Cloud Types for Eventual Consistency

ECOOP 2012

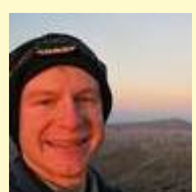
Microsoft Research, Redmond

Sebastian Burckhardt

Manuel Fähndrich

Daan Leijen

University of Washington

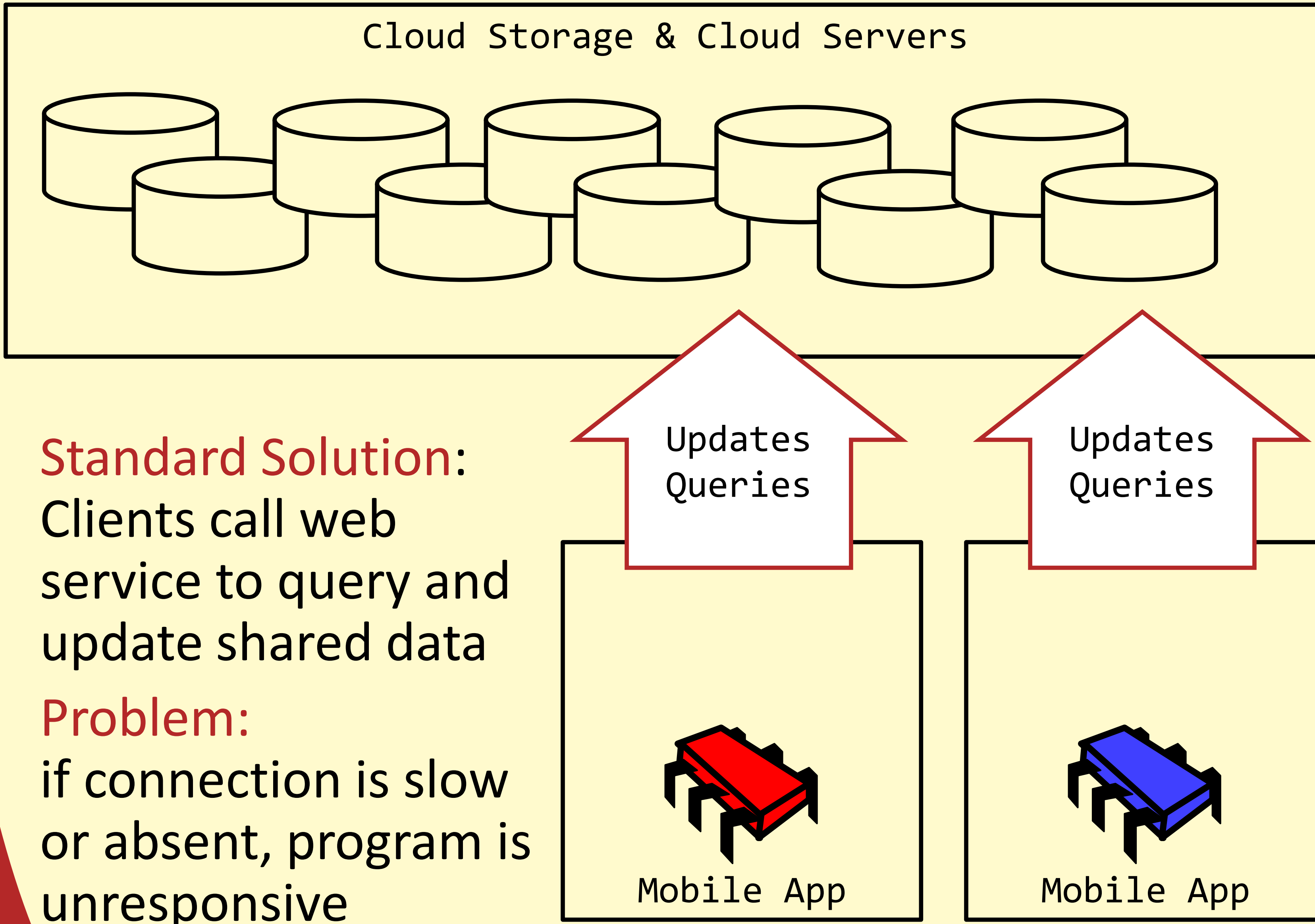


Benjamin P. Wood

Sharing Data Across Mobile Devices

- Sharing data in the cloud makes apps more social, fun, and convenient.
- Examples: Games, Settings, Chat, Favorites, Ratings, Comments, Grocery List…
- But implementation is challenging.

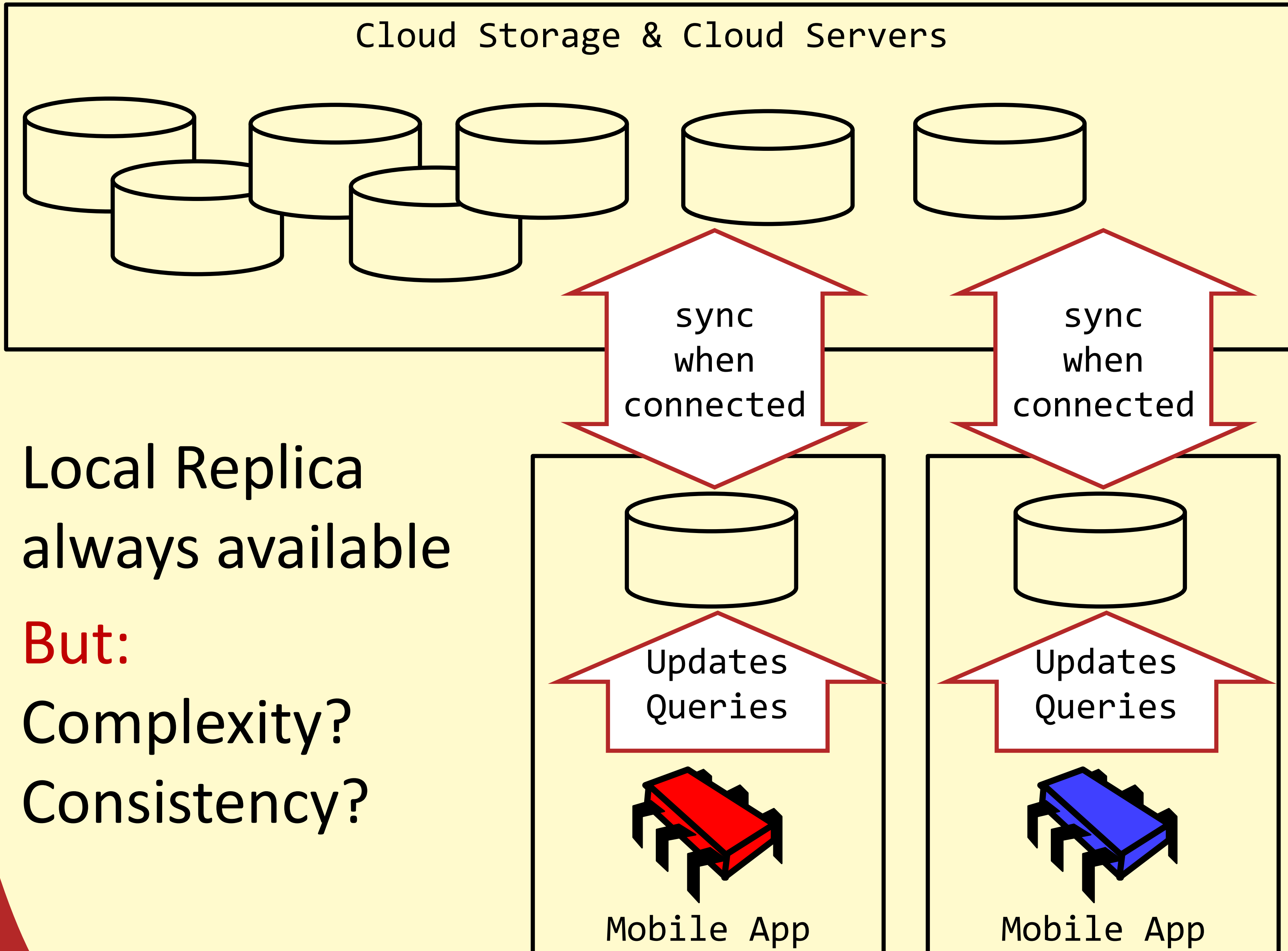
Sharing Data Across Mobile Devices



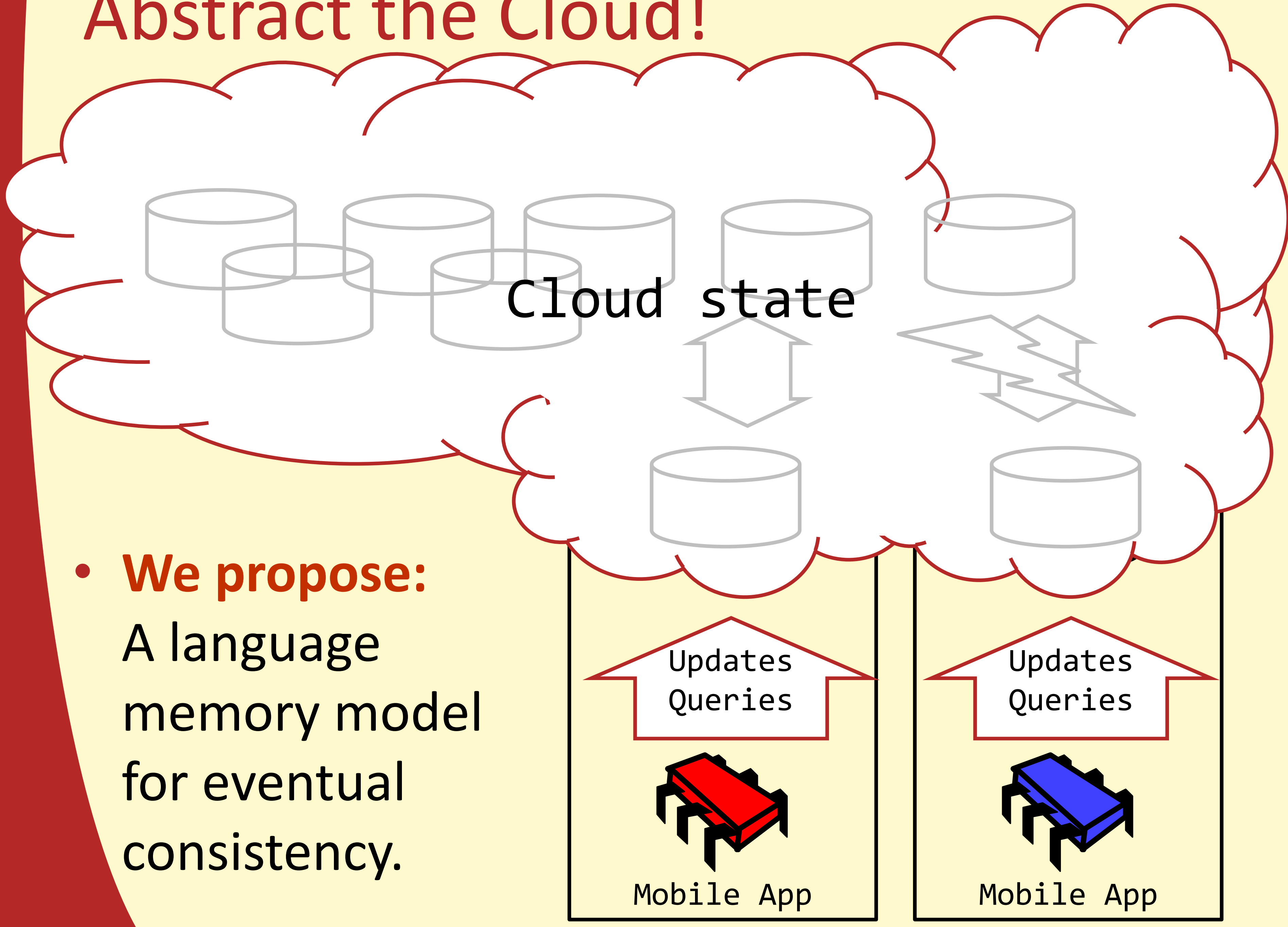
- Standard Solution: Clients call web service to query and update shared data
- Problem: if connection is slow or absent, program is unresponsive

Sharing Data w/ Offline Support

- Local Replica always available
- But: Complexity? Consistency?



Abstract the Cloud!



- **We propose:** A language memory model for eventual consistency.

Abstract the Cloud!

- **We propose:** A language memory model for eventual consistency.

Strong models, i.e.

- Sequential consistency
- Serializable Transactions

can’t handle disconnected clients.
(CAP theorem)

Neither do existing weak models (TSO, Power, Java…)

How do we define this memory model?

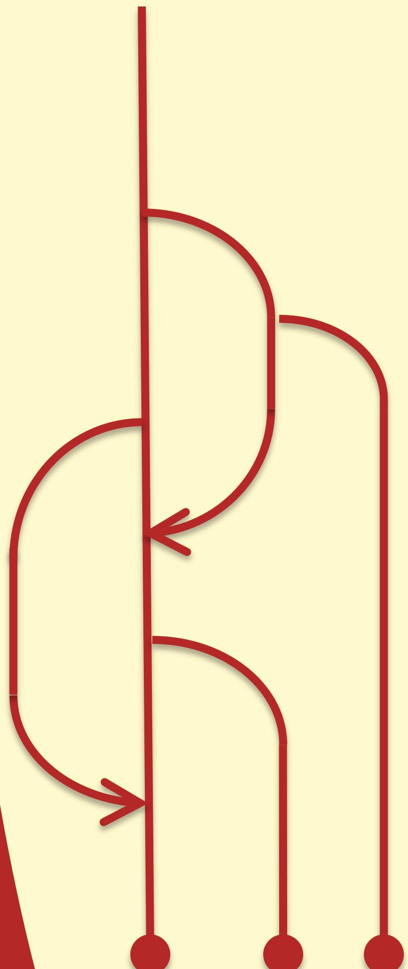
- Informal operational model

We will give you a quick intro on the next couple slides

- Formal operational model
- 2 Example Implementations (single server, server pool)
- Formal axiomatic model

Beyond the scope of this talk, see papers [ESOP2012, ECOOP2012]

Powered By **Concurrent Revisions**



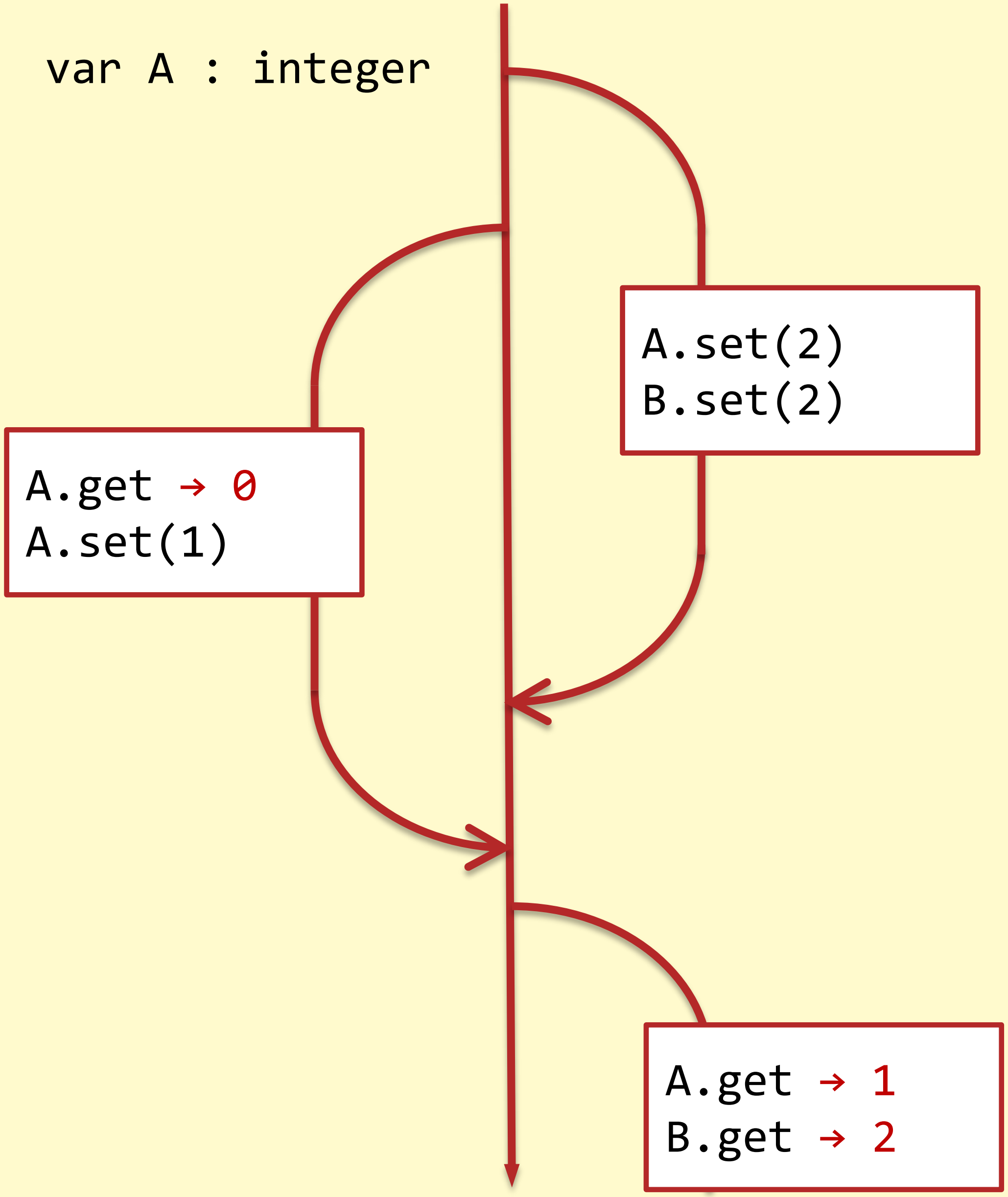
[OOPSLA'10] [WoDet'11] [ESOP'11] [OOSPLA'11] [ESOP'12] [ECOOP'12]

- reminiscent of source control systems
- but: about application state, not source code

1. Models state as a *revision diagram*
   - *Fork:* creates revision (snapshot)
   - Queries/Updates target specific revision
   - *Join:* apply updates to joining revision
2. Raises data abstraction level
   - Record operations, not just states

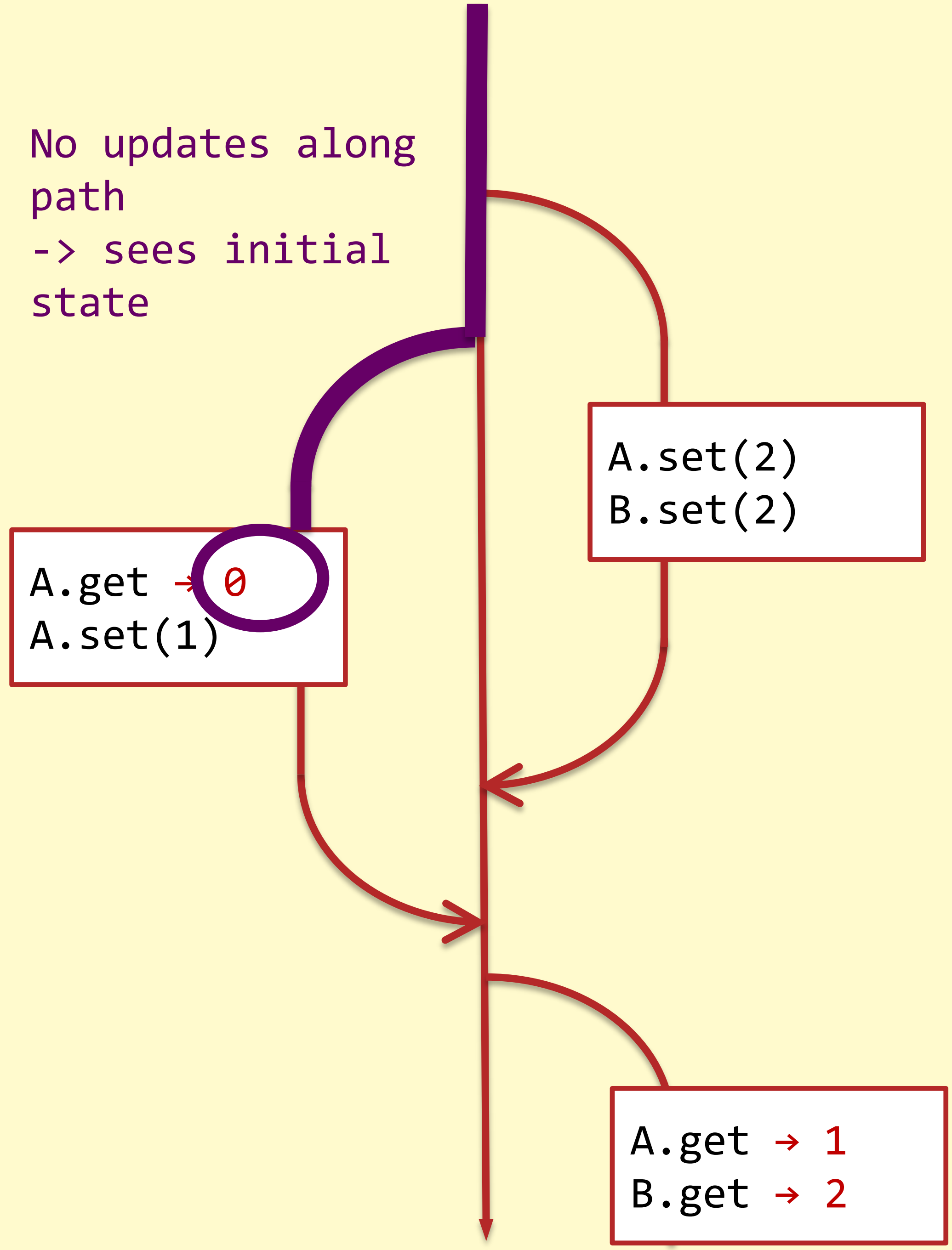
Semantics of Concurrent Revisions

- State determined by sequence of updates along path from root
- Inserts updates at tip of arrow.



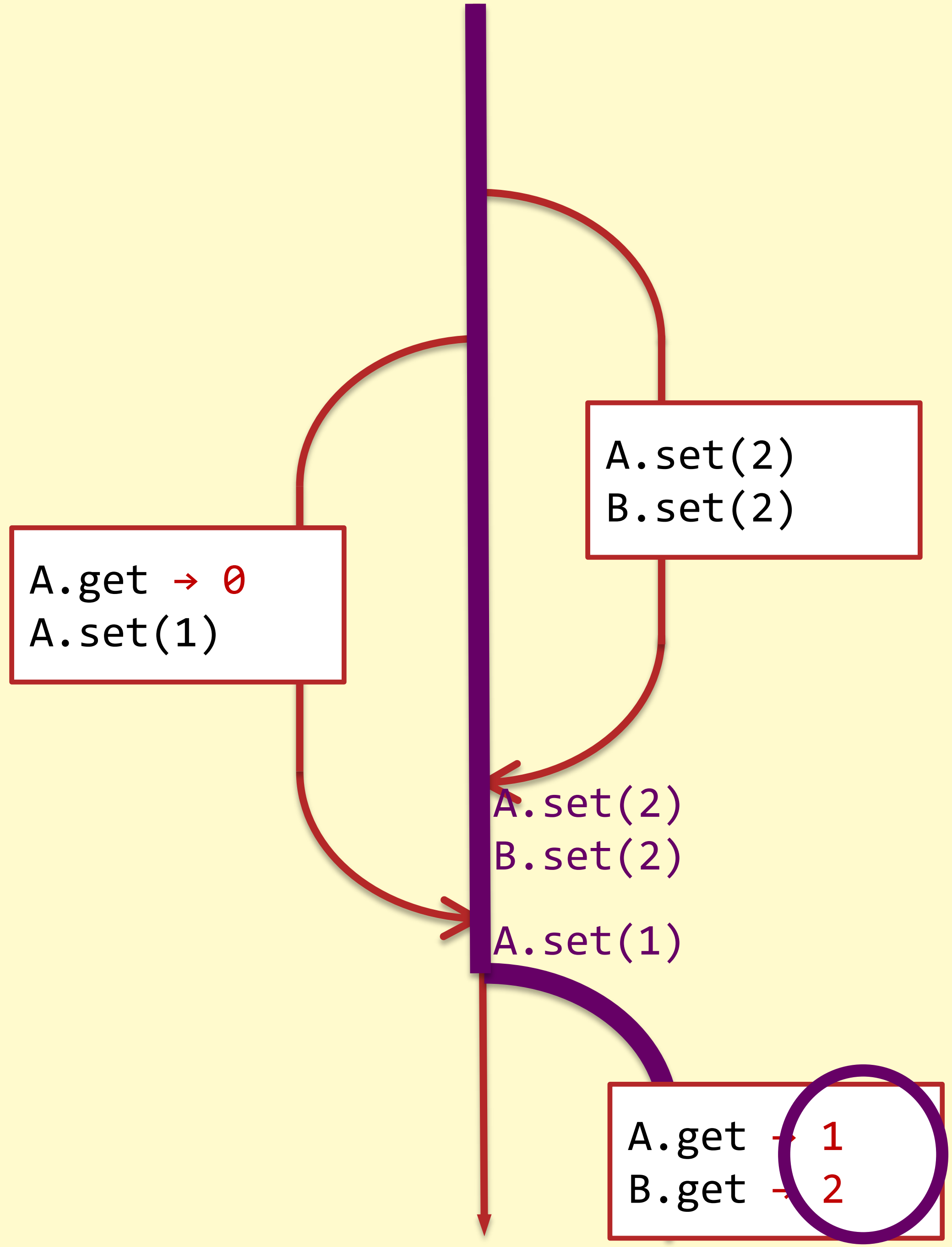
Semantics

- State determined by sequence of updates along path from root
- Inserts updates at tip of arrow.



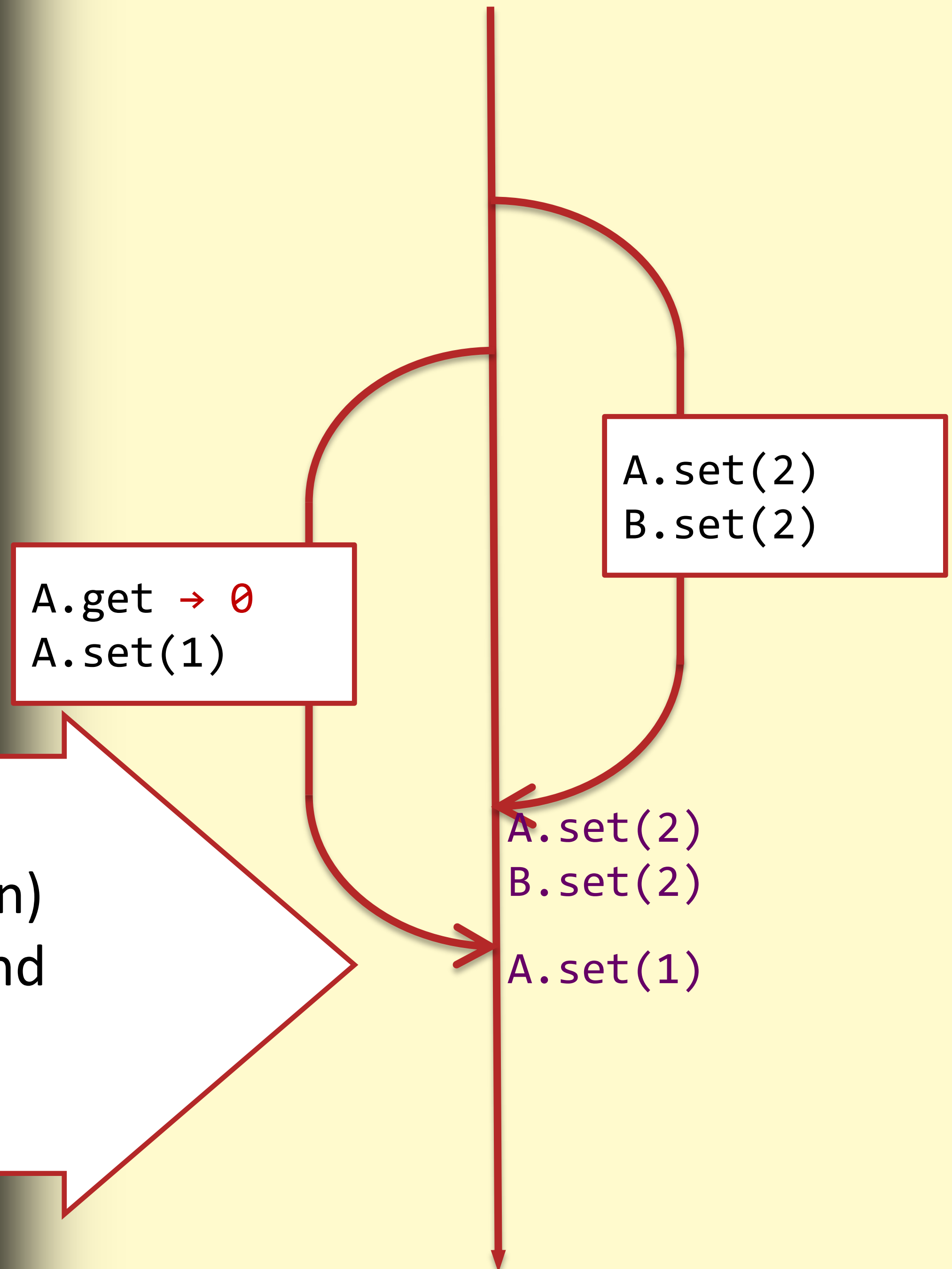
Semantics

- State determined by sequence of updates along path from root
- Inserts updates at tip of arrow.

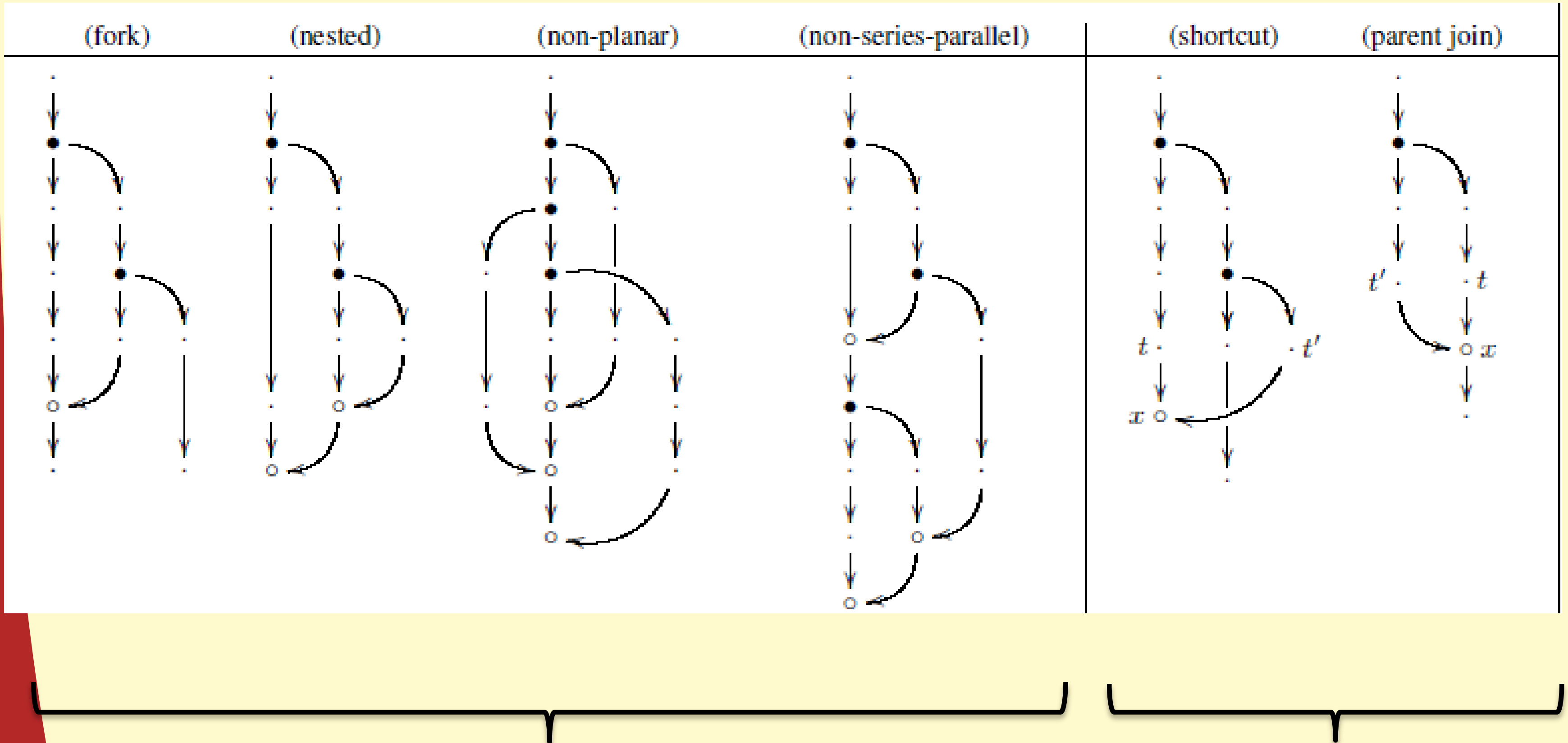


Traditional transactions (serializable, snapshot isolation) would detect a conflict here and fail.

We just keep going.



Revision Diagrams



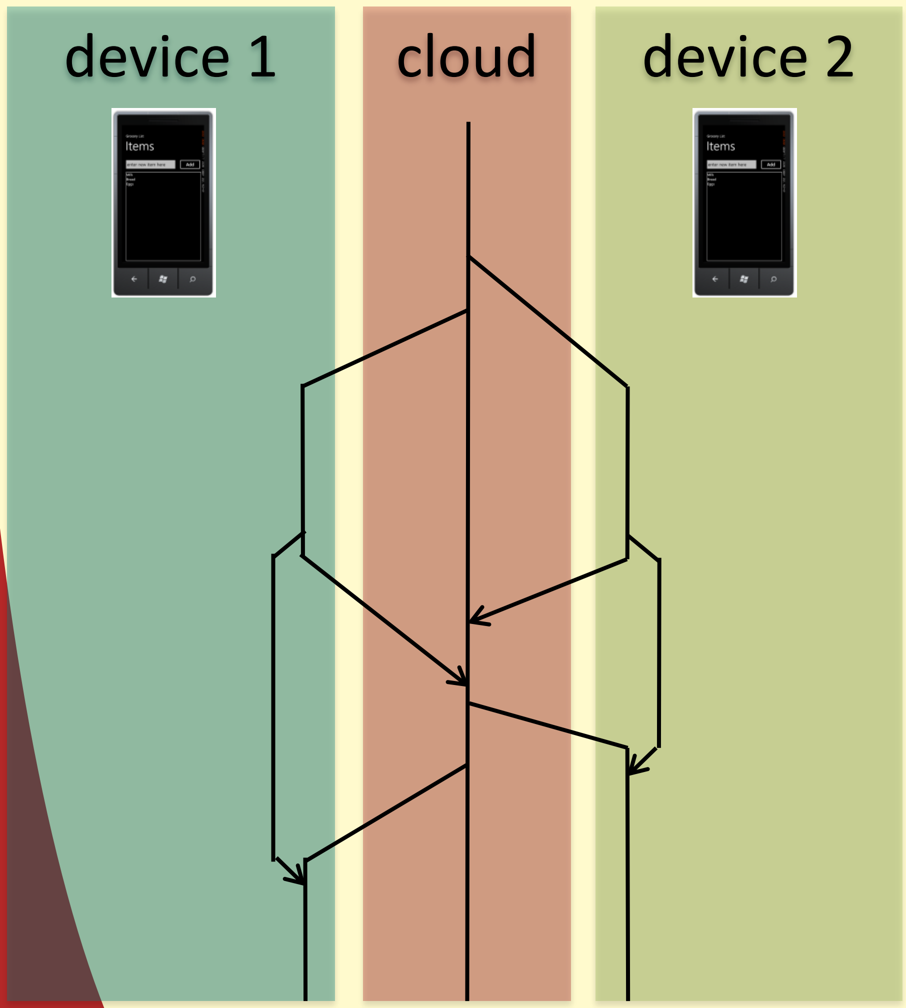
These are revision diagrams

These are not revision diagrams

- Less general than DAGs, more general than SP-graphs

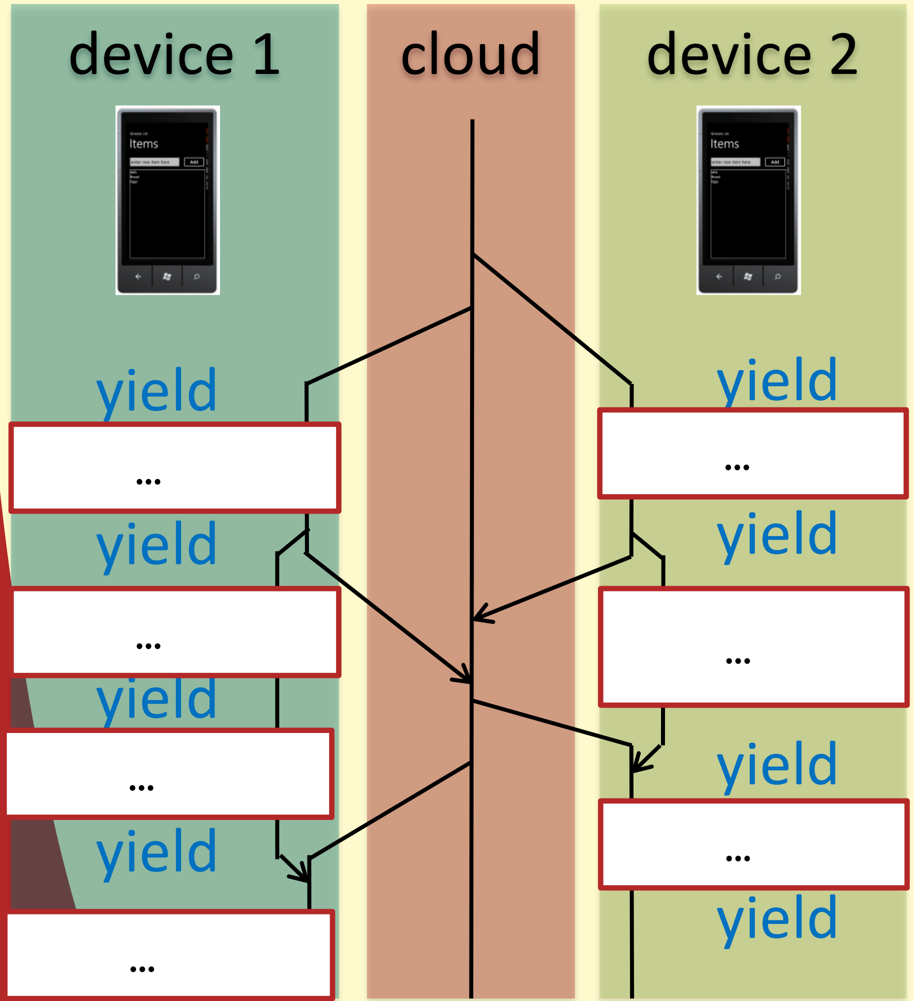
See [ESOP11], [ESOP10] for formal definitions

Cloud State = Revision Diagram



- Client code:
  - reads/modifies data
  - yields
- Runtime:
  - Applies operations to local revision
  - Asynchronous sends/receive at yield points

Yield marks transaction boundaries



- **At yield**
  Runtime has permission to send or receive updates
- **In between yields**
  Runtime is not allowed to send or receive updates

Litmus Test for Atomicity

Declare cloud variables (2 cloud integers).

Read and write cloud variables using get() and set().

```
var x : CInt;
var y : CInt;
```

Give code snippets that execute on different clients.

```
yield;
x.set(1);
y.set(1);
yield;
```

```
yield;
int a = x.get();
int b = y.get();
yield;
```

transaction boundaries given by yield statements.

```
always a == b
```

Assertion about possible final states.

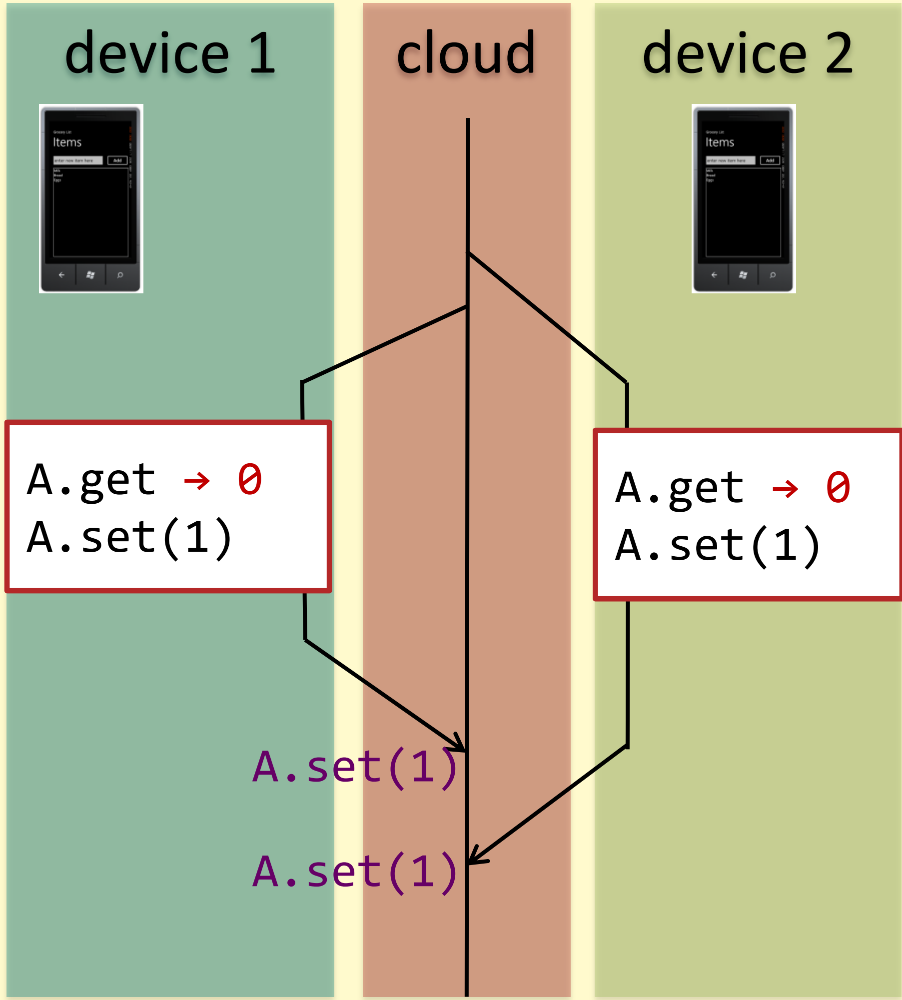
- This litmus test always passes.

Another simple Litmus Test

```
var x : CInt;
```

yield; x.set(x.get() + 1)); yield;	yield; x.set(x.get() + 1)); yield;

```
always x == 2
```



- This litmus test fails! Final value x == 1 possible.
- Because devices operate on local snapshots which may be stale.

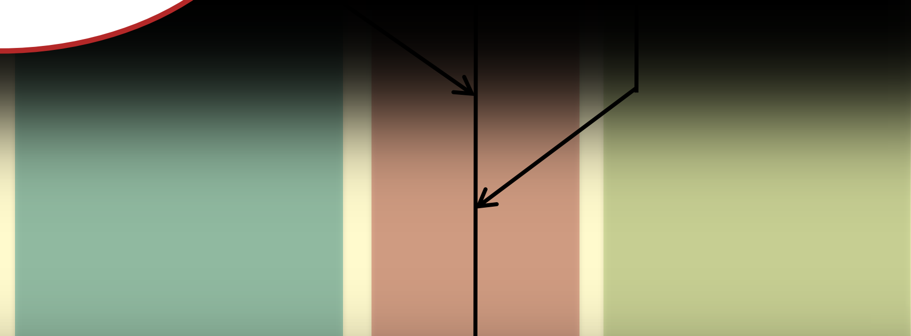
How can we write sensible programs under these conditions?

Idea: Raise Abstraction Level of Data

Use **Cloud Types** to capture more semantic information about updates.

al value x == 1 possible.

- Because devices operate on local snapshots which may be stale.



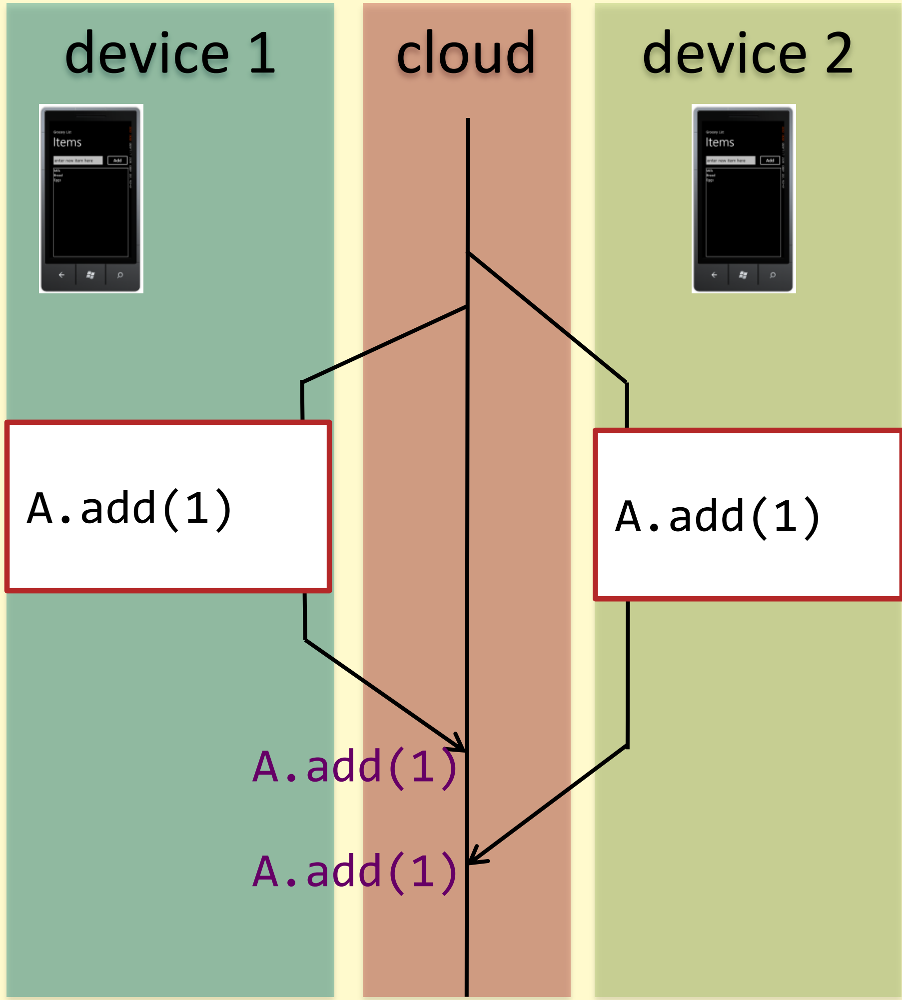
It works if we add instead of set

```
var x : CInt;
```

```
yield;
x.add(1);
yield;
```

```
yield;
x.add(1);
yield;
```

```
always x == 2
```



- Final value is determined by serialization of updates in main revision.
- Effect of adds is cumulative!

  Final value is always 2.

What is a cloud type?

- An abstract data type with
  - Initial value e.g. { 0 }
  - Query operations e.g. { get }
    - No side effects
  - Update operations e.g. { set(x), add(x) }
    - Total (no preconditions)
- Good cloud types minimize programmer surprises.

Our goals for finding cloud types…

- to select only a few
  - But ensure many others can be derived

- to choose types with minimal anomalies
  - Updates should make sense even if state changes

Forces us to rethink basic data structuring.

- objects&pointers fail the second criterion
- entities&relations do better

Our Collection of Cloud Types

Primitive cloud types

- **Cloud Integers**
  { get } { set(x), add(x) }
- **Cloud Strings**
  { get } { set(s), set-if-empty(s) }

Structured cloud types

- **Cloud Tables**
  (cf. entities, tables with implicit primary key)
- **Cloud Arrays**
  (cf. key-value stores, relations)

Cloud Tables

- Declares
  - Fixed columns
  - Regular columns
- Initial value: empty
- Operations:
  - new $E(f_1,f_2)$ — add new row (at end)
  - all E — return all rows (top to bottom)
  - delete e — delete row
  - $e.f_1$
  - $e.col_i.op$ — perform operation on cell
    - If e deleted: queries return initial value, updates have no effect

```
cloud table E
(
   f1: index_type1;
   f2: index_type1;
)
{
   col1: cloud_type1;
   col2: cloud_type2;
}
```

Cloud Arrays

- Example:

```
cloud array A
[
   idx₁: index_type₁;
   idx₂: index_type₂;
]
{
   val₁: cloud_type₁;
   val₂: cloud_type₂;
}
```

- Initial value:
  for all keys, fields have initial value
- Operations:
  - $A[i_1,i_2].val_i.op$ — perform operation on value
  - entries $A.val_i$ — return entries for which val_i is not initial value

Index types

- Used for keys in arrays
- Used for fixed columns in tables

- Can be
  - Integer
  - String
  - Table entry
  - Array entry

Example App: Birdwatching

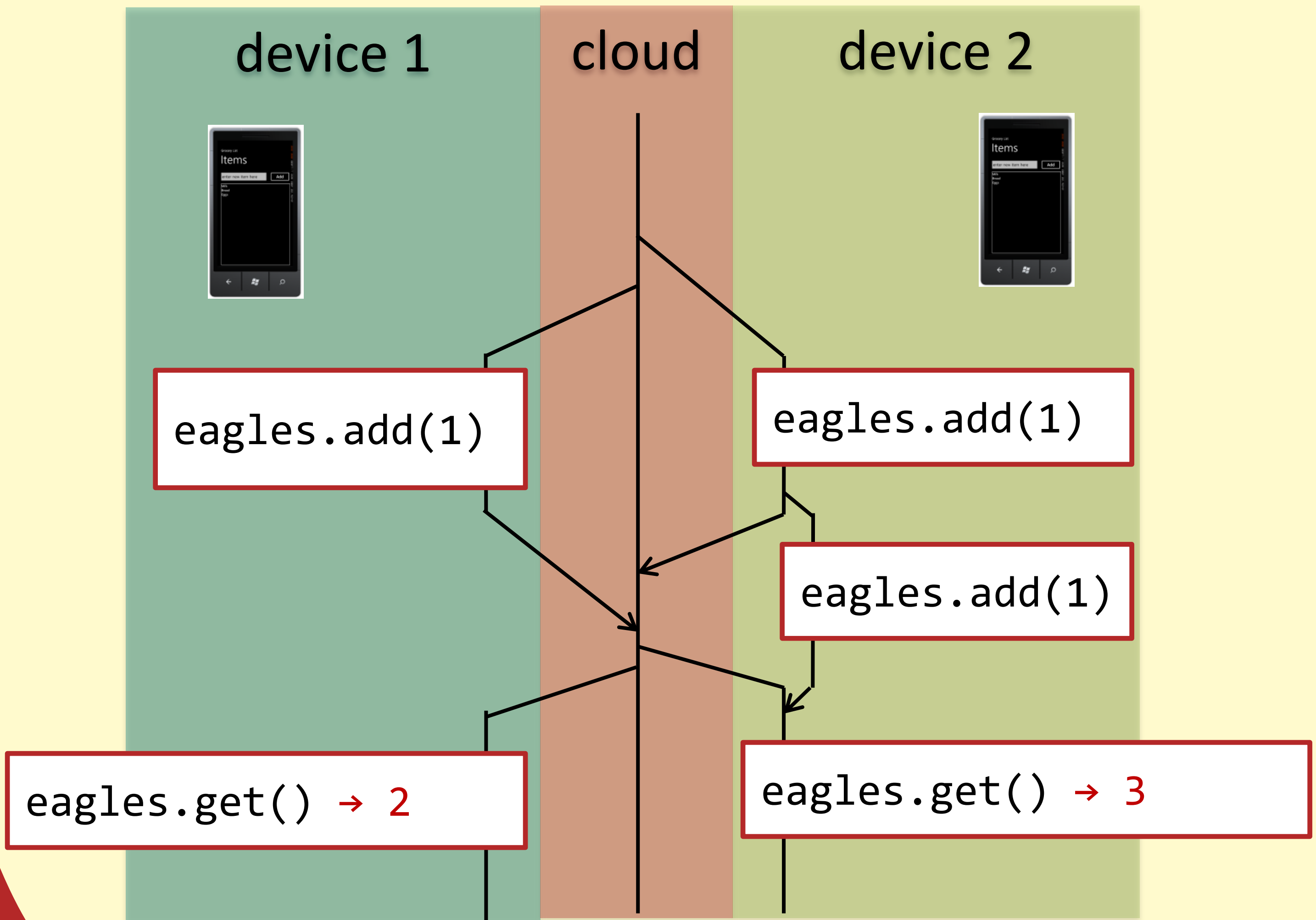
- An app for a birdwatching family.

- Start simple:
  let’s count the number of eagles seen.

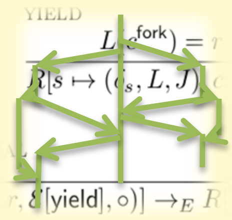
```
var eagles : cloud integer;
```

Eventually consistent counting

```
var eagles : cloud integer;
```



Counting by bird




```
var birds: cloud array
           [name: string]
           {count : cloud integer}
```

device 1 | cloud | device 2

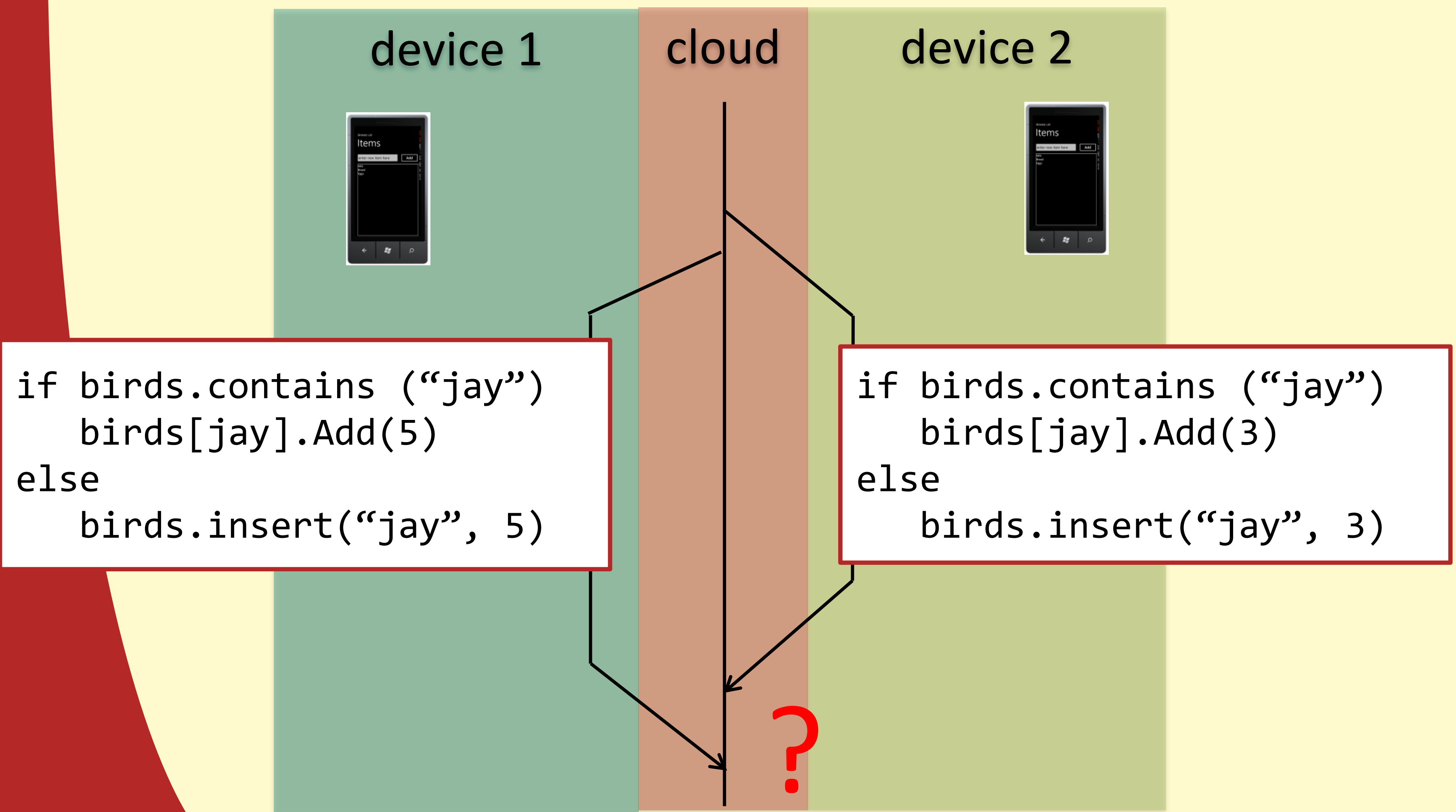

```
birds[“jay”].count.Add(5)
```

```
birds[“jay”].count.Add(1)
birds[“gull”].count.Add(2)
```

```
birds[“jay”].count.Get()
                  -> 6
```

Important: all entries are already there, no need to insert key-value pairs.

Standard Map Semantics Would not Work!



Arrays + Tables = Relational Data

- Tables
  - Define entities
  - Row identity = Invisible primary key
- Arrays
  - Define relations

- Code can access data using queries
  - For example, LINQ queries

Arrays + Tables = Relational Data

- Example: shopping cart

```
cloud table Customer
{
   name: cloud string;
}

cloud table Product
{
   description: cloud string;
}
```

```
cloud array ShoppingCart
[
   customer: Customer;
   product: Product;
]
{
   quantity: cloud integer;
}
```

Arrays + Tables = Relational Data

- Example: binary relation

```
cloud table User
{
   name: cloud string;
}

cloud array friends
(
   user1 : User;
   user2 : User;
)
{
   value: cloud boolean;
}
```

Standard math: { relations AxBxC } = { functions AxBxC -> bool }

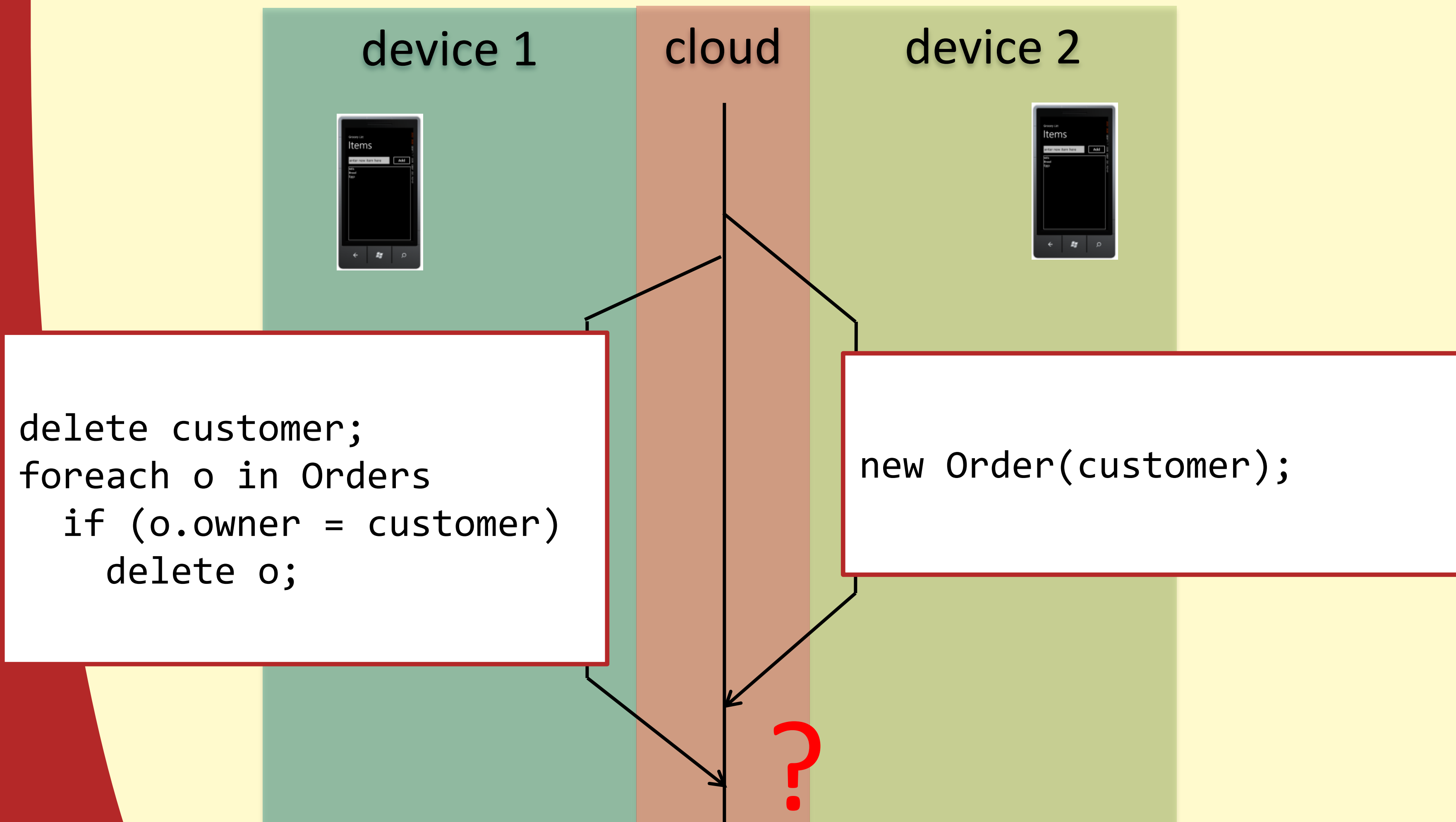
Arrays + Tables = Relational Data

- Example: linked tables

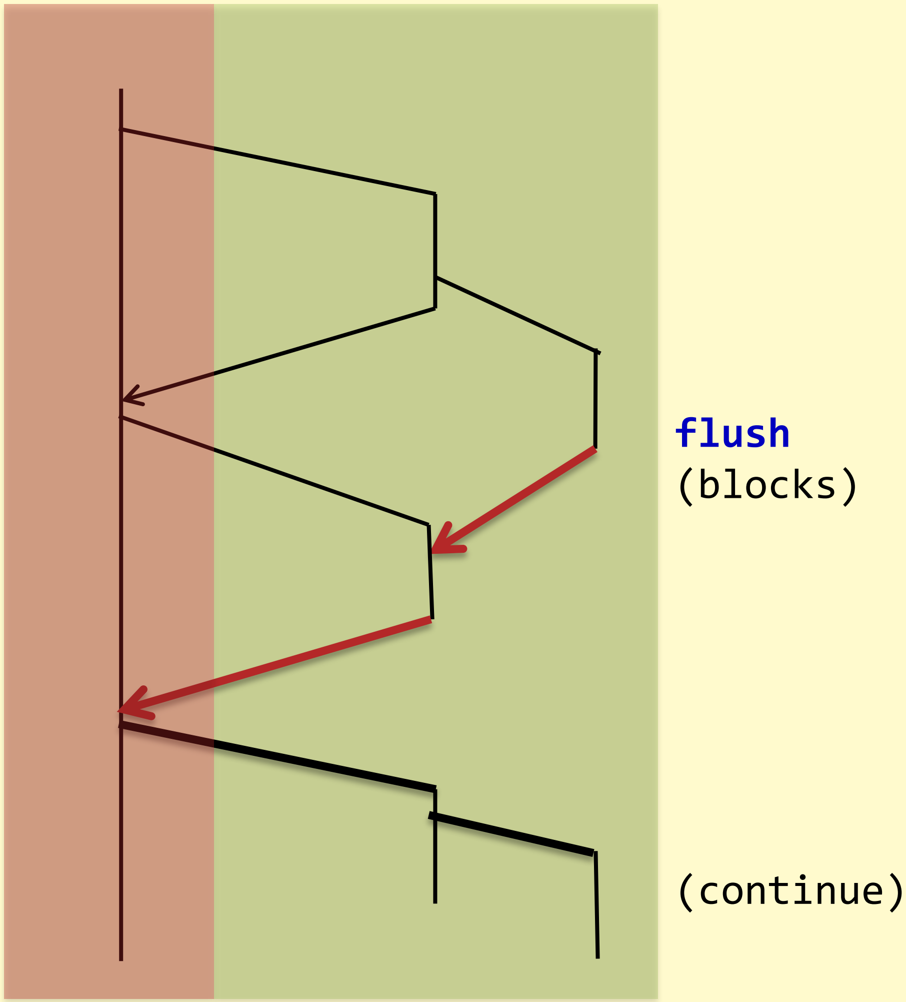
```
cloud table Customer
{
   name: cloud string;
}

cloud table Order
[
   owner: Customer
]
{
   description: cloud string;
}
```

  - Cascading delete: Order is deleted automatically when owning customer is deleted

Linked tables solve following problem:



Recovering stronger consistency



- While connected to server, we may want more certainty
- flush primitive blocks until local state has reached main revision and result has come back to device
- Sufficient to implement strong consistency

- Claim: this is not too hard. Developers can write correct programs using these primitives.
- Future work: evidence?

Implementation for TouchDevelop

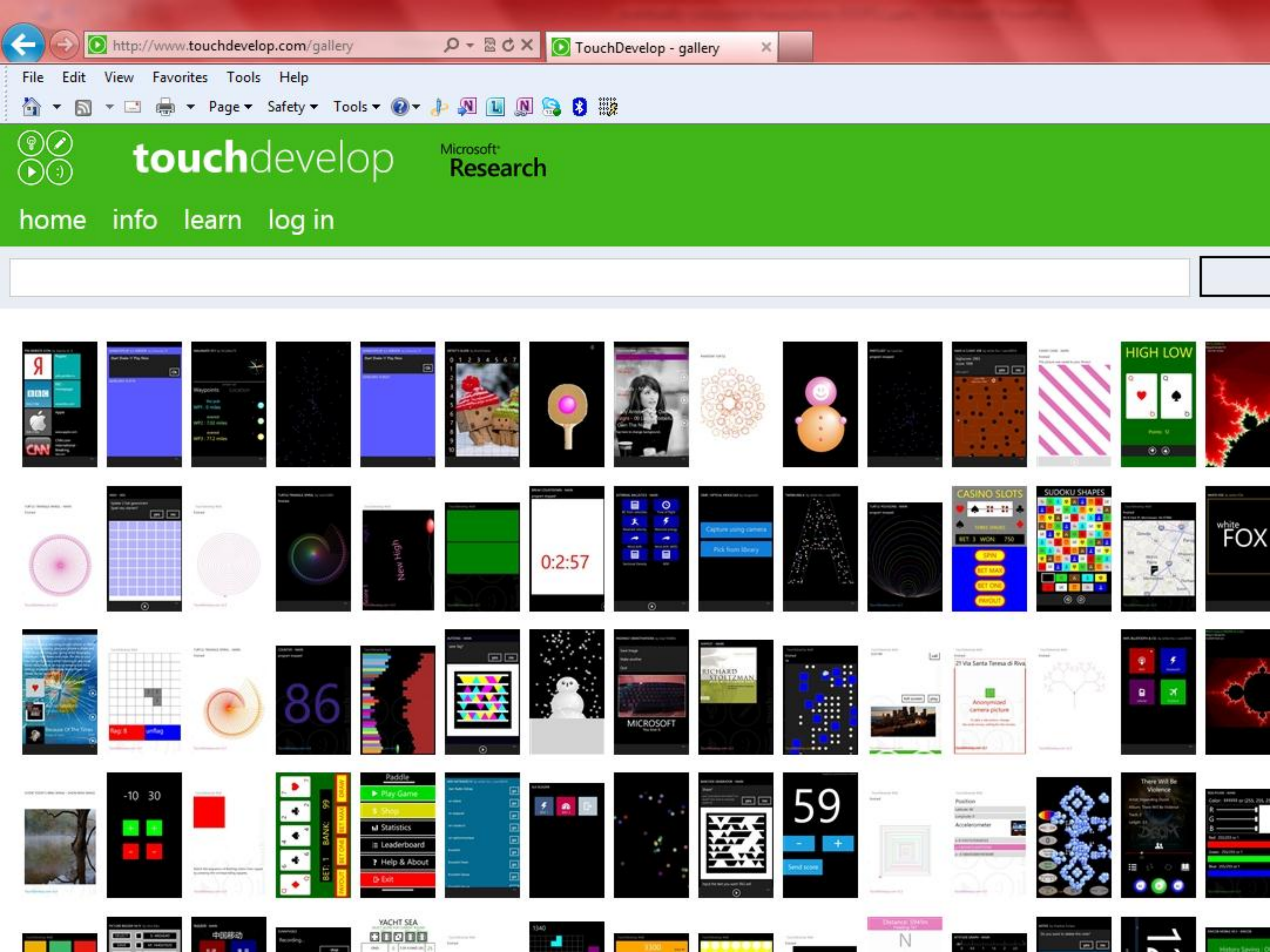
- Currently working on integration into TouchDevelop Phone-Scripting IDE.

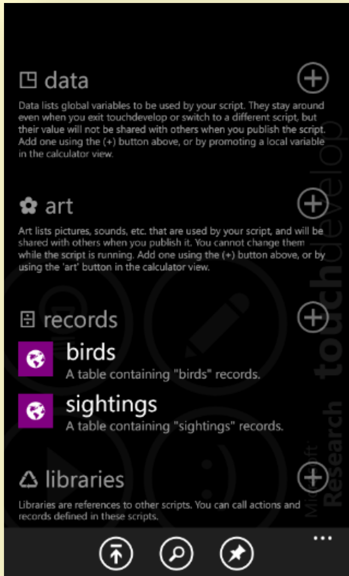
- TouchDevelop: Free app for Windows Phone, with a complete IDE, scripting language, and bazaar.

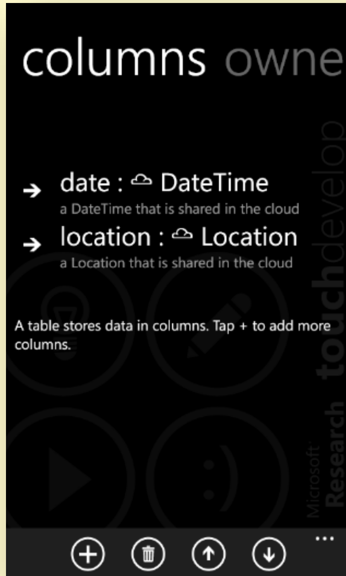
touchdevelop

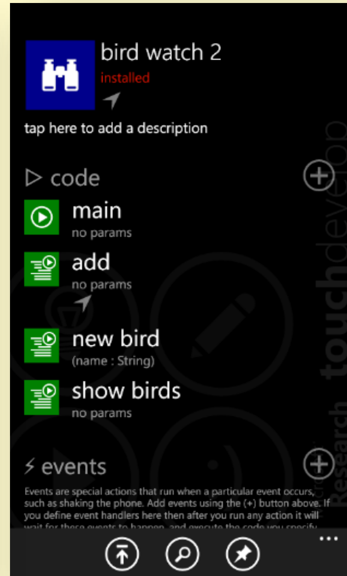
Download now!

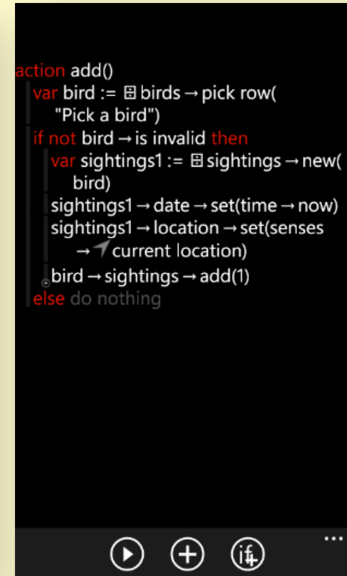
with Zune, for Windows Phone 7!


http://www.touchdevelop.com/gallery
TouchDevelop - gallery
File Edit View Favorites Tools Help
Page Safety Tools
touchdevelop
Microsoft Research
home info learn log in
HIGH LOW
CASINO SLOTS
SUDOKU SHAPES
white FOX
0:2:57
Capture using camera
Pick from library
86
MICROSOFT
-10 30
Paddle
Play Game
Shop
Statistics
Leaderboard
Help & About
Exit
59
Send score
YACHT SEA









- Declare cloud types in graphical editor
- Automatic yield
  - Before and after each script execution
  - Between iterations of the event loop

Related Work

- CRDTs (Conflict-Free Replicated Data Types)
  - [Shapiro, Preguica, Baquero, Zawirski]
  - Similar motivation and similar techniques
  - use commutative operations only
  - not clear how to do composition
- Bayou
  - user-defined conflict resolution (merge fcts.)
- Transactional Memory
- Relaxed Memory Models

Conclusion

- eventually consistent shared state is difficult to implement and reason about on traditional platforms.
- revision diagrams [ESOP11],[ESOP12] provide a natural and formally grounded intuition.
- cloud types [ECOOP12] provide a general way to declare eventually consistent storage.