join** template function was defined for taking **max** from ordered primitive types: char, int, float, bool, …

```cpp
int a=2, b=0;
cout << join(a,b) << endl;   // Output is 2

char x='a', y='b';
x=join(x,y);
cout << x << endl;   // Output is b
```
STL has a template pair composition. A point-wise join was defined

```cpp
pair<int, char> a(1, 'a'), b(0, 'x');
cout << join(a, b) << endl; // Output is pair (1,x)
```

While the point-wise version is default, a lexicographic join was also defined

```cpp
cout << lexjoin(a, b) << endl; // Output is pair (1,a)
```

Pairs can be nested and include non primitive types

```cpp
pair<int, pair<gset<int>, char>> triplet;
```
GSet, a simple anonymous CRDT
Family: GSet, TwoPSet

Classic use

```cpp
gset<string> a, b;

a.add("red"); b.add("blue");

a=b=join(a, b);

cout << a << endl; // GSet: ( blue red )
```

Obtaining deltas

```cpp
gset<string> d = a.add("green");

b.join(d);

cout << a << endl; // GSet: ( blue green red )
cout << b << endl; // GSet: ( blue green red )
```
Counters can be formed from any number type. Deltas can be anonymous but mutable instances must have a unique id.

```cpp
pncounter<long,char> x('a'), y('b'), d;

x.inc(4); x.dec();
d=y.dec();

x.join(d);

cout << x.read() << endl; // Output is 2
```

Default template types are **int** for counter and **string** for id

```cpp
pncounter<> z("syncfree"); z.inc();
cout << z.read() << endl; // Output is 1
```
Causal CRDTs, implemented by a kernel type with a universal join. All supported types are optimized (aka without tombstones)

**Kernel Stucture**

- **DotContext**: A version vector plus a sparse dot cloud
- **DataStore**: Mapping dots to chosen payload values

- Causal CRDTs hold an instance of a kernel.
- Used to add new dot to value pairs, remove pairs, join
- Causal information is grow-only and compacted when possible
- Possible to share a DotContext among instances, in maps
Classic use

```c++
aworset<float> x("uid-x"), y("uid-y"), d;

x.add(3.14); x.add(2.718); x.rmv(3.14);
d=y.add(3.14); // Concurrent add to above remove

x.join(d);

cout << x.read() << endl; // Output is ( 2.718 3.14 )
```

All kernel types support an observed reset

```c++
x.reset(); x.join(y);

cout << x.read() << endl; // Output is ( )
```
Collecting and merging deltas on site x

mvreg\textless string\textgreater x(\textquotedblleft uid\textendash x\textquoteright\textquotedblright), y(\textquotedblleft uid\textendash y\textquoteright\textquotedblright), d;

d=x.write(\textquotedblleft hello\textquoteright\textquotedblright); d.join(x.write(\textquotedblleft world\textquoteright\textquotedblright));

y.write(\textquotedblleft world\textquoteright\textquotedblright); y.write(\textquotedblleft hello\textquoteright\textquotedblright);

y.join(d);

cout \ll y.read() \ll endl; // Output is ( hello world )
Concurrent values related in an order can be reduced.
Total order example

```cpp
mvreg<int> x("uid-x"), y("uid-y");
x.write(0); y.write(3);
x.join(y); x.resolve();

cout << x.read() << endl;  // Output is (3)
x.write(1);  // Value can go up and down
```
Partial order example

```cpp
mvreg<pair<int, int>> x("uid-x"), y("uid-y"), z("uid-z");
x.write(pair<int, int>(0, 0));
y.write(pair<int, int>(1, 0));
z.write(pair<int, int>(0, 1));
x.join(y); x.join(z); x.resolve();

cout << x.read() << endl; // Output is ( (0,1) (1,0) )
```
Embedded map objects share a common causal context. Removing entries leads to a reset on the value.

One level maps

```
ormap< String , AWORSet< String >> mx("x"), my("y");
mx["paint"].add("blue");
my["paint"].add("red");
mx.join(my);
```

Nested maps

```
ormap< int , ormap< String , AWORSet< String >> > ma("alice");
ma[23]["color"].add("red at 23");
ma[44]["sound"].add("loud at 44");
```
C++, nice error messages at compile time
Questions?

Email: cbm@di.uminho.pt
Twitter: @xmal