Implementation of CRDTs with δ-mutators

Optimizing State-based CRDTs with Deltas and Composition

Carlos Baquero

HASLab, INESC Tec & Minho, Portugal

RICON 2015, San Francisco, November 2015

Universidade do Minho

Where I Work

Portugal @ Europe

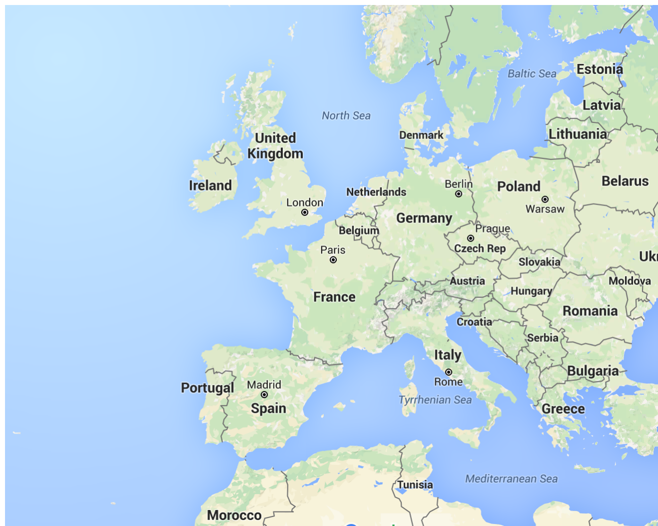


Image from Google Maps

Where I Work

HASLab @ Braga U.Minho Campus

Image from Google Maps

No Cats

But my kids pet is a kind of cat food

Why CRDTs?
Why Conflict-free Replicated Data Types?

- Synchronization requires connectivity and adds delays

Alexandre Duret-Lutz @ Flickr

Why CRDTs?
Why Conflict-free Replicated Data Types?

- Synchronization requires connectivity and adds delays
- Often a long queue can be replace by many small queues

$\Rightarrow$

Who cares if Alice buys bread, before or after, Bob buys a soda!

CC BY: Pin Add @ Flickr

Not Always

An Eiffel Tower with limited visitor capacity, needs synchronization.

But

Often Yes

Build as much as possible with CRDTs, synchronize the remaining.

Q.B. of Synchronization

Not Always

An Eiffel Tower with limited visitor capacity, needs synchronization.

But

Often Yes

Build as much as possible with CRDTs, synchronize the remaining.

Not Always

An Eiffel Tower with limited visitor capacity, needs synchronization.

But

Often Yes

Build as much as possible with CRDTs, synchronize the remaining.

Trading Time for Space

- Less synchronization can lead to more bookkeeping
- Single checkout queue:
  - $0 \to 1 \to 2 \to 3 \to \ldots$
- Multiple checkout queues (three):
  - $[0, 0, 0] \to [0, 1, 0] \to [0, 1, 1] \to [0, 1, 2] \to \ldots$
  - More precisely: There is no global sequence!

$$[\mathbf{0}, 0, 0] \longrightarrow \ldots$$

$$[0, \mathbf{0}, 0] \longrightarrow [0, \mathbf{1}, 0] \longrightarrow \ldots$$

$\sqcup$ (join)

$$[0, 0, \mathbf{0}] \longrightarrow [0, 0, \mathbf{1}] \longrightarrow [0, 1, \mathbf{1}] \longrightarrow [0, 1, \mathbf{2}] \longrightarrow \ldots$$

Trading Time for Space

- Less synchronization can lead to more bookkeeping
- Single checkout queue:
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots$
- Multiple checkout queues (three):
  - $[0,0,0] \rightarrow [0,1,0] \rightarrow [0,1,1] \rightarrow [0,1,2] \rightarrow \ldots$
  - More precisely: There is no global sequence!

$[\mathbf{0},0,0] \longrightarrow \ldots$

$[0,\mathbf{0},0] \longrightarrow [0,\mathbf{1},0] \longrightarrow \ldots$

$\sqcup$ (join)

$[0,0,\mathbf{0}] \longrightarrow [0,0,\mathbf{1}] \longrightarrow [0,1,\mathbf{1}] \longrightarrow [0,1,\mathbf{2}] \longrightarrow \ldots$

Trading Time for Space

- Less synchronization can lead to more bookkeeping
- Single checkout queue:
  - $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots$
- Multiple checkout queues (three):
  - $[0,0,0] \rightarrow [0,1,0] \rightarrow [0,1,1] \rightarrow [0,1,2] \rightarrow \ldots$
  - More precisely: There is no global sequence!

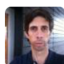
$$[\mathbf{0},0,0] \longrightarrow \ldots$$

$$[0,\mathbf{0},0] \longrightarrow [0,\mathbf{1},0] \longrightarrow \ldots$$

$$[0,0,\mathbf{0}] \longrightarrow [0,0,\mathbf{1}] \longrightarrow [0,1,\mathbf{1}] \longrightarrow [0,1,\mathbf{2}] \longrightarrow \ldots$$

($[0,\mathbf{1},0]$ and $[0,0,\mathbf{1}]$ combine into $[0,1,\mathbf{1}]$ via $\sqcup$ (join))

Join derives a Least Upper Bound



Carlos Baquero @xmal · 20 Jun 2014

#seancribbsholdingthings Least Upper Bounds

FAVORITES

2

Why δs for state-based CRDTs?

(Joint work with Ali Shoker and Paulo Almeida)

- State-based CRDTs
  - ⊞ Simple middleware, Gossip
  - ⊟ Complex/Big state, with UIDs and concurrency info
- Replicas evolve by mutations, inflations in a lattice
  - $X' = m(X)$ e.g. GSet $\{a, b, c\} = add(b, \{a, c\})$
  - Updating other replicas $\Rightarrow$ shipping the big X'
- δ−mutations
  - $\delta = m^{\delta}(X)$ e.g. $\{b\} = add^{\delta}(b, \{a, c\})$
  - $X' = X \sqcup \delta$ $\{a, b, c\} = \{a, c\} \sqcup \{b\}$
  - Updating other replicas $\Rightarrow$ shipping δ, hoping $\delta \ll X'$
- δs can be merged in transit. Usually applied in causal order

Why δs for state-based CRDTs?
(Joint work with Ali Shoker and Paulo Almeida)

- State-based CRDTs
  - ⊞ Simple middleware, Gossip
  - ⊟ Complex/Big state, with UIDs and concurrency info
- Replicas evolve by mutations, inflations in a lattice
  - $X' = m(X)$ e.g. GSet $\{a, b, c\} = add(b, \{a, c\})$
  - Updating other replicas $\Rightarrow$ shipping the big X'
- $\delta-$mutations
  - $\delta = m^{\delta}(X)$ e.g. $\{b\} = add^{\delta}(b, \{a, c\})$
  - $X' = X \sqcup \delta$ $\{a, b, c\} = \{a, c\} \sqcup \{b\}$
  - Updating other replicas $\Rightarrow$ shipping δ, hoping $\delta \ll X'$
- δs can be merged in transit. Usually applied in causal order

Why δs for state-based CRDTs?

(Joint work with Ali Shoker and Paulo Almeida)

- State-based CRDTs
  - ⊞ Simple middleware, Gossip
  - ⊟ Complex/Big state, with UIDs and concurrency info
- Replicas evolve by mutations, inflations in a lattice
  - $X' = m(X)$ e.g. GSet $\{a, b, c\} = add(b, \{a, c\})$
  - Updating other replicas $\Rightarrow$ shipping the big X'
- $\delta-$mutations
  - $\delta = m^{\delta}(X)$ e.g. $\{b\} = add^{\delta}(b, \{a, c\})$
  - $X' = X \sqcup \delta$ $\{a, b, c\} = \{a, c\} \sqcup \{b\}$
  - Updating other replicas $\Rightarrow$ shipping δ, hoping $\delta \ll X'$
- δs can be merged in transit. Usually applied in causal order

Math, lots of Math

$$
\begin{aligned}
\mathsf{CCounter} &= \mathsf{DotKernel}\langle\mathbb{N}\rangle \\
\mathsf{val}_i(j,(s,c)) &= \begin{cases} v & \text{if } (j,n,v) \in s \\ 0 & \text{otherwise} \end{cases} \\
&\quad \text{with } n = \mathsf{max}(\{k \mid (j,k) \in c\} \cup \{0\}) \\
\mathsf{inc}_i^\delta((s,c)) &= (\bot,(i,n)) \sqcup \mathsf{add}_i^\delta(\mathsf{val}_i(i,(s,c)) + 1,(s,c)) \\
&\quad \text{with } n = \mathsf{maximal}_i(c) \\
\mathsf{dec}_i^\delta((s,c)) &= (\bot,(i,n)) \sqcup \mathsf{add}_i^\delta(\mathsf{val}_i(i,(s,c)) - 1,(s,c)) \\
&\quad \text{with } n = \mathsf{maximal}_i(c) \\
\mathsf{value}_i(dk) &= \sum_{j \in \mathbb{I}} \mathsf{val}_i(j,dk) \\
\mathsf{reset}_i^\delta(dk) &= \mathsf{reset}_i^\delta(dk)
\end{aligned}
$$

Fig. 16: δ-CRDT Causal Counter, replica i.

Reference library of data types

GitHub

https://github.com/CBaquero/delta-enabled-crdts

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

Reference library of data types

GitHub

https://github.com/CBaquero/delta-enabled-crdts

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

Delta Enabled CRDTs

16 data types and growing

README.md

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
- MVRegister: An optimized multi-value register (new unpublished datatype)
- EWFlag: Flag with enable/disable. Enable wins (Riak Flag inspired)
- DWFlag: 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

Primitive Types

A **join** template function was defined for taking **max** from ordered primitive types: char, int, float, bool, . . .

```
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
```

Pair Composition

STL has a template pair composition. A point-wise join was defined

```
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

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

Pairs can be nested and include non primitive types

```
pair<int,pair<gset<int>,char>> triplet;
```

GSet, a simple anonymous CRDT

Family: GSet, TwoPSet

Classic use

```
gset<string> a,b;

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

a=b=join(a,b);

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

Obtaining deltas

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

b.join(d);

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

GSet, a simple anonymous CRDT

Family: GSet, TwoPSet

Accumulating deltas allows batching of mutations

```
gset<string> a,b;

// ... replica "a" sees many adds ...

b.join(a);

auto acum = a.add("one");
acum.join(a.add("two"));
acum.join(a.add("three"));

b.join(acum);
```

Transmit information at a lower rate than replica mutation

PNCounter, a simple identified CRDT

Family: GCounter, PNCounter, LexCounter

Counters can be formed from any number type. Deltas can be anonymous but mutable instances must have a unique id.

```
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

```
pncounter<> z("syncfree"); z.inc();
cout << z.read() << endl; // Output is 1
```

Dot Kernel for Causal CRDTs

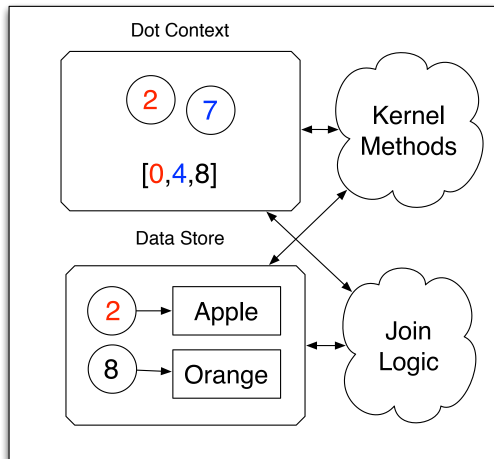
Family: CCounter, AWSet, RWSet, MVRegister, EWFlag, DWFlag

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

Dot Kernel



Looking at the causal Dot Contexts compare Data Stores. Check the dot key for each mapping to payloads, under the rules:

- If we both have the mapping, keep the mapping.
- If only me has a mapping
  - If the other didn't see it before, keep it
  - If the other saw it before, drop it
- Repeat last step vice-versa

Universal Causal Join

$$(s, c) \sqcup (s', c') = ((s \cap s') \cup \{(i, n, e) \in s \mid (i, n) \notin c'\} \cup \{(i, n, e) \in s' \mid (i, n) \notin c\}, c \sqcup c')$$

Looking at the causal Dot Contexts compare Data Stores. Check the dot key for each mapping to payloads, under the rules:

- If we both have the mapping, keep the mapping.
- If only me has a mapping
  - If the other didn't see it before, keep it
  - If the other saw it before, drop it
- Repeat last step vice-versa

Universal Causal Join

$$(s, c) \sqcup (s', c') = ((s \cap s') \cup \{(i, n, e) \in s \mid (i, n) \notin c'\} \cup \{(i, n, e) \in s' \mid (i, n) \notin c\}, c \sqcup c')$$

Dot Kernel for Causal CRDTs

Family: CCounter, AWSet, RWSet, MVRegister, EWFlag, DWFlag

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
- Data Store can grow and shrink
- Possible to share a DotContext among instances, in maps

Add-Wins Observed-Remove Set

Family: CCounter, AWSet, RWSet, MVRegister, EWFlag, DWFlag

Classic use

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

float pi=3.14, e=2.718;

x.add(pi); x.add(e); x.rmv(pi);
d=y.add(pi); // 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*

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

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

(Optimized) Multi-Value Register

Family: CCounter, AWSet, RWSet, MVRegister, EWFlag, DWFlag

Collecting and merging deltas on site x

```
mvreg<string> x("uid-x"),y("uid-y"),d;

d=x.write("hello"); d.join(x.write("world"));

y.write("world"); y.write("hello");

y.join(d);

cout << y.read() << endl; // Output is ( hello world )
```

(Optimized) Multi-Value Register

Reducing siblings, (joint design with Marek, Nuno, Annette, Marc)

Concurrent values related in an order can be reduced.

Total order example

```
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
```

(Optimized) Multi-Value Register

Reducing siblings, (joint design with Marek, Nuno, Annette, Marc)

Partial order example

```
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) )
```

Observed-Remove Map (joint design Paulo, Russell, . . .)

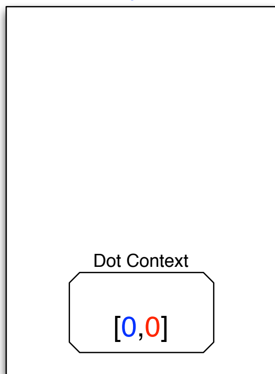
Embeds: CCounter, AWSet, RWSet, MVRegister, EWFlag, DWFlag, ORMap

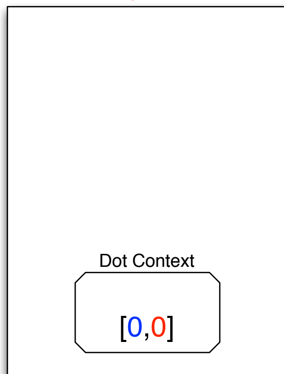
Embedded map objects share a common causal context

```
ormap<string,aworset<string>> mx("x"),my("y");
```

Observed-Remove Map

String mapping to an AWSet





Observed-Remove Map

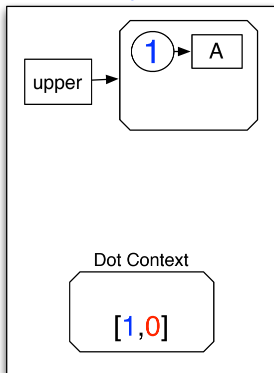
String mapping to an AWSet

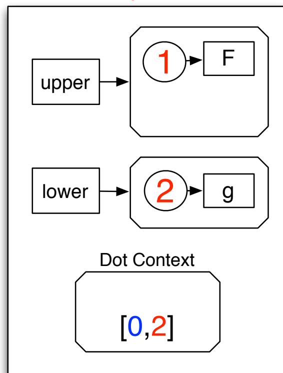
Embedded map objects share a common causal context

```
ormap<string,aworset<string>> mx("x"),my("y");

mx["upper"].add("A");
my["upper"].add("F");
my["lower"].add("g");
```

Observed-Remove Map

String mapping to an AWSet






```
mx["upper"].add("A");
```

```
my["upper"].add("F");
my["lower"].add("g");
```

Observed-Remove Map

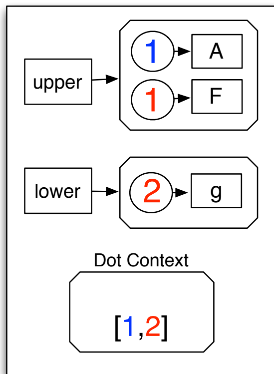
String mapping to an AWSet

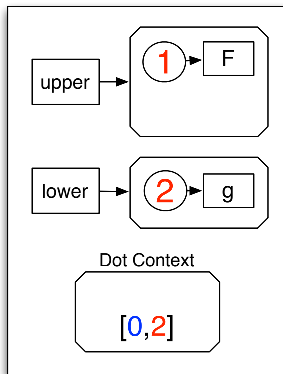
Embedded map objects share a common causal context

```
ormap<string,aworset<string>> mx("x"),my("y");

mx["upper"].add("A");
my["upper"].add("F");
my["lower"].add("g");

mx.join(my);
```

Observed-Remove Map

String mapping to an AWSet





mx.join(my);

Observed-Remove Map

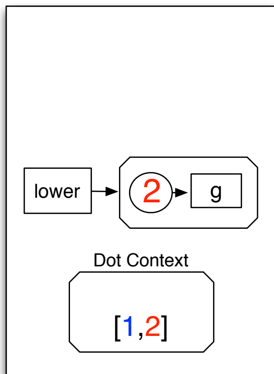
String mapping to an AWSet

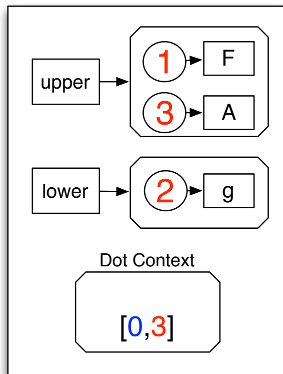
Removing/erasing entries leads to a reset on the value

```
ormap<string,aworset<string>> mx("x"),my("y");

mx["upper"].add("A");
my["upper"].add("F");
my["lower"].add("g");


mx.join(my);

mx.erase("upper");
my["upper"].add("A");
```

Observed-Remove Map

String mapping to an AWSet





mx.erase("upper");

my["upper"].add("A");

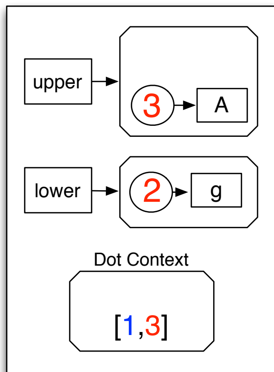
Reseted state does not come back

```
ormap<string,aworset<string>> mx("x"),my("y");

mx["upper"].add("A");
my["upper"].add("F");
my["lower"].add("g");


mx.join(my);

mx.erase("upper");
my["upper"].add("A");

mx.join(my);
```

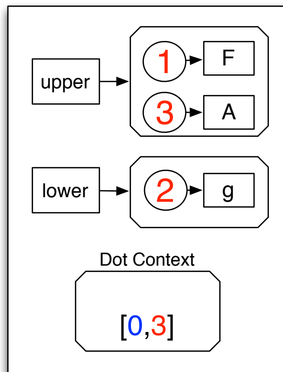
Observed-Remove Map

String mapping to an AWSet

Map X



Map Y



mx.join(my);

Observed-Remove Map

Integer mapping to String mapping to an AWSet

Maps can be nested

```
ormap<int,ormap<string,aworset<string>>> ma("alice");

ma[23]["color"].add("red at 23");
ma[44]["sound"].add("loud at 44");
```

Different key types at different nesting levels

C++, nice error messages at compile time

```
delta-enabled-crdts — bash — 80×24
MacBook-Pro:delta-enabled-crdts cbm$ make
g++ -std=c++11 -ferror-limit=2 delta-crdts.cc delta-tests.cc -o delta-tests
In file included from delta-tests.cc:38:
./delta-crdts.cc:47:9: error: no member named 'join' in
      'std::__1::basic_string<char>'
    res.join(r);
    ~~~ ^
./delta-crdts.cc:68:52: note: in instantiation of function template
      specialization 'join_selector<false>::join<std::__1::basic_string<char> >'
      requested here
  return join_selector< is_arithmetic<T>::value >::join(l,r);
                                                   ^
./delta-crdts.cc:967:15: note: in instantiation of function template
      specialization 'join<std::__1::basic_string<char> >' requested here
            ::join(dsa.second,dsb.second) == dsb.second ) // < based on join
              ^
delta-tests.cc:805:3: note: in instantiation of member function
      'mvreg<std::__1::basic_string<char>, std::__1::basic_string<char>
      >::resolve' requested here
y.resolve();
  ^
1 error generated.
make: *** [delta-tests] Error 1
MacBook-Pro:delta-enabled-crdts cbm$
```

Future Work

- Add more data types: Sequences, Graphs, ...
- MSc thesis starting
  - Choose serialization format
  - Spread deltas over gossip. E.g. Plumtree
  - Recover from failed nodes and link breaks
- Compare delta state based, with causal shipping of operations
- (Hypothesis: Less synchronous networks favor state based)
- Replace use of in-memory STL Sets, and Maps, by a local DB.

Questions?

Carlos Baquero
HASLab, Universidade do Minho and INESC Tec
Portugal

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