Programming models for eventual consistency

Peter Zeller

TU Kaiserslautern, AG Softech

July 2014
What is a programming model?

- How to specify the interface to the outside world?
- How to write a correct implementation?
- How to reason about the correctness of an application?
- What interfaces and which guarantees are provided by the infrastructure.
1. Introduction: Replication, System topologies, Infrastructure, CRDTs
2. Programming models:
   - Cloud types
   - SwiftCloud
   - Riak
3. Correctness
Replication: Storing the same data at multiple locations

Motivation:
- High availability
- High throughput
- Low delay, geo-replication
- Systems, which are not always connected
- Cheap hardware
- ...
System topologies

Clients:

Server:
System topologies

Clients:

DC1  DC2  DC3

partial replicas
System topologies
System topologies

Clients:

Application Server:

Database

DC1

DC2

DC3
Where are the borders of our application?
Where is state stored (persistently)?
Which connections are possible?
Where do we have concurrency?
...
Data store infrastructure:

Distinguishing points:

- Transactions
- Atomicity
- Isolation
- Failure model
- Causality (How exactly is causality defined, how is it tracked)
- Extending the database (Define own datatypes)
- Which parts are active, which parts just respond to requests?
- Level of concurrency
- ...
Simple example: Replicated integer variable $x$

replica A:

$x=2$ \arrow{update} $x=4$ \arrow{update} $x=?$

replica B:

$x=2$ \arrow{update} $x=3$ \arrow{merge}
Replicated counter

replica A:

replica B:

merge
Replicated multi-value register

replica A:  
\[ x=2 \rightarrow x=4 \rightarrow x=\{3,4\} \]

replica B:  
\[ x=2 \rightarrow x=3 \]

merge
Replicated data types

- Data types, for example
  - Counters
  - Registers
  - Sets
  - Maps
  - Graphs
  - ...
- Replicated on several nodes
- Integrated consistency

---

1 Shapiro, N. Preguiça, Baquero, and Zawirski, *A comprehensive study of Convergent and Commutative Replicated Data Types*. 
“Cloud types”\textsuperscript{23} programming model - overview

- Central database + clients with full replication
- Single-threaded clients with implicit transactions
  - Everything between two yield statements is considered as a transaction
- Explicit flush operation to get latest state
- Cloud types for handling concurrent updates to data

\textsuperscript{2}Burckhardt, Fähndrich, Leijen, and Wood, “Cloud Types for Eventual Consistency”.
\textsuperscript{3}Burckhardt, Leijen, and Fahndrich, \textit{Cloud Types: Robust Abstractions for Replicated Shared State}. 
Global log of update transactions (GLUT)

Clients see some **prefix** of GLUT and own updates

Merging with GLUT = appending to GLUT
“Cloud types” programming model - consistency model

- Global log of update transactions (GLUT)
- Clients see some **prefix** of GLUT and own updates
- Merging with GLUT = appending to GLUT
“Cloud types” programming model - cloud types

- Similar to CRDTs but more flexible
  - Because operations are totally ordered in the GLUT updates can be non-commutative

- Types:
  - Cloud integer
    - get, set, add
  - Cloud string
    - get, set, setIfEmpty
  - Cloud table
    - Key→Value store with explicit creation and deletion
  - Cloud index
    - Key→Value store with default values for all keys
  - ...

- Not possible to define own types
SwiftCloud⁴ programming model - consistency model

---

⁴Zawirski, Bierniusa, Balegas, Duarte, Baquero, Shapiro, and N. M. Preguica, “SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine”.

---
SwiftCloud\textsuperscript{4} programming model - consistency model

\begin{center}
\begin{tikzpicture}[node distance=2cm,auto,>=latex,]
\node (DC1) [draw,shape=circle,fill=green] {DC1};
\node (DC2) [draw,shape=circle,fill=blue] at (3,0) {DC2};
\node (Green_client) [draw,shape=circle,fill=green] at (-4,1) {Green client:};
\node (Yellow_client) [draw,shape=circle,fill=yellow] at (-4,-1) {Yellow client:};
\node (Blue_client) [draw,shape=circle,fill=cyan] at (0,0) {Blue client:};
\draw [->,thick] (DC1) -- (Green_client);
\draw [->,thick] (DC1) -- (Yellow_client);
\draw [->,thick] (DC1) -- (Blue_client);
\draw [->,thick,dashed] (Green_client) -- (DC2);
\draw [->,thick,dashed] (Yellow_client) -- (DC2);
\draw [->,thick,dashed] (Blue_client) -- (DC2);
\draw [->,thick,dashed] (DC2) -- (Green_client);
\draw [->,thick,dashed] (DC2) -- (Yellow_client);
\draw [->,thick,dashed] (DC2) -- (Blue_client);
\end{tikzpicture}
\end{center}

\begin{itemize}
\item DC1
\item DC2
\end{itemize}

\textsuperscript{4}Zawirski, Bieniusa, Balegas, Duarte, Baquero, Shapiro, and N. M. Preguiça, “SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine”.

15
SwiftCloud programming model - consistency model

- Transactions see some causally consistent snapshot + local updates
  - Monotonic: Later snapshot → later state
- Clients execute transactions sequentially
- No total order on transactions, but parallel transactions always commute
  - Commutativity ensured by using CRDTs
- Clients only have a cache, no full replication
Riak\textsuperscript{5} - consistency model

\textsuperscript{5}http://basho.com/riak/
Riak - consistency model

- No cross-object consistency
- No transactions, just bundling of several updates on one object
- Causality independent of program order
- Parallel updates handled by CRDTs
Example

Task: Store the maximum score a player has reached
Task: Store the maximum score a player has reached
Sequential solution:

```plaintext
function updateScore(player, newScore)
  if (score[player] < newScore)
    score[player] := newScore
```
function updateScore(player, newScore)
    if (score[player] < newScore)
        score[player] := newScore

Just taking the sequential solution does not work:
function updateScore(player, newScore)
    if (score[player] < newScore)
        score[player] := newScore

Just taking the sequential solution does not work:
1. Initially score[p] = 1 (everywhere)
Example - Cloud types

```plaintext
function updateScore(player, newScore)
    if (score[player] < newScore)
        score[player] := newScore
```

Just taking the sequential solution does not work:
1. Initially score[p] = 1 (everywhere)
2. client1.updateScore(p, 3)
   → client1.score[p] = 3
Example - Cloud types

function updateScore(player, newScore)
    if (score[player] < newScore)
        score[player] := newScore

Just taking the sequential solution does not work:

1. Initially score[p] = 1 (everywhere)
2. client1.updateScore(p, 3)
   → client1.score[p] = 3
3. client2.updateScore(p, 4)
   → client2.score[p] = 4
Example - Cloud types

```java
function updateScore(player, newScore)
    if (score[player] < newScore)
        score[player] := newScore
```

Just taking the sequential solution does not work:

1. Initially score[p] = 1 (everywhere)
2. client1.updateScore(p, 3)
   → client1.score[p] = 3
3. client2.updateScore(p, 4)
   → client2.score[p] = 4
4. client2 yield
   → global.score[p] = 4
Example - Cloud types

```python
function updateScore(player, newScore)
    if (score[player] < newScore)
        score[player] := newScore
```

Just taking the sequential solution does not work:

1. Initially score[p] = 1 (everywhere)
2. client1.updateScore(p, 3)
   → client1.score[p] = 3
3. client2.updateScore(p, 4)
   → client2.score[p] = 4
4. client2 yield
   → global.score[p] = 4
5. client1 yield
   → global.score[p] = 3
“The anti-pattern here is that updates to a cloud value must make sense even if some ‘earlier’ updates are not yet visible to the local client”\textsuperscript{6}

\textsuperscript{6}Burckhardt, Leijen, and Fahndrich, \textit{Cloud Types: Robust Abstractions for Replicated Shared State}. 
Example - Cloud types

Possible solution: Store operation in a log (cloud table)

```
function updateScore(player, newScore)
    if (score[player] < newScore)
        scoreLog.newEntry(player, newScore)
```

- When reading: calculate maximum (and purge log)
- Using a log is a general pattern
  - No lost updates, no conflicts
  - Idempotence and commutativity
  - Fault tolerant
- Disadvantages:
  - Much work for clients
  - Efficiency
SwiftCloud already includes a CRDT for keeping track of maximum values:

```swift
function updateScore(player, newScore)
    transaction
        MaxCRDT scoreCRDT = score[player]
        scoreCRDT.set(newScore)
```
SwiftCloud already includes a CRDT for keeping track of maximum values:

```swift
func updateScore(player: Int, newScore: Int) {
    transaction {
        let scoreCRDT = score[player]
        scoreCRDT.set(newScore)
    }
}
```

General pattern:
- Find right CRDT for the problem
- Write new CRDT no suitable type exists
Example - Riak

Riak does not have a MaxCRDT, but Multi-Value-Registers can be used as a fall-back:

```haskell
function updateScore(player, newScore)
    oldScore, context := getScore(player)
    if (oldScore < newScore)
        setScore(context, player, newScore)
```

---

---

7 DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels, “Dynamo: Amazon’s Highly Available Key-value Store”. 
24
Example - Riak

Riak does not have a MaxCRDT, but Multi-Value-Registers can be used as a fall-back:

```go
function updateScore(player, newScore)
    oldScore, context := getScore(player)
    if (oldScore < newScore)
        setScore(context, player, newScore)
```

General pattern:
- Use Multi-Value-Register for mutable state
- Merge values in application when reading
- Write back merged value

---

7DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels, “Dynamo: Amazon’s Highly Available Key-value Store”. 24
Example - Riak

Riak does not have a MaxCRDT, but Multi-Value-Registers can be used as a fall-back:

```plaintext
function updateScore(player, newScore)
    oldScore, context := getScore(player)
    if (oldScore < newScore)
        setScore(context, player, newScore)
```

General pattern:
- Use Multi-Value-Register for mutable state
- Merge values in application when reading
- Write back merged value

Causality tracking:
- Explicit context value
- Reading a value yields a context
- Context can be given in write operations

---

7 DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels, “Dynamo: Amazon’s Highly Available Key-value Store”.
Fault tolerance

```python
function updateScore(player, newScore)
    updatePlayerScore(player, newScore)
    updateLeaderBoard(player, newScore)
```

Problem:
- Two updates, second might fail
- Process might crash
- Database operation might timeout

Solutions:
- Use a transaction
- Use a queue + idempotent operations
- Repeat until successful

---

Correctness

```python
function tryJoinGame(player, minScore)
    if score[player] >= minScore
        assert global.score[player] >= minScore
        joinGame(player)
```

Is this assertion always true?

- Score grows monotonically
- Condition is monotonic
Correctness

```python
function tryJoinGame(player, minScore)
    if score[player] >= minScore
        assert global.score[player] >= minScore
        joinGame(player)
    else
        assert global.score[player] <= minScore
        print("You are not good enough for this game.")
```

Is this assertion always true?

- Could read old value of score
- Might print a wrong message
Correctness

Monotonicity as a programming model\(^9\):

- CALM principle (consistency and logical monotonicity)
- use monotonicity as much as possible
- use synchronization otherwise
- prototype implementation “Bud” as a domain specific language embedded in Ruby
  - Programming with tables, lattices, streams and monotonic operations on them
  - Static program analysis finds places which might need synchronization

---

\(^9\)Conway, Marczak, Alvaro, Hellerstein, and Maier, “Logic and lattices for distributed programming”. 
Correctness - Reservations

```
function tryBuyItem(item)
    if localMoney >= item.cost
        buyItem(item)
    else if globalMoney >= item.cost
        tryToReserveMoneyLocally()
        retry
    else
        print("Insufficient money")
```

- Split resource
- Replicas own parts of a resource and have the rights to use it
- Needs some protocols to transfer rights
- Best case: local check sufficient, no synchronization necessary
- Worst case: fall back to synchronization
Correctness - Reservations

References


Other patterns

Avoid execution order dependencies

- Implicit object creation
  - Cloud index vs cloud array
- Object deletion by tombstones
- Use unordered types when possible
  - set instead of list data type
- Generate unique identifiers locally
- Repair invariants when reading
  - Example: graph
Specification of applications

- State based specifications (e.g. pre- and post-conditions)
  - Hard to base specification on states, because there are different states at different replicas
  - Talking about the “state after all updates are merged” not always useful
  - Usable when state changes monotonically

- Equivalence to sequential execution
  - Not always possible (e.g. Multi-Value Register)

- Principle of permutation equivalence\textsuperscript{11}
  - If all possible sequential executions of the updates yield the same state, then the concurrent execution should yield the same state.
  - Other cases?

- Axiomatic specification\textsuperscript{12}
  - Specification is a predicate on the visible events, the causal order between events, and the arbitration order between events.
  - Expressive, powerful, but difficult to use

\textsuperscript{11}Bieniusa, Zawirski, N. M. Preguiça, Shapiro, Baquero, Balegas, and Duarte, “Brief Announcement: Semantics of Eventually Consistent Replicated Sets”.

\textsuperscript{12}Burckhardt, Gotsman, and Yang, \textit{Understanding Eventual Consistency}.\textsuperscript{32}
Some programming models accepted for most models:
- Causality
- Replicated Data Types
- Monotonicity and idempotence

In discussion / it depends:
- Transactions
- Monotonic / dataflow programming
- Reservations

Still lacking:
- Methods for specification and reasoning about correctness
- Advanced tools which simplify writing applications

Burckhardt, Sebastian, Alexey Gotsman, and Hongseok Yang. *Understanding Eventual Consistency*. Tech. rep. MSR-TR-2013-39. This document is work in progress. Feel free to cite, but note that we will update the contents without warning (the first page contains a timestamp), and that we are likely going to publish the content in some future venue, at which point we will update this paragraph. Mar. 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=189249.


References III


References V


