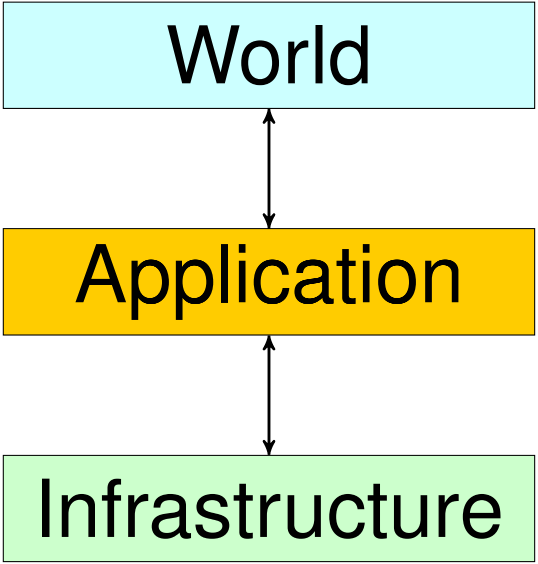
Programming models for eventual consistency

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.

Outline

1. Introduction: Replication, System topologies, Infrastructure, CRDTs
2. Programming models:
   - Cloud types
   - SwiftCloud
   - Riak
3. Correctness

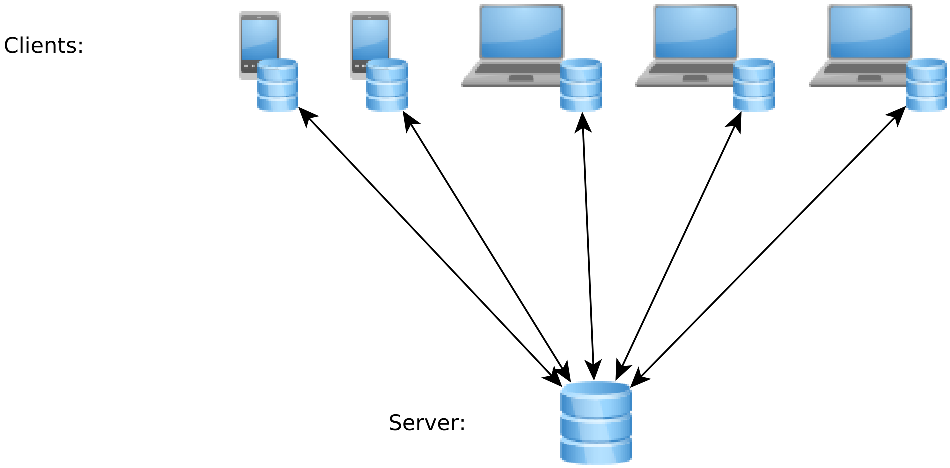
Replication

Replication: Storing the same data at multiple locations

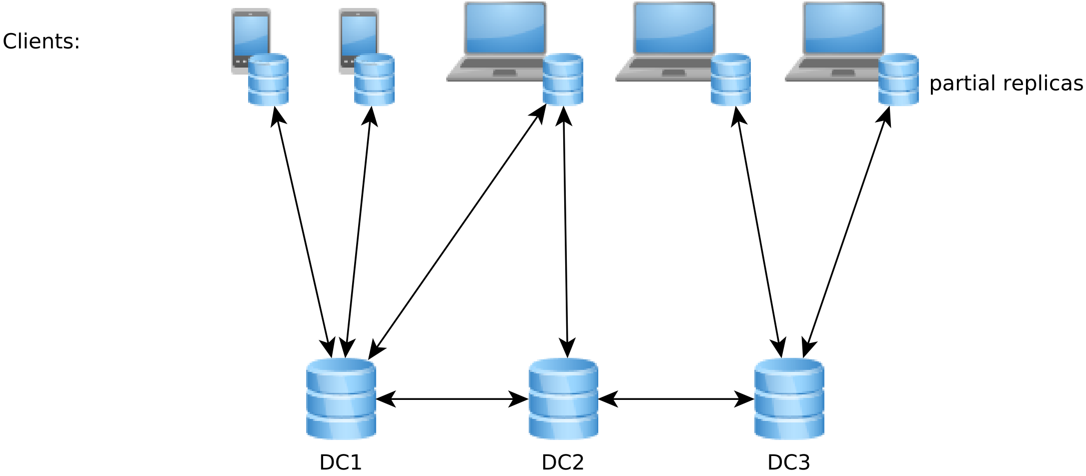
Motivation:

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

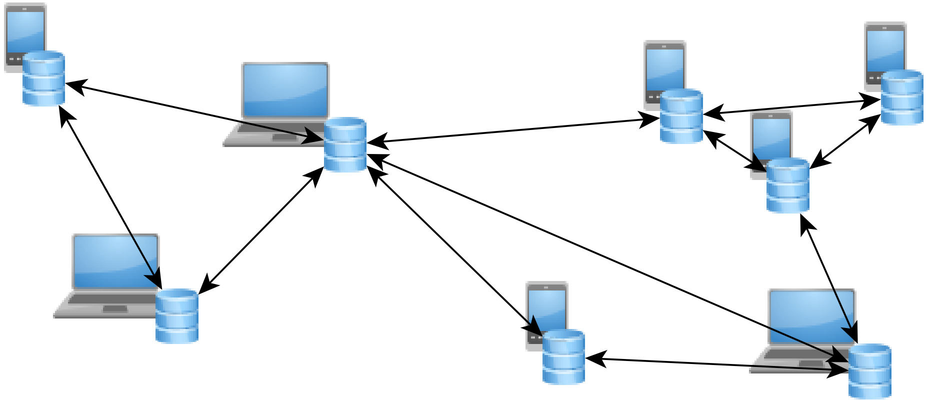
System topologies



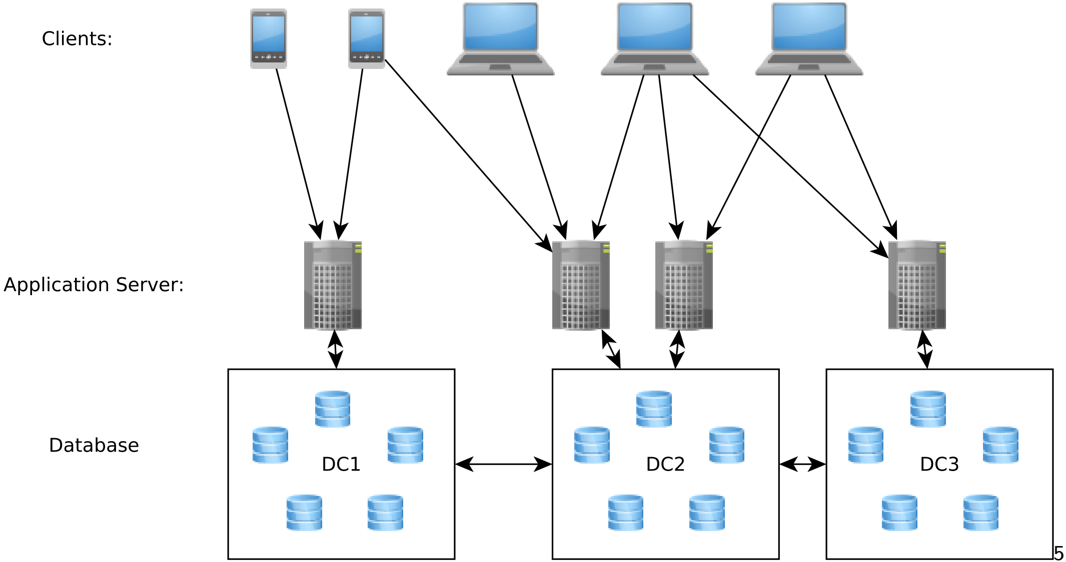
System topologies



System topologies

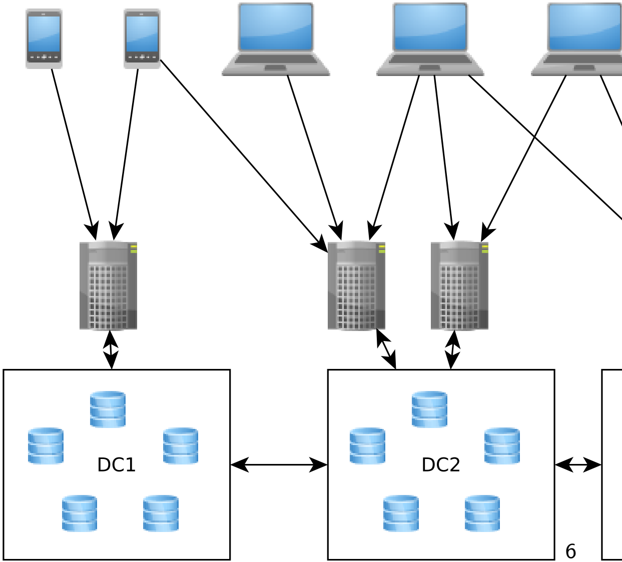


System topologies



System topology

- Where are the borders of our application?
- Where is state stored (persistently)?
- Which connections are possible?
- Where do we have concurrency?
- ...

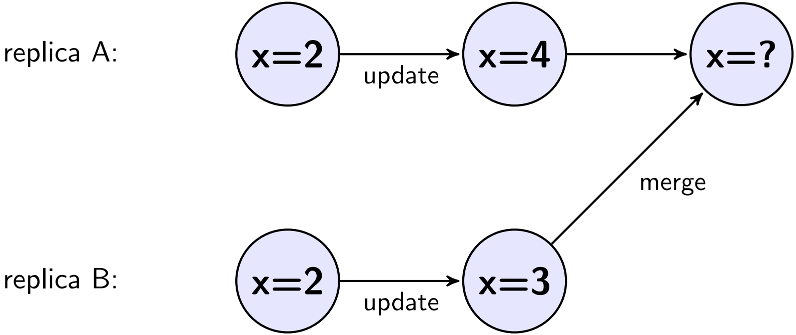


Data store infrastructure:

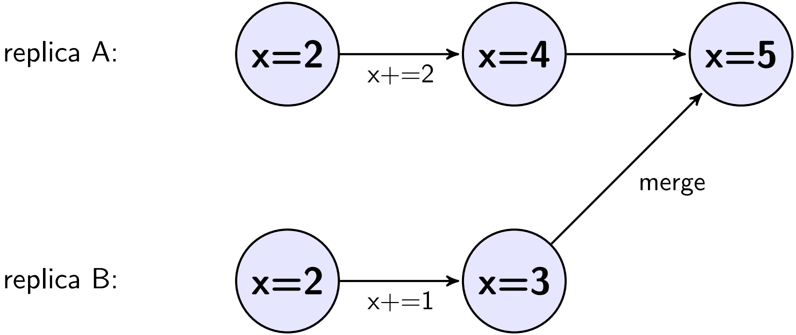
Distinguishing points:

- 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
- ...

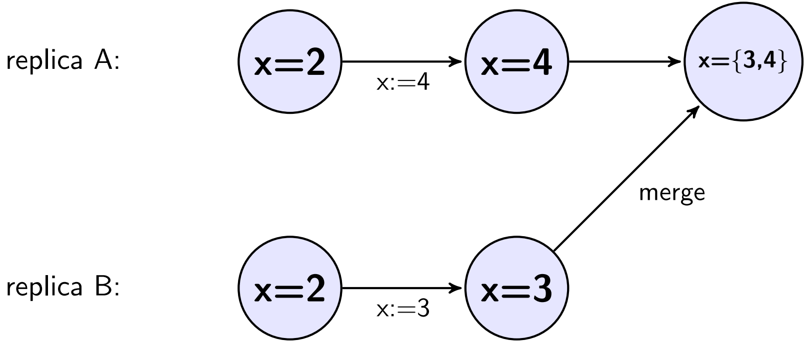
Simple example: Replicated integer variable x



Replicated counter



Replicated multi-value register



Replicated data types[1]

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

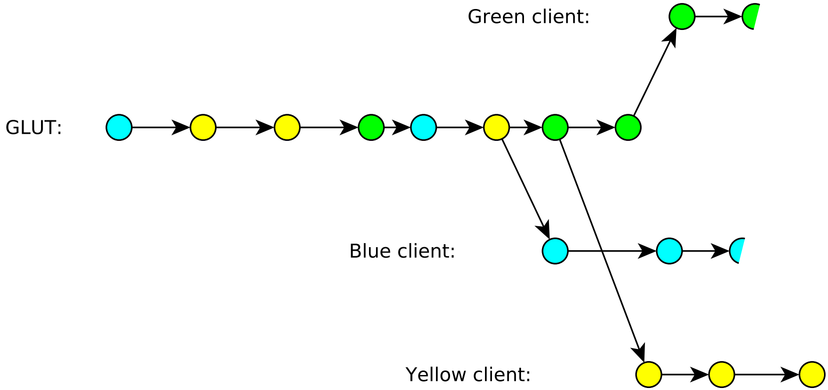
[1] Shapiro, N. Preguiça, Baquero, and Zawirski, *A comprehensive study of Convergent and Commutative Replicated Data Types*.

"Cloud types"[2][3] 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

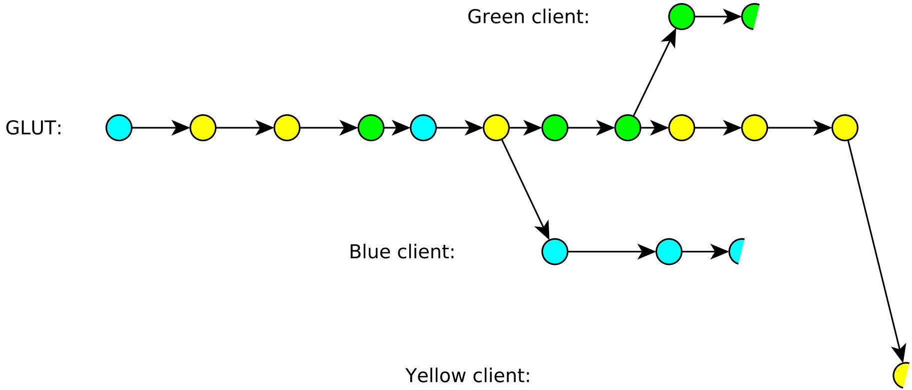
[2] Burckhardt, Fähndrich, Leijen, and Wood, "Cloud Types for Eventual Consistency".
[3] Burckhardt, Leijen, and Fahndrich, *Cloud Types: Robust Abstractions for Replicated Shared State*.

“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 - consistency model



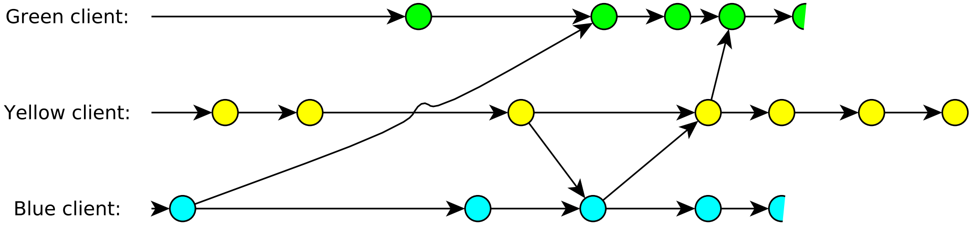
- Global log of update transactions (GLUT)
- Clients see some **prefix** of GLUT and own updates
- Merging with GLUT = appending to GLUT

“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
  - ...

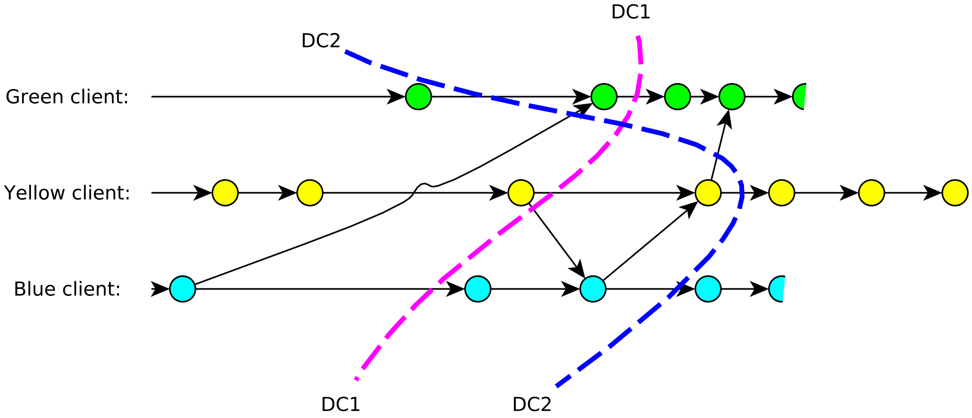
  - Not possible to define own types

SwiftCloud[4] programming model - consistency model



[4] Zawirski, Bieniusa, Balegas, Duarte, Baquero, Shapiro, and N. M. Preguiça, "SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine".

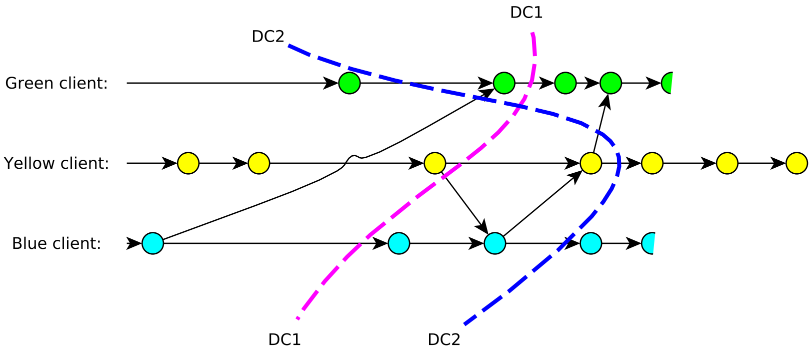
SwiftCloud[4] programming model - consistency model



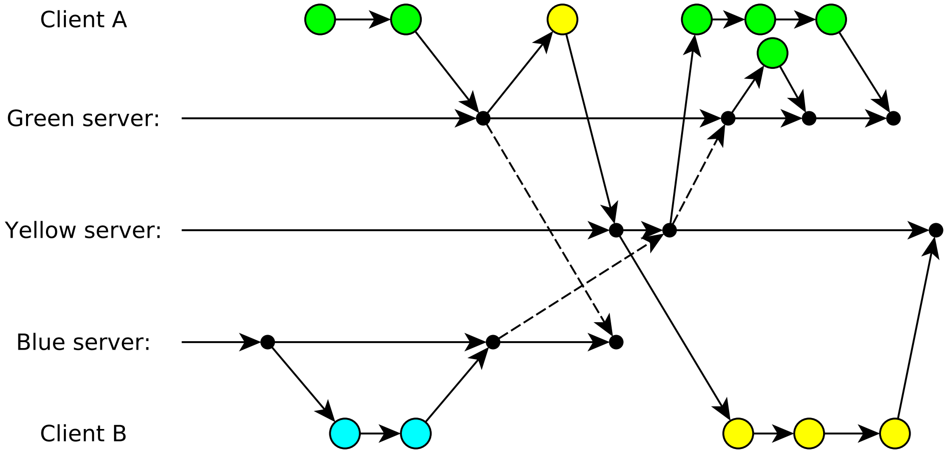
[4] Zawirski, Bieniusa, Balegas, Duarte, Baquero, Shapiro, and N. M. Preguiça, "SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine".

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[5] - consistency model



[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

Example

Task: Store the maximum score a player has reached

Sequential solution:

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

Example - Cloud types

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

Just taking the sequential solution does not work:

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)

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)
   $\rightarrow$ 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)
   $\rightarrow$ client1.score[p] $=3$
3. client2.updateScore(p, 4)
   $\rightarrow$ client2.score[p] $=4$

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
4. client2 yield
   → global.score[p] = 4

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
4. client2 yield
   → global.score[p] = 4
5. client1 yield
   → global.score[p] = 3

Example - Cloud types

"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"[6]

[6] Burckhardt, Leijen, and Fahndrich, *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

Example - SwiftCloud

SwiftCloud already includes a CRDT for keeping track of maximum values:

```
function updateScore(player, newScore)
    transaction
        MaxCRDT scoreCRDT = score[player]
        scoreCRDT.set(newScore)
```

Example - SwiftCloud

SwiftCloud already includes a CRDT for keeping track of maximum values:

```
function updateScore(player, newScore)
    transaction
        MaxCRDT 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:

```
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".

Example - Riak

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

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

General pattern:

- Use Multi-Value-Register for mutable state[7]
- Merge values in application when reading
- Write back merged value

[7] DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels, "Dynamo: Amazon's Highly Available Key-value Store".

Example - Riak

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

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

General pattern:

- Use Multi-Value-Register for mutable state[7]
- 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

```
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[8]
  - Repeat until successful

[8] Pritchett, "BASE: An Acid Alternative"; Ramalingam and Vaswani, "Fault Tolerance via Idempotence"; Helland and Haderle, "Engagements: Building Eventually ACiD Business Transactions".

Correctness

```
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

```
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[10]

[10] Najafzadeh, Shapiro, Balegas, and N. M. Preguiça, "Improving the Scalability of Geo-replication with Reservations"; N. Preguiça, Martins, Cunha, and Domingos, "Reservations for Conflict Avoidance in a Mobile Database System"; O'Neil, "The Escrow Transactional Method"; Shrira, Tian, and Terry, "Exo-Leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers"; Kraska, Hentschel, Alonso, and Kossmann, "Consistency Rationing in the Cloud: Pay Only when It Matters".

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[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[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

[11] Bieniusa, Zawirski, N. M. Preguiça, Shapiro, Baquero, Balegas, and Duarte, "Brief Announcement: Semantics of Eventually Consistent Replicated Sets".

[12] Burckhardt, Gotsman, and Yang, *Understanding Eventual Consistency*.

Conclusion

- 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

References I

- Bieniusa, Annette, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. "Brief Announcement: Semantics of Eventually Consistent Replicated Sets". In: *DISC*. Ed. by Marcos K. Aguilera. Vol. 7611. Lecture Notes in Computer Science. Springer, 2012, pp. 441–442. ISBN: 978-3-642-33650-8.
- Burckhardt, Sebastian, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. "Cloud Types for Eventual Consistency". In: *ECOOP*. Ed. by James Noble. Vol. 7313. Lecture Notes in Computer Science. Springer, 2012, pp. 283–307. ISBN: 978-3-642-31056-0.

References II

- 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.
- Burckhardt, Sebastian, Daan Leijen, and Manuel Fahndrich. *Cloud Types: Robust Abstractions for Replicated Shared State*. Tech. rep. MSR-TR-2014-43. Mar. 2014. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=211340.
- Conway, Neil, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. “Logic and lattices for distributed programming”. In: *SoCC*. Ed. by Michael J. Carey and Steven Hand. ACM, 2012, p. 1. ISBN: 978-1-4503-1761-0.

- DeCandia, Giuseppe, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.
- Helland, Pat and Don Haderle. "Engagements: Building Eventually ACiD Business Transactions". In: *CIDR*. www.cidrdb.org, 2013.
- Kraska, Tim, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. "Consistency Rationing in the Cloud: Pay Only when It Matters". In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 253–264. ISSN: 2150-8097. DOI: 10.14778/1687627.1687657. URL: http://dx.doi.org/10.14778/1687627.1687657.

References IV

- Najafzadeh, Mahsa, Marc Shapiro, Valter Balegas, and Nuno M. Preguiça. "Improving the Scalability of Geo-replication with Reservations". In: *UCC*. IEEE, 2013, pp. 441–445.
- O'Neil, Patrick E. "The Escrow Transactional Method". In: *ACM Trans. Database Syst.* 11.4 (Dec. 1986), pp. 405–430. ISSN: 0362-5915. DOI: 10.1145/7239.7265. URL: http://doi.acm.org/10.1145/7239.7265.
- Preguiça, Nuno, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. "Reservations for Conflict Avoidance in a Mobile Database System". In: *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*. MobiSys '03. San Francisco, California: ACM, 2003, pp. 43–56. DOI: 10.1145/1066116.1189038. URL: http://doi.acm.org/10.1145/1066116.1189038.

References V

- Pritchett, Dan. "BASE: An Acid Alternative". In: *Queue* 6.3 (May 2008), pp. 48–55. ISSN: 1542-7730. DOI: 10.1145/1394127.1394128. URL: http://doi.acm.org/10.1145/1394127.1394128.
- Ramalingam, Ganesan and Kapil Vaswani. "Fault Tolerance via Idempotence". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy: ACM, 2013, pp. 249–262. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429100. URL: http://doi.acm.org/10.1145/2429069.2429100.
- Shapiro, Marc, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Anglais. Rapport de recherche RR-7506. INRIA, Jan. 2011, p. 50. URL: http://hal.inria.fr/inria-00555588.

References VI

- Shrira, Liuba, Hong Tian, and Doug Terry. "Exo-Leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers". English. In: *Middleware 2008*. Ed. by Valérie Issarny and Richard Schantz. Vol. 5346. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 42–61. ISBN: 978-3-540-89855-9. DOI: 10.1007/978-3-540-89856-6_3. URL: http://dx.doi.org/10.1007/978-3-540-89856-6_3.
- Zawirski, Marek, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno M. Preguiça. "SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine". In: *CoRR* abs/1310.3107 (2013).