Cloud Types for Eventual Consistency

Sebastian Burckhardt[1], Manuel Fähndrich[1], Daan Leijen[1], and Benjamin P. Wood[2]

[1] Microsoft Research
[2] University of Washington

Abstract. Mobile devices commonly access shared data stored on a server. To ensure responsiveness, many applications maintain local replicas of the shared data that remain instantly accessible even if the server is slow or temporarily unavailable. Despite its apparent simplicity and commonality, this scenario can be surprisingly challenging. In particular, a correct and reliable implementation of the communication protocol and the conflict resolution to achieve eventual consistency is daunting even for experts.

To make eventual consistency more programmable, we propose the use of specialized cloud data types. These cloud types provide eventually consistent storage at the programming language level, and thus abstract the numerous implementation details (servers, networks, caches, protocols). We demonstrate (1) how cloud types enable simple programs to use eventually consistent storage without introducing undue complexity, and (2) how to provide cloud types using a system and protocol comprised of multiple servers and clients.

1 Introduction

As technology progresses, new applications emerge. Of growing popularity are downloadable applications, so-called *apps*, that offer specialized functionality on a mobile device such as a phone or a tablet. Often, these apps include social aspects where users share information online. The capability of sharing data between devices is typically achieved by developing custom webservices. Increasingly, such services are deployed in the *cloud*, hosted environments that offer virtualized storage and computing resources.

Some apps require synchronization of data among multiple devices by the same users. For example, users may want to share settings, calendars, contact lists, or personal music databases. Other apps empower data sharing and communication among multiple users. For example, a simple grocery list can help family members keep track of items to be purchased on the next trip to the store. Moreover, many apps can benefit from including social features that let users share information, comments, reviews, achievements, high scores, and so on.

A pervasive requirement for apps is that they remain responsive at all times. Unfortunately, server connections are notoriously prone to slowness or temporary

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 283–307, 2012.
© Springer-Verlag Berlin Heidelberg 2012

outages. Another important requirement is that apps consume little battery power, and do not transfer much data. Thus, well-engineered apps must avoid reliance on excessive server communication.

A common technique is to maintain a replica of the data on each device. Since this replica is always available for queries and updates, apps remain responsive even if the device is disconnected. When reconnected, updates can be propagated to all other replicas in such a way that the resulting data is *eventually consistent* [12,11]. Not only does this ensure responsiveness, but users can limit connections to times where power or bandwidth are ample (such as while the device is charging at home and connected to a home network).

Clearly, sharing data via eventually consistent local replicas is an attractive solution for many applications; unfortunately, it can be daunting to implement. Typical challenges include:

- *Representation.* Because app programming is at the intersection of historically separate communities, programmers often end up writing and maintaining inordinate amounts of code to translate between different data representations (SQL, HTTP, JSON, XML, object heaps). Moreover, app programmers are forced to write custom web services, and may have to deal with subtle programming platform differences between clients and servers.
- *Consistency.* Since multiple devices can update their local replicas at the same time while disconnected, clients can detect conflicts only after the fact, when sending changes to the server. When such transactions fail, one must write code explicitly to resolve the conflict. For example, if several users update the same entry in a grocery list, we must be careful not to lose updates.
- *Change sets.* Support for disconnected operation typically means that an app must store not just a local replica, but also log a delta of all the updates that are performed locally. When the device is reconnected, these are the updates that are now sent to the server replica. Reliably resolving conflicting operations inside a large change set can be a difficult problem.

Given these hurdles and challenges it is not surprising that many apps do not implement our requirements fully: for example, updates can often only be performed while connected, and the app blocks while the transaction takes place. Other apps allow non-blocking updates but do not guarantee eventual consistency.

To make it easier for app developers to share data in the cloud, we propose the use of *cloud types* at the programming language level. Cloud types provide an abstraction layer that frees app developers of the recurring engineering challenges (web service implementation, communication protocol, local storage) and allows them to focus on the essentials: declaring the data strucures and writing client code that accesses them. The two main ingredients of our solution are:

1. *Cloud Types.* Programmers declare the data they wish to share using special cloud types. This data is automatically shared between all devices, and is automatically persisted both on local storage as well as in cloud storage. Our cloud types include both simple types (cloud integers, cloud strings)

and structured types (cloud arrays, cloud entitities). Because cloud types are carefully chosen to behave predictably under concurrent modfication, conflict resolution is automatic and the developer need not write special code to handle merging.
2. *Revision Consistency.* Our system uses *revision diagrams* to guarantee eventual consistency, as proposed in [1]. Conceptually, the cloud stores the main revision, while devices maintain local revisions that are periodically synchronized. Revision diagrams are reminiscent of source control systems and provide an excellent intuition for reasoning about multiple versions and eventual consistency.

Our approach integrates all aspects of the data model (declarations, queries, and updates) directly into the programming language. Thus, there is only one program and only one data format. Code can read and modify the data directly, without buffering or copying, and without blocking. Note that we do not propose a new language as such. Many existing languages could be extended seamlessly with support for our data model, or use libraries to expose the functionality. Currently, we have a partial implementation of our model in the TouchDevelop [17] language and development environment.

Although cloud servers are used to maintain consistency, the app developer does not write any code that executes on the server. The data declarations completely determine the functionality of the server. For the purpose of this paper, we leave out session and authorization management as they can be dealt with separately.

Overall, we make the following contributions:

- We present a data model that directly integrates support for eventually consistent data into the programming language. We demonstrate how this model allows us to write simple programs for common scenarios by walking through several examples (Section 2).
- We show how the data schema can be composed from basic cloud types (Section 2.4), and how advanced cloud types (such as observed-remove sets [13]) can be built up from simpler ones (Section 5).
- We provide a comprehensive formal syntax and semantics. It connects a small, but sufficiently expressive programming language (Section 3) with a detailed operational model of a distributed system containing a server-pool and multiple client devices (Section 4). These models are connected by a fork-join automaton (an abstract data type supporting eventual consistency) derived automatically from the schema (Section 5). Together, these models extend and concretize earlier work on eventually consistent transactions [1].

2 Overview

In this section we introduce our model in more detail, by gradually introducing the basic cloud types (cloud integers, cloud strings, cloud arrays, and cloud entitites) using examples. Along the way, we explain the execution model (revision diagrams) and the synchronization primitives (*yield*, *flush*). We start with a

simple grocery list, followed by a customer database, and conclude with a seat reservation example.

For the purposes of this paper, all examples are in pseudocode using a typed javascript-like language. We believe that the essential features of our system can be incorporated in most real-world static or dynamic languages in a seamless way. We currently have a partial implementation directly in the TouchDevelop language and as a library in C#.

2.1 Cloud Integers and Cloud Arrays

A simple but quite common scenario is the ever popular grocery list application found on all mobile devices. We show the code for this example in Fig. 1, and now proceed to explain it in detail.

```
// declaration of cloud data

global totalItems : CInt ;

array Grocery[ name : String ]
{
  toBuy : CInt ;
}

// operations representing user actions

function ToBuy( name : String, count : Int )
{
  totalItems.add(count) ;
  Grocery[name].toBuy.add(count) ;
}

function Bought( name : String, count : Int )
{
  totalItems.add(− count) ;
  Grocery[name].toBuy.add(− count) ;
}

function Display()
{
  foreach g in entries Grocery.toBuy
  {
    Print(g.toBuy.get() + " " + g.name) ;
  }
  Print(totalItems.get() + " total") ;
}

// main event loop

function main()
{
   bool done = false ;
   while (not done)
   {
     //allow send/receive of updates
     yield() ;

     match (NextUserCommand()) with
     {
        buy s n:
          ToBuy(s, n) ;
        bought s n:
          Bought(s, n) ;
        display:
          Display() ;
        quit:
          done = true ;
     }
   }
}
```

Fig. 1. Pseudocode for the grocery list example

First, consider the cloud data declarations (Fig. 1, left column, top). We use the term "cloud data" to emphasize that the data declared in this section is automatically replicated across all devices. One could say we are declaring *global variables, literally.* For this example, we chose to store both (1) a count of the total groceries to buy, and (2) a count for each individual grocery item. Although not really important for this example, storing the total count is helpful to illustrate the consistency model later on.

– To represent the total count, we declare a variable called TotalCount of type **CInt**. This type is a primitive datatype for storing and manipulating cloud integers. It differs from ordinary integer variables in that it offers *higher level operations* that have better conflict resolution semantics than using get and set operations alone. In particular, it offers an add operation to express a *relative* change, reminiscent of atomic or interlocked instructions in shared-memory programming.
– To represent the quantity of each item, we use a *cloud array* called **Grocery**. The array is indexed by the name of the grocery item, and each entry stores the quantity toBuy. This quantity is again of type **CInt**.

Cloud arrays are a bit different from standard arrays as the index type can be infinite (as in this case, strings). Cloud array entries can have multiple fields, although there is only one in this example (toBuy). Moreover, all array entries are always defined, and we guarantee that all fields are initialized with the default value (which is 0 for **CInt**). Next, consider the actions on the data (Fig. 1)

– (ToBuy) When adding count items of name name, we adjust both the total totalItems as well as the specific item count stored in the array, using the primitive add which is supported by the cloud integer type. To access the array entry, we use the name of the cloud array (**Grocery**) and an index [name] and field (toBuy).
– (Bought) When removing items from the list, we proceed the same way, but subtract the quantity rather than adding it.
– (Display) To display the list, we need to iterate over the array. This requires a bit of extra thought since we cannot just iterate over all (infinitely many) strings. Rather, we iterate over *entries* **Grocery**.toBuy, which returns only array entries for which the field toBuy is not the default value (0 for **CInt**). Thus, we print the name and the count of all items for which the count is not zero.

Finally, consider the pseudocode for the main program (Fig. 1, right column). Since the grocery list is an interactive program, it executes some form of loop to handle user commands.[1] The important part is the *yield* statement. Essentially, the *yield* statement gives the runtime system the permission to both (1) propagate changes made locally to the replica to other devices, and (2) apply changes made by other devices to the local replica. *yield* is nonblocking and guaranteed

[1] In a realistic event-based application framework the API is likely to be different, but we believe our simple loop is sufficient to convey the idea.

to execute very quickly. *yield* does not force synchronization: it is perfectly acceptable for *yield* to do nothing at all (which is in fact all it can do in situations where the device is not connected).

Another way to describe the effect of *yield* is that the *absence* of a yield guarantees isolation and atomicity; *yield* statements thus partition the execution into a form of transaction (called eventually consistent transactions in [1]). Effectively, this implies that everything is always executing inside a transaction. The resulting atomicity is important for maintaining invariants; in this example, it guarantees that the total count is always equal to the sum of all the individual counts, since all changes made to **Grocery** and totalCount are always applied atomically.

2.2 Revision Diagrams and Cloud Types

Now that we have sketched some basic language features, it is time to explain the execution model in more detail. Our semantics are based on concurrent revisions [2], and rely on the following main concepts:

- *Revision diagrams* are reminiscent of source control systems, and show the order in which revisions are forked and joined. Conceptually, each revision keeps a log of all the updates that were performed in it. When a revision is joined into another revision, it replays all logged updates into that revision.
- *Cloud types* are abstract data types that offer a precisely defined collection of update and query operations. Moreover, cloud types can provide optimized fork and join implementations and space-bounded representations of logs.

For example, consider a cloud variable x of type **CInt** and the revision diagram examples in Fig 2. Note that join is not symmetric; join orders updates of the joined revision *after* the updates of the joining revision, and update operations are not always commutative (for example, two add operations do commute, but set operations do not commute).

2.3 Execution Model and Eventual Consistency

We can now employ revision diagrams to build an eventually consistent distributed system. The idea is to keep the main revision on the server, and to keep some revision always available on each device, whether connected or not. We can

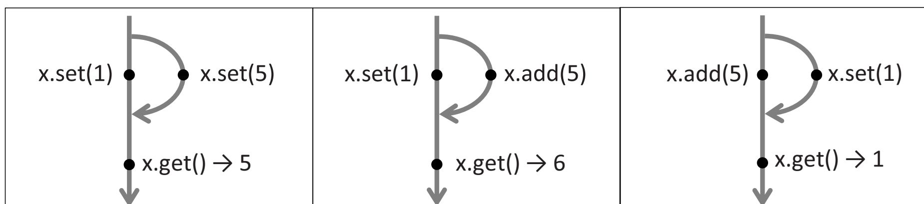
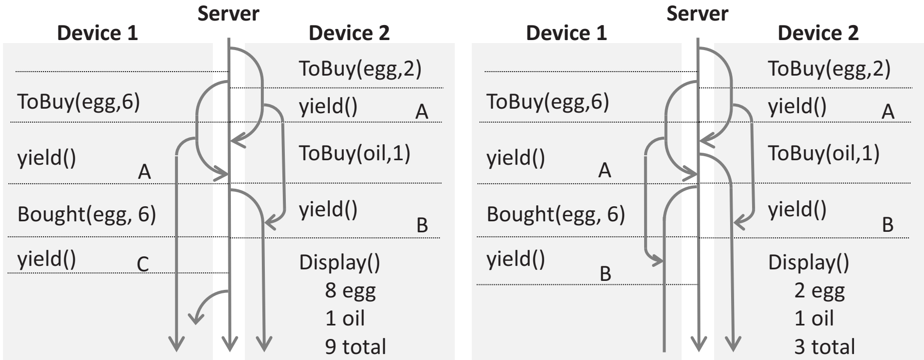


Fig. 2. Conflict resolution for **CInt**. Updates of the revision are replayed at the join point.

send revisions from the server to clients, and vice versa, and perform forks and joins on either one.

Program execution is nondeterministic if multiple devices are involved.[2] Both of the following revision diagrams represent possible executions of the grocery list example. Because of timing differences, the Display() on device 2 may either see the first update by device 1 (left) or not see it (right).



Forking and joining of revisions on the server is straightforward. The implementation of *yield* on the device is a bit more clever, since it is guaranteed to always execute quickly and never block (regardless of message speed or lost messages). We achieve this by distinguishing 3 cases (labeled A,B,C in the executions above): (A) If we are not currently expecting a server response, we send the current revision to the server, after forking a new revision for continued local use. (B) If a revision from the server has arrived, we merge the current local revision into it. (C) If we are expecting a revision from the server but it is not present yet, we do nothing.

As long as clients repeatedly call *yield*, and as long as messages are eventually delivered (using retransmission if necessary), eventual consistency is achieved. We present formal operational semantics for *yield* in Section 4.

Since revision diagrams are quite general, a wide variety of implementation choices beyond the one sketched above can be employed (such as servers organized as trees, or clients bundled with servers connected peer-to-peer). We discuss one specific multi-server implementation model (a server pool) in Section 4.

2.4 Entities

Our model is versatile enough to store complex relational data. In Fig. 3, we consider a mobile application that maintains a database of customers and orders (as may be used by a small business in an emerging market).

[2] Determinism makes no sense for eventually consistent systems, since such systems are expected to adapt opportunistically to unpredictable message latency and loss.

```
entity Customer
{
  name : CString
}
array Product[ id : string ]
{
  name : CString
  price : CInt
}
entity Order(customer : Customer)
{
  time : CTime
  totalprice : CInt
}
array CartItem [
  customer : Customer,
  product : Product ]
{
  quantity : CInt ;
}
array OrderItem [
  order : Order,
  product : Product ]
{
  quantity : CInt
  price : CInt
}
function AddToCart(c: Customer,
     p: Product, q: int)
{
  CartItem[c,p].quantity.add(q) ;
}
function DeleteCustomer(c: Customer)
{ delete c ; }

function SubmitOrder(customer : Customer)
{
  // create fresh order
  var order = new Order(customer) ;
  order.time.set(now()) ;
  // move items from cart
  foreach cartitem in entries CartItem.quantity
    where cartitem.customer == customer
  {
     var oitem = OrderItem[order, cartitem.product] ;
     oitem.quantity = cartitem.quantity ;
     oitem.price = cartitem.quantity
                   * cartitem.product.price ;
     cartitem.quantity.add(−oitem.quantity) ;
     order.totalprice.add(oitem.price) ;
  }
}

function ShowOrders(customer : Customer)
{
  foreach order in all Order
    where order.Customer == customer
    orderby order.time
  {
    Print("Order of " + order.time) ;
    foreach(i in all OrderItem)
      where i.order == order
    {
       Print(i.quantity + " " + i.product.name +
             " for " + i.price) ;
    }
  }
}
```

Fig. 3. Pseudocode for the customer database example

Since arrays do not support dynamic creation or deletion of entries, we introduce an alternative form of data structures, called entities.[3] We model customers and orders as entities rather than array entries, which has two advantages: (1) we can create them without first determining an index by which to identify them uniquely, and (2) we can delete them explicitly, which removes them (as well as all associated data) from the database.

[3] Note that both our arrays and our entities are special cases of the general notion of entities as used in Chen's entity-relationalship model [3]. The distinction is that our array entries have visible primary keys (the indexes), and can not be created or deleted, while our entities have hidden, automatically managed primary keys, and are explicitly created and deleted by the user.

Product is an array of products, indexed by a unique id, and **CartItem** is an array of cart items, indexed by customers and products, storing the quantity. The entity **Order** takes a customer as a construction argument (construction arguments are like immutable fields, but also play an additional role explained below), and the array **OrderItem** stores the quantity of each product in each order.

The function AddToCart adds items to a customer's cart, just as we added items to the grocery list in the previous example. The function SubmitOrder creates a new order entity for the customer, then iterates through the cart items of this customer, and adds them to the order, totaling the prices. Note that since there is no *yield* in this function, we need not worry about the order entity becoming visible to other devices before all of its information is computed. The function ShowOrders prints all orders by a customer, sorted by date. It uses the query *all* **Order** *where* order.**Customer** == order which returns all order entities belonging to this customer.

The function DeleteCustomer is simple, but has some interesting effects. Not suprisingly, it deletes the customer entity. But beyond that, it also clears all entries in all arrays that have the deleted customer as an index, and it even deletes all orders that have the deleted customer as a construction argument.[4]

2.5 Stronger Consistency

Eventual consistency is not always sufficient. Some problems, such as reserving a seat on an airplane, or withdrawing money from a bank account, involve a limited resource and require true arbitration. In such cases, we *must* establish a server connection and wait for a response. In this section, we show how to reintroduce strong synchronization.

Consider an application making seat reservations, which may attempt something like the following:

```
array Seat [
  row : int,
  letter : string ]
{
  assignedTo : CString;
}
```

```
function NaiveReserve(seat: Seat, customer : string)
{
  if (seat.assignedTo.get() == "")
    seat.assignedTo.set(customer);
  else
    print("reservation failed");
}
```

Unfortunately, this does not work as desired: a seat may appear empty in the local revision, but already be filled on the server. In this case, the NaiveReserve function would appear to succeed, but in fact may overwrite another reservation once the update reaches the server. We fix this problem by introducing a primitive operation setIfEmpty for the cloud type **CString**. This operation sets the string only if it is currently empty, and this condition is reevaluated when the update operation is applied on the server. Thus, existing reservations are never overwritten.

[4] Entities whose existence depends on other entities are sometimes called 'weak entities' in the literature. In our system, those 'weak entities' correspond to (1) entities that have other entities appearing in their construction arguments, and (2) array entries that have entities appearing as an index.

However, *yield* is still not sufficient to force mutual exclusion, since we can not tell when the update has reached the server. Thus we support an additional synchronization primitive called *flush*. Upon *flush*, execution blocks until (1) all local updates have been applied to the main revision, and (2) the result has become visible to the local revision. Now we can implement the body of the reservation function as follows:

```
seat.assignedTo.setIfEmpty(customer) ;
flush ;
if (seat.assignedTo.get() ≠ customer) print("reservation failed") ;
```

Since *flush* could block indefinitely if the device is not connected, our implementation supports the specification of a timeout.

This example is interesting since it shows that our model is at least as expressive as shared-memory programming with locks (locks can be implemented analogously). However, it does not represent the type of application for which our model is most suited. On the contrary, for applications that frequently require strong synchronization, the benefits of our model are marginal, and traditional OLTP (online transaction processing) is likely more appropriate.

3 Syntax, Types, and Local Semantics

Figure 4 describes the syntax of types, schemas, and expressions. We distinguish three kinds of types. The index type ι is the type of values that can be used as indices into an array or entity, and consists of simple read-only values like **Int**, **String**, and array and entity identifiers (A and E). The cloud type ω is used for mutable cloud values that are persisted. We prefix such types with the letter **C** to distinguish them from regular value types. Examples of cloud types are **CInt** and **CString**. In Section 5 we give precise semantics to these cloud values using *fork-join automata*. The type **CSet**$\langle\iota\rangle$ is the type of *observed-remove* sets as described by Shapiro [13]. Finally, we have expression types τ which includes index types ι, functions, products, and regular sets. We denote the trivial product ($n = 0$) by **Unit**.

A schema $\mathcal{S}$ consists of a sequence of declarations. A declaration is either an array A, an entity E, or a property p. Properties map an index ι to a mutable cloud type ω. In our examples, we used the following syntactic sugar to define properties as part of an array or entity declaration:

$$\textit{entity}\ E(k_1 : \iota_1, ..., k_m : \iota_m)\ \{\, p_1 : \omega_1, ..., p_n : \omega_n \}$$
$$\equiv\ \textit{entity}\ E(k_1 : \iota_1, ..., k_m : \iota_m);\ \textit{property}\ p_1 : E \rightarrow \omega_1;\ ...;\ \textit{property}\ p_n : E \rightarrow \omega_n$$

$$\textit{array}\ A[k_1 : \iota_1, ..., k_m : \iota_m]\ \{\, p_1 : \omega_1, ..., p_n : \omega_n \}$$
$$\equiv\ \textit{array}\ A[k_1 : \iota_1, ..., k_m : \iota_m];\ \textit{property}\ p_1 : A \rightarrow \omega_1;\ ...;\ \textit{property}\ p_n : A \rightarrow \omega_n$$

Also, global persisted values (as used for example in the grocery list in Figure 1) are syntactic sugar for cloud arrays without any keys and a single value property:

$$\textit{global}\ x : \omega\ \equiv\ \textit{array}\ x[\,]\ \{\, value : \omega \,\}$$

entity names	$Ent \ni$	E	$::= ...$
array names	$Arr \ni$	A	$::= ...$
index types		ι	$::= \textsf{\textbf{Int}} \mid \textsf{\textbf{String}} \mid E \mid A$
cloud types		ω	$::= \textsf{\textbf{CInt}} \mid \textsf{\textbf{CString}} \mid \textsf{\textbf{CSet}}\langle\iota\rangle \mid ...$
expression types		τ	$::= \iota \mid \textsf{\textbf{Set}}\langle\tau\rangle \mid \tau \rightarrow \tau \mid (\tau_1, ..., \tau_n)$
key names		k	$::= ...$
property names		p	$::= ...$
declarations		$decl$	$::= \textit{entity}\ E(k_1 : \iota_1, ..., k_n : \iota_n)$
			$\mid\ \textit{array}\ A[k_1 : \iota_1, ..., k_n : \iota_n]$
			$\mid\ \textit{property}\ p\ :\ \iota \rightarrow \omega$
schema		$\mathcal{S}$	$::= decl_1; ...; decl_n$
unique id's	$Uid \ni$	uid	$::= ...$ (abstract)
constants	$Con \ni$	c	$::= ...$ (integer and string literals)
updates		op_u	$::= ...$ (predefined)
queries		op_q	$::= ...$ (predefined)
operations		op	$::= op_u \mid op_q$
values	$Val \ni$	v	$::= A[v_1, ..., v_n] \mid E[uid, v_1, ..., v_n]$
			$\mid\ c \mid x \mid (v_1, ..., v_n) \mid \lambda(x : \tau).\, e$
expressions		e	$::= \textit{new}\ E(e_1, ..., e_n)$
			$\mid\ \textit{delete}\ e$
			$\mid\ A[e_1, ..., e_n]$
			$\mid\ e.p.op(e_1, .., e_n)$
			$\mid\ e.k$
			$\mid\ \textit{all}\ E$
			$\mid\ \textit{entries}\ p$
			$\mid\ \textit{yield} \mid \textit{flush}$
			$\mid\ v \mid e_1\, e_2 \mid e_1; e_2 \mid (e_1, ..., e_n)$
program		$program$	$::= \mathcal{S}; e$

Fig. 4. Syntax of types, schemas, and expressions. A subscript n without an explicit bound is assumed $n \geqslant 0$.

where all operations on x are replaced with operations on the array value:

$$x.op(e_1, ..., e_n) \;\equiv\; x[\,].value.op(e_1, ..., e_n)$$

The syntax of expressions is separated into values v and expressions e to facilitate the description of the evaluation semantics. Values can be regular values such

as literals c, variables x, products of values, or lambda expressions. Moreover, we have array and entity values which encode a particular entry of an array, as $A[v_1, ..., v_n]$, or a particular entity as $E[uid, v_1, ..., v_n]$. The entity value is not an expression that users can write down themselves and only occurs in the evaluation semantics as the result of a *new* expression (which also supplies the unique id uid for the entity value).

Expressions consist of both cloud specific expressions, and of regular expressions like applications $e_1\, e_2$, sequence $e_1; e_2$, products and lambda expressions. The keywords *new* and *delete* respectively create and delete entities. The expression $A[e_1, ..., e_n]$ is used to index into an array. The operation expression $e.p.op(e_1, ..., e_n)$ invokes an update or query operation op on a property p indexed by e. The creation keys of an entity, or the indices of an array expression can be queried using the $e.k$ expression.

The *all* and *entries* keywords return all elements of an entity or all non-initial entries of a property respectively. These primitive expressions allow us to construct general queries. Finally, the *yield* and *flush* operations are used for synchronization with the cloud.

Figure 5 defines a type system for our expression language. A derivation $\mathcal{S}, \Gamma \vdash e : \tau$ states that for a certain (well-formed) schema $\mathcal{S}$ and type environment Γ, the expression e is well-typed with a type τ. The initial Γ is written as Γ_0 and contains the type of primitive functions (i.e. $add : (\textsf{Int}, \textsf{Int}) \rightarrow \textsf{Int}$), together with the types of primitive cloud type operations (i.e. $\textsf{CInt}.add : (\textsf{Int}) \rightarrow \textsf{Unit}$).

Most rules are standard and self-explanatory. There are some important details though. In particular, in the type rule for operation expressions, we can see that the type ω of the mutable cloud value never 'escapes': values with a cloud type ω are not first-class and expressions always have a type τ (which does not include ω). This is by construction since an operation expression $e.p.op(e_1, ..., e_n)$ always occurs as a bundle and the cloud value never occurs in isolation.

3.1 Client Execution

Figure 6 and 7 give the evaluation semantics for local client execution. Figure 6 defines the evaluation order within an expression. An execution context $\mathcal{E}$ is an expression "with a hole $\square$", and we use the notation $\mathcal{E}[\![e]\!]$ to denote the expression obtained from $\mathcal{E}$ by replacing the hole with e. Essentially, the execution context acts as an abstraction of a program counter and specifies where the next evaluation step can take place.

Figure 7 defines the operational semantics in the form of transition rules $e; \sigma \rightarrow e'; \sigma'$ where an expression e with a local state σ is evaluated to a new expression e' and updated local state σ'. The client state σ is the state of the schema fork-join automaton $\Sigma^{\mathcal{S}}$ described in Section 5.

The first three rules, *new*, *delete*, and operation expressions just update the local state by invoking the corresponding updates on the fork-join automaton. The following three query rules just return the result of executing the corresponding query on the fork-join automaton. The fresh uid for the create call is produced locally. We assume each client can generate such globally unique ids.

$$\frac{\textit{entity}\ E(k_1 : \iota_1, ..., k_n : \iota_n) \in \mathcal{S} \quad \mathcal{S}, \Gamma \vdash e_i : \iota_i}{\mathcal{S}, \Gamma \vdash \textit{new}\ E(e_1, ..., e_n) : E} \qquad \frac{\mathcal{S}, \Gamma \vdash e : E}{\mathcal{S}, \Gamma \vdash \textit{delete}\ e : \textbf{Unit}}$$

$$\frac{\textit{array}\ A[k_1 : \iota_1, ..., k_n : \iota_n] \in \mathcal{S} \quad \mathcal{S}, \Gamma \vdash e_i : \iota_i}{\mathcal{S}, \Gamma \vdash A[e_1, ..., e_n] : A} \qquad \frac{\textit{entity}\ E(...) \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash E[uid, v_1, ..., v_n] : E}$$

$$\frac{\mathcal{S}, \Gamma \vdash e : \iota \quad \textit{property}\ p : \iota \to \omega \in \mathcal{S} \quad \omega.op : (\tau_1, ..., \tau_n) \to \tau \in \Gamma \quad \mathcal{S}, \Gamma \vdash e_i : \tau_i}{\mathcal{S}, \Gamma \vdash e.p.op(e_1, ..., e_n) : \tau}$$

$$\frac{\textit{entity}\ E(...) \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash \textit{all}\ E : \textbf{Set}\langle E \rangle} \qquad \frac{\mathcal{S}, \Gamma \vdash e : E \quad \textit{entity}\ E(..., k : \iota, ...) \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash e.k : \iota}$$

$$\frac{\textit{property}\ p : \iota \to \omega \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash \textit{entries}\ p : \textbf{Set}\langle \iota \rangle} \qquad \frac{\mathcal{S}, \Gamma \vdash e : A \quad \textit{array}\ A[..., k : \iota, ...] \in \mathcal{S}}{\mathcal{S}, \Gamma \vdash e.k : \iota}$$

$$\frac{x : \tau \in \Gamma}{\mathcal{S}, \Gamma \vdash x : \tau} \qquad \frac{\mathcal{S}, (\Gamma, x : \tau_1) \vdash e : \tau_2}{\mathcal{S}, \Gamma \vdash \lambda(x : \tau_1).\, e \ : \tau_1 \to \tau_2}$$

$$\frac{\mathcal{S}, \Gamma \vdash e_1 : \tau_2 \to \tau \quad \mathcal{S}, \Gamma \vdash e_2 : \tau_2}{\mathcal{S}, \Gamma \vdash e_1\, e_2 : \tau} \qquad \frac{\mathcal{S}, \Gamma \vdash e_i : \tau_i}{\mathcal{S}, \Gamma \vdash (e_1, ..., e_n) : (\tau_1, ..., \tau_n)}$$

$$\frac{\mathcal{S}, \Gamma \vdash e_1 : \tau_1 \quad \mathcal{S}, \Gamma \vdash e_2 : \tau_2}{\mathcal{S}, \Gamma \vdash e_1; e_2 : \tau_2} \qquad \frac{}{\mathcal{S}, \Gamma \vdash \textit{yield} : \textbf{Unit}} \qquad \frac{}{\mathcal{S}, \Gamma \vdash \textit{flush} : \textbf{Unit}}$$

Fig. 5. Types of expressions

$$\begin{aligned}
\mathcal{E} ::= \ & \Box \\
| \ & \textit{new}\ E(v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n) \\
| \ & \textit{delete}\ \mathcal{E} \\
| \ & A[v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n] \\
| \ & \mathcal{E}.p.op(e_1, ..., e_n) \\
| \ & v.p.op(v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n) \\
| \ & \mathcal{E}.k \\
| \ & \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathcal{E}; e \\
| \ & (v_1, ..., v_i, \mathcal{E}, e_j, ..., e_n)
\end{aligned}$$

Fig. 6. Evaluation contexts

The final four rules are standard evaluation rules on the expressions and do not use the local state at all. Note that we chose to keep the creation keys of arrays and entities around explicitly in the value representation which makes the key selection a completely local operation. However, realistic implementations can use just the *uid* to represent entities and store the creation values in the local state (and similarly for arrays).

The operations *yield*, *flush*, and *barrier* cannot be described as local operations and are handled by the semantic rules defined over the clients and servers as shown in the next section.

$$
\begin{array}{lll}
\mathcal{E}[\![\textsf{\textit{new}}\, E(v_1,...,v_n)]\!];\sigma & \rightarrow & \mathcal{E}[\![E[uid,v_1,...,v_n]]\!];\sigma.\mathsf{create}_E(E[uid,v_1,...,v_n]) \quad \text{(fresh } uid\text{)} \\
\mathcal{E}[\![\textsf{\textit{delete}}\, E[uid,...]]\!];\sigma & \rightarrow & \mathcal{E}[\![()]\!];\sigma.\mathsf{delete}_E(uid) \\
\mathcal{E}[\![v.p.op_u(v_1,...,v_n)]\!];\sigma & \rightarrow & \mathcal{E}[\![()]\!];\sigma.\mathsf{update}_p(v,op_u(v_1,...,v_n)) \\
\\
\mathcal{E}[\![v.p.op_q(v_1,...,v_n)]\!];\sigma & \rightarrow & \mathcal{E}[\![\sigma.\mathsf{query}_p(v,op_q(v_1,...,v_n))]\!];\sigma \\
\mathcal{E}[\![\textsf{\textit{all}}\ E]\!];\sigma & \rightarrow & \mathcal{E}[\![\sigma.\mathsf{all}_E]\!];\sigma \\
\mathcal{E}[\![\textsf{\textit{entries}}\ p]\!];\sigma & \rightarrow & \mathcal{E}[\![\sigma.\mathsf{entries}_p]\!];\sigma \\
\\
\mathcal{E}[\![A[v_1,...,v_n].k_i]\!];\sigma & \rightarrow & \mathcal{E}[\![v_i]\!];\sigma \\
\mathcal{E}[\![E[uid,v_1,...,v_n].k_i]\!];\sigma & \rightarrow & \mathcal{E}[\![v_i]\!];\sigma \\
\mathcal{E}[\![(\lambda(x:\tau).e)\ v]\!];\sigma & \rightarrow & \mathcal{E}[\![e[v/x]]\!];\sigma \\
\mathcal{E}[\![v;e]\!];\sigma & \rightarrow & \mathcal{E}[\![e]\!];\sigma
\end{array}
$$

Fig. 7. Expression semantics

4 System Model and Distribution

In the previous section, we have established the local execution semantics of expressions. In this section, we present an operational whole-system model including multiple clients and an elastic server pool. We follow the general blueprint for modeling eventually consistent systems presented in [1], where we prove that to achieve eventual consistency, it is sufficient to enforce that all executions produce proper revision diagrams, and that we use proper fork and join functions to manage the state of replicas.

Fig. 8(a) shows a brief example of an execution with three servers in the pool, and two clients. Clients that perform *yield* or *flush* initiate transitions of two kinds, push and pull. These transitions communicate with an eligible server in the pool. Not all servers are eligible, as we will explain shortly. Servers behave similarly to clients, initiating push and pull transitions with other eligible servers.

When synchronizing, clients and servers need to ensure that proper revision diagrams result. In particular, they must observe the join rule [1]: joiners must be downstream from the fork that forked the joinee (see Fig. 8b for examples). To ensure this condition, we assign round numbers to servers and clients, and use round maps (a form of vector clocks) to determine eligibility (by determining which forks are in the visible history). We show round numbers in Fig. 8(a). All clients and servers start with round 0, except the main revision that starts (and forever remains) in round 1.

We now proceed to give formal definitions of the ideas outlined above. We begin by introducing some notation to prepare for the operational rules in Fig. 9 and Fig. 10. We define a system configuration $\mathcal{C}$ to be a partial function from identifiers (representing servers or clients) to a server or client state, respectively. For a client identifier c, the client state $\mathcal{C}(c)$ is a tuple (r, e, σ) consisting of a round number r, an expression e, and the revision σ. For a server identifier s, the server state $\mathcal{C}(s)$ is a tuple (r, R, σ) consisting of a round number r, a round map R and a revision state σ.

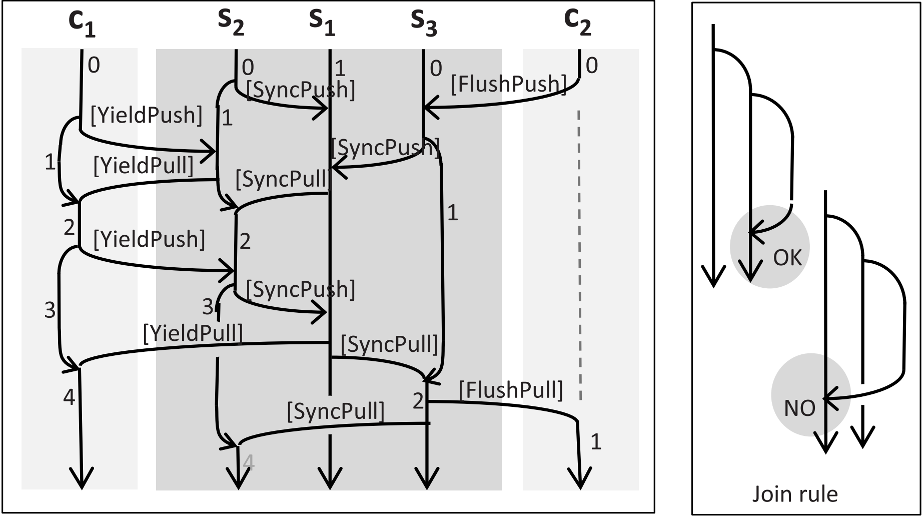


Fig. 8. **(a)** (left) An illustration of an execution with 2 clients and 3 servers. **(b)** (right) An illustration of the join rule.

The revision state σ represents the state of the current replica. We defer the description of the implementation of σ until Section 5, where we discuss cloud types and define fork-join automata. For now, we simply postulate that σ is in some set Σ, supports all the local data operations, has an initial state σ_0, and supports fork and join functions $fork : \Sigma \to \Sigma \times \Sigma$ and $join : \Sigma \times \Sigma \to \Sigma$, respectively. Moreover, we assume that $fork(\sigma_0) = (\sigma_0, \sigma_0)$ (forking from the initial state yields the initial state).

The round numbers r are used to track which clients (and servers) can synchronize with particular servers. After each fork, the round number of a client or server is incremented. The round map R on a server s is a total function that maps each identifier i of a client or server to a round number $R(i)$ which is the number of the last round whose fork is in the visible history of s. The initial round map R_0 maps all clients and servers to round 0 (since round 0 is always forked from the initial state of the main revision, it is retroactively in the visible history). The rules are set up to enforce that a client c (or server s) with round number r can only communicate with a server where $R(c) = r$.

Fig. 9 presents transition rules of the form $\mathcal{C} \Rightarrow \mathcal{C}'$ where cloud state $\mathcal{C}$ updates to $\mathcal{C}'$. We use the pattern match notation $\mathcal{C}(a_1 \mapsto b_1, ..., a_n \mapsto b_n)$ to match on a partial function $\mathcal{C}$ satisfying $\mathcal{C}(a_i) = b_i \ \forall i.1 \leqslant i \leqslant n$. We write $\mathcal{C}[a \mapsto b]$ to denote a partial function that is equivalent to $\mathcal{C}$ except that $\mathcal{C}(a) = b$.

For any cloud state there are potentially many valid transitions which capture the inherent concurrency and non-determinism of cloud execution. For example, clients can be spawned at any time using the rule [Spawn],and clients can arbitrarily interleave local evaluation.

The rules [Yield-Nop], [Yield-Push] and [Yield-Pull] describe how clients synchronize with servers. The [Yield-Nop] states that a *yield* instruction can be ignored,

$$[\textsc{Eval}]\ \frac{e;\sigma \rightarrow e';\sigma'}{\mathcal{C}(c \mapsto (r, e, \sigma)) \ \Rightarrow\ \mathcal{C}[c \mapsto (r, e', \sigma')]}$$

$$[\textsc{Spawn}]\ \frac{c \notin \mathsf{dom}(\mathcal{C})}{\mathcal{C} \ \Rightarrow\ \mathcal{C}[c \mapsto (0, e, \sigma_0)]}$$

[Yield-Push]

$$\frac{R(c) = r \quad R' = R[c \mapsto r+1] \quad \mathit{fork}(\sigma_c) = (\sigma'_c, \sigma''_c) \quad \mathit{join}(\sigma_s, \sigma'_c) = \sigma'_s}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\, c \mapsto (r, \mathcal{E}[\![\textit{yield}]\!], \sigma_c)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R', \sigma'_s),\, c \mapsto (r+1, \mathcal{E}[\![()]\!], \sigma''_c)]}$$

[Yield-Pull]

$$\frac{R(c) = r \quad R' = R[c \mapsto r+1] \quad \mathit{fork}(\sigma_s) = (\sigma'_s, \sigma''_s) \quad \mathit{join}(\sigma''_s, \sigma_c) = \sigma'_c}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\, c \mapsto (r, \mathcal{E}[\![\textit{yield}]\!], \sigma_c)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R', \sigma'_s),\, c \mapsto (r+1, \mathcal{E}[\![()]\!], \sigma'_c)]}$$

$$[\textsc{Yield-Nop}]\ \frac{}{\mathcal{C}(c \mapsto (r, \mathcal{E}[\![\textit{yield}]\!], \sigma)) \ \Rightarrow\ \mathcal{C}[c \mapsto (r, \mathcal{E}[\![()]\!], \sigma)]}$$

Fig. 9. Cloud evaluation rules for clients

$$[\textsc{Create}]\ \frac{s \notin \mathsf{dom}(\mathcal{C})}{\mathcal{C} \ \Rightarrow\ \mathcal{C}[s \mapsto (0, R_0, \sigma_0)]}$$

$$[\textsc{Sync-Push}]\ \frac{R_s(t) = r_t \qquad R'_s = \mathsf{max}(R_s, R_t) \qquad R''_s = R'_s[t \mapsto r_t + 1] \quad \mathit{fork}(\sigma_t) = (\sigma'_t, \sigma''_t) \quad \mathit{join}(\sigma_s, \sigma'_t) = \sigma'_s}{\mathcal{C}(s \mapsto (r_s, R_s, \sigma_s),\, t \mapsto (r_t, R_t, \sigma_t)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R''_s, \sigma'_s),\, t \mapsto (r_t + 1, R_t, \sigma''_t)]}$$

$$[\textsc{Sync-Pull}]\ \frac{R_s(t) = r_t \qquad R'_t = \mathsf{max}(R_s, R_t) \qquad R'_s = R_s[t \mapsto r_t + 1] \quad \mathit{fork}(\sigma_s) = (\sigma'_s, \sigma''_s) \quad \mathit{join}(\sigma''_s, \sigma_t) = \sigma'_t}{\mathcal{C}(s \mapsto (r_s, R_s, \sigma_s),\, t \mapsto (r_t, R_t, \sigma_t)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R'_s, \sigma'_s),\, t \mapsto (r_t + 1, R'_t, \sigma'_t)]}$$

$$[\textsc{Retire}]\ \frac{R_s(t) = r_t \quad R'_s = \mathsf{max}(R_s, R_t) \quad \mathit{join}(\sigma_s, \sigma_t) = \sigma'_s}{\mathcal{C}(s \mapsto (r_s, R_s, \sigma_s),\, t \mapsto (r_t, R_t, \sigma_t)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R'_s, \sigma'_s),\, t \mapsto \bot]}$$

Fig. 10. Cloud evaluation rules for servers

allowing disconnected clients to keep executing. The rule [Yield-Push] sends a revision to an eligible server, while the rule [Yield-Pull] receives a revision from an eligible server. In both cases, the round number of the client is incremented and the round map of the server is updated. The new states of the client and server are determined by forking/joining revisions appropriately (see Fig. 8).

Figure 10 shows the rules for server synchronization. The rules [Create] and [Retire] create and retire servers on demand. The [Sync] rule is the synchronization rule for servers and is like a simplified (more synchronous) version of [Yield]. The premise ensures that the round number matches, the round number is incremented, and the state is first joined and then forked again. What is different is that the round maps of both servers are also joined using $R = \mathsf{max}(R_s, R_t)$ (taking the pointwise max of the vector clocks).

[FLUSH-PUSH]

$$\frac{R(c) = r \qquad R' = R[c \mapsto r+1] \qquad join(\sigma_s, \sigma_c) = \sigma'_s}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\ c \mapsto (r, \mathcal{E}[\![\textsf{\textit{flush}}]\!], \sigma_c)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R', \sigma'_s),\ c \mapsto (r+1, \mathcal{E}[\![\textsf{\textit{block}}]\!], \sigma_c)]}$$

[FLUSH-PULL]

$$\frac{R(c^{\textsf{flush}}) = r \qquad fork(\sigma_s) = (\sigma'_s, \sigma'_c)}{\mathcal{C}(s \mapsto (r_s, R, \sigma_s),\ c \mapsto (r, \mathcal{E}[\![\textsf{\textit{block}}]\!], \sigma_c)) \ \Rightarrow\ \mathcal{C}[s \mapsto (r_s, R, \sigma'_s),\ c \mapsto (r, \mathcal{E}[\![()]\!], \sigma'_c)]}$$

$$[\text{COMMIT}]\ \frac{R' = R[\forall c.\ c^{\textsf{flush}} \mapsto R(c))}{\mathcal{C}(s_{\textsf{main}} \mapsto (0, R, \sigma)) \ \Rightarrow\ \mathcal{C}[s_{\textsf{main}} \mapsto (0, R', \sigma)]}$$

Fig. 11. Semantics of the *flush* operation

4.1 Flush

To describe the *flush* operation, we distinguish an initial main server $s_{\textsf{main}}$. The *flush* operation must not only guarantee that all updates of a client are joined in the main server $s_{\textsf{main}}$, but also that the client sees all the state changes in the main server that were applied before the client state was joined.

To track which client updates have been seen by the main server, we add an extra round number $c^{\textsf{flush}}$ in the round map. As shown in Figure 11, the main server can always execute the rule [COMMIT] to set the $c^{\textsf{flush}}$ entries to the corresponding round numbers of the clients that have synchronized with the main server. Through rule [SYNC] (Fig 10) any servers that synchronize with the main server will propagate these $c^{\textsf{flush}}$ entries automatically.

The rule [FLUSH-PUSH] is applied whenever the client does a *flush* operation. The rule is similar to [YIELD-PUSH] but blocks the client. Also, only the state of the client is joined with the server state, but the client state itself does not fork a new revision. The round map of the server s is updated though with the new round number $c \mapsto r+1$. Now, servers can execute [SYNC] until the state changes are propagated all the way up to $s_{\textsf{main}}$. At that point, the main server can make a [COMMIT] transition, making $c^{\textsf{flush}} \mapsto r+1$. After again doing more [SYNC] transitions, the new $c^{\textsf{flush}}$ entry makes it back to the original server. At this point, [FLUSH-PULL] can apply where the server state is forked now into a new server state σ'_s and client state σ'_c, and where the client is unblocked again.

4.2 Message Protocols and Server State

The rules presented are still somewhat more abstract than needed for an actual implementation, to keep the presentation from becoming too technical. In practice, all communication is asynchronous (based on message delivery) and unreliable. Thus, our actual implementation breaks synchronous transition rules (like Yield-Push, Yield-Pull, Sync, Flush-Pull, Flush-Push) into message protocols, uses state machines that are locally persisted, and retransmits messages if they are lost.

Another very important optimization concerns the size of messages. Sending the full replica state in messages is of course impractical. Thus we use compression by sending diffs of the state.

5 Cloud Types

We now examine our cloud type implementations in more detail. To this end we define the concept of a *fork-join automaton*. Fork-join automata are concrete implementations of cloud types, consisting of implementations for the abstract update and query operations, and concrete implementations of fork and join.

Definition 1. *A* fork-join automaton *(FJA) is a tuple* $(Q, U, \Sigma, \sigma_0, f, j)$ *where*

- Q *is an abstract set of query operations*
- U *is an abstract set of update operations*
- Σ *is a set of states*
- $\sigma_0 \in \Sigma$ *is the initial state*
- *Queries and updates have an interpretation as functions, specifically (1) each query operation* $q \in Q$ *defines a function* $q^{\#} : \Sigma \to Val$*, and (2) each update operation* $u \in U$ *defines a function* $u^{\#} : \Sigma \to \Sigma$.
- $f : \Sigma \to \Sigma \times \Sigma$ *is a function for splitting the current state on a fork.*
- $j : \Sigma \times \Sigma \to \Sigma$ *is a function for merging states on a join.*

Fork-join automata must satisfy a correctness conditions: they must correctly track and apply updates when revisions are forked and joined (as we illustrated earlier in Section 2.2). We discuss this condition only informally here, since its definition depends on the definition of revision diagrams, which is outside the scope of this paper. A full exposition is available in [1].

In the remainder of this section, we define a fork-join automaton for the entire cloud state (i.e. for all cloud data declared by the user). First, we define fork-join automata for the primitive cloud types **CInt** and **CString**. Then we show how to define the cloud types for entities and arrays. Finally, we show how to provide the cloud type **CSet** as syntactic sugar.

5.1 A Fork-Join Automaton for CInt

For cloud integers, we support operations get and set to read and write the current value, as well as add (Fig. 12). In the state, we store three values: a boolean indicating whether the current revision performed any set operations, a base value, and an offset. On fork, the boolean is reset, the base value is set to the current value, and the offset is set to zero. Add operations change only the offset, while set operations set the boolean to true, set the base value, and reset the offset. On join, we assume the value of the joined revision (if it performed a set) or add its offset (otherwise). This produces the desired semantics (see Section 2.2 for examples).

$$
\begin{aligned}
Q^{\mathsf{CInt}} &: \{\mathsf{get}\} \\
U^{\mathsf{CInt}} &: \{\mathsf{set}(n) \mid n \in \text{int}\} \cup \{\mathsf{add}(n) \mid n \in \text{int}\} \\
\Sigma^{\mathsf{CInt}} &: \text{bool} \times \text{int} \times \text{int} \\
\sigma_0^{\mathsf{CInt}} &: (\text{false}, 0, 0) \\
\mathsf{add}(n)^{\#}\ &(r, b, d) = (r, b, d + n) \\
\mathsf{set}(n)^{\#}\ &(r, b, d) = (\text{true}, n, 0) \\
\mathsf{get}^{\#}\ &(r, b, d) = b + d \\
f^{\mathsf{CInt}}\ &(r, b, d) = (r, b, d), (\text{false}, b + d, 0) \\
j^{\mathsf{CInt}}\ &(r_1, b_1, d_1)(r_2, b_2, d_2) = \begin{cases} (\text{true}, b_2, d_2) & \text{if } r_2 = \text{true} \\ (r_1, b_1, d_1 + d_2) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 12. Fork-join automaton for **CInt**

5.2 A Fork-Join Automaton for CString

For cloud strings, we support operations get and set to read and write the current value, and a conditonal operation setIfEmpty (Fig. 13). In the state, we record the current value and whether it has not been written ($\bot$), has been written (wr), or has been conditionally written (cond). A conditional write succeeds only if the current value is empty, and this test is repeated on merge.

5.3 A Fork-Join Automata for the Complete State

For a fixed schema $\mathcal{S}$, we can now define the entire state as a fork-join automaton. First, we define the query operations $Q^{\mathcal{S}}$ and the update operations $U^{\mathcal{S}}$ as in the following table.

$$
\begin{aligned}
Q^{\mathsf{CString}} &: \{\mathsf{get}\} \\
U^{\mathsf{CString}} &: \{\mathsf{set}(s) \mid s \in \text{string}\} \cup \{\mathsf{setIfEmpty}(s) \mid s \in \text{string} \setminus \{""\}\} \\
\Sigma^{\mathsf{CString}} &: \{\bot, \mathsf{wr}, \mathsf{cond}(\text{string})\} \times \text{string} \\
\sigma_0^{\mathsf{CString}} &: (\bot, "") \\
\mathsf{set}(s)^{\#}\ &(r, t) = (\mathsf{wr}, s) \\
\mathsf{setIfEmpty}(s)^{\#}\ &(r, t) = \begin{cases} (\mathsf{wr}, s) & \text{if } r = \mathsf{wr} \wedge t = "" \\ (\mathsf{cond}(\text{s}), s) & \text{if } r = \bot \wedge t = "" \\ (\mathsf{cond}(\text{s}), t) & \text{if } r = \bot \wedge t \neq "" \\ (r, t) & \text{otherwise} \end{cases} \\
\mathsf{get}^{\#}\ &(r, s) = s \\
f^{\mathsf{CString}}\ &(r, s) = (r, s), (\bot, s) \\
j^{\mathsf{CString}}\ &(r_1, s_1)(r_2, s_2) = \begin{cases} (\mathsf{wr}, s_2) & \text{if } r_2 = \mathsf{wr} \\ (\mathsf{wr}, s) & \text{if } r_1 = \mathsf{wr} \wedge s_1 = "" \wedge r_2 = \mathsf{cond}(s) \\ (\mathsf{cond}(s), s) & \text{if } r_1 = \bot \wedge s_1 = "" \wedge r_2 = \mathsf{cond}(s) \\ (\mathsf{cond}(s), s_1) & \text{if } r_1 = \bot \wedge s_1 \neq "" \wedge r_2 = \mathsf{cond}(s) \\ (r_1, s_1) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 13. Fork-join automaton for **CString**

operation	argument types	return type	entity/property definition
all_E		$\mathbf{Set}\langle E\rangle$	*entity* $E(k_1 : \iota_1, ..., k_n : \iota_n)$
$\mathsf{create}_E(e)$	E		*entity* $E(k_1 : \iota_1, ..., k_n : \iota_n)$
$\mathsf{delete}_E(e)$	E		*entity* $E(k_1 : \iota_1, ..., k_n : \iota_n)$
$\mathsf{entries}_p$		$\mathbf{Set}\langle\iota\rangle$	*property* $p : \iota \to \omega$
$\mathsf{query}_p(i, q)$	ι, Q^ω	Val	*property* $p : \iota \to \omega$
$\mathsf{update}_p(i, u)$	ι, U^ω		*property* $p : \iota \to \omega$

Next, we define the state space to consist of separate components for each entity type and each property

$$\Sigma^{\mathcal{S}} = \prod_{p\in\mathcal{S}} \Sigma_p \times \prod_{E\in\mathcal{S}} \Sigma_E.$$

For each declaration *property* $p : \iota \to \omega$ we store a total function from keys to values, where keys are of the corresponding index type, and values belong to the state space of the corresponding fork-join automaton:

$$\Sigma_p = \iota \to \Sigma^\omega$$

For each declaration *entity* $E(k_1 : \iota_1, ..., k_n : \iota_n)$ we store a total function from entities to a state that indicates whether this entity is not yet created ($\bot$), exists as a normal entity (ok), or has been deleted ($\top$):

$$\Sigma_E = E \to \{\bot, ok, \top\}$$

For a state $\sigma \in \Sigma^{\mathcal{S}}$, we let σ_p and σ_E be the projection on the respective components.

Naturally, in the initial state $\sigma_0^{\mathcal{S}}$, we map all property indexes to Σ_0^ω (the initial state of the corresponding fork-join automaton) and all entities to $\bot$. We show the implementation of queries, updates, fork, and join in Fig. 14, using pseudocode, and explain them in the following.

- Create adds a fresh element to an entity by mapping it to ok. We assume each client can create fresh elements (based on a local id and counter).
- Delete maps the deleted element to $\top$ to mark it as deleted. We cannot simply remove it because at joins, it would be impossible to determine if one side is fresh, or the other deleted. Extra book-keeping can be used to eventually collect these tombstones.
  Deletion also causes any dependent entities to be deleted. This is achieved by Propagate. Note that entity dependencies cannot be cyclic, since an entity can only be used in the creation of another when it is already defined.
- all_E returns all non-deleted values of a given entity.
- A query q on an entry i of property p is answered by delegating it to the FJA of p at i, provided that i is not deleted.
- Similarly, an update u on an entry i of property p is delegated to the FJA of p at i, provided i is not deleted.
- $\mathsf{entries}_p$ returns all the entries of a property p that map to non-default FJAs and are not deleted.

– Forking the overall FJA turns into a point-wise forking of all the FJA's of each property. The entity maps are unaffected by forking.
– Joining is similarly performed point-wise on all properties. For entities, joining requires computing the maximum in the order $\bot < ok < \top$. This achieves deleting the entry, provided any one side has it deleted, or keeping it allocated, if any one side has it allocated. At joins, we also need to repropagate deletions to all dependent elements, as new deletions can be merged into the revision.

It is remarkable that the complete state FJA operations are commutative by themselves. The only non-commutative operations are in the FJAs implementing cloud types. This property makes using arrays and entities very natural and does not introduce unexpected conflict resolutions. Furthermore, our design was careful to enable a completely modular implementation of the complete state FJA with respect to the cloud type implementations. In part, this structure makes a single parameterized, reusable implementation of cloud storage and synchronization possible. Any schema and any extensions of cloud types can be supported without further changes.

5.4 Implementation of **CSet**

Rather than defining sets directly, we encode them relationally, building on the abstraction mechanism provided by entities. Given a schema definition for a property of type **CSet**$\langle\iota\rangle$, we rewrite it to an entity definition whose entities represent "instances" of additions of elements:

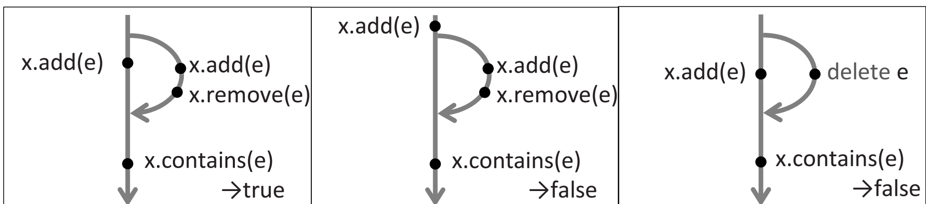
property p: $\iota' \rightarrow$ **CSet**$\langle\iota\rangle \quad \equiv \quad$ *entity* E_p[index : ι', element : ι]

Then we encode operations as follows:

x.add(i) $\equiv$ { *new* $E_p(x, i)$; }
x.contains(i) $\equiv$ { *return* (*all* E_p *where* index == x and element == i).isNotEmpty() ; }
x.remove(i) $\equiv$ { *foreach* (e *in all* E_p *where* index == x and element == i) e.*delete*() ; }

Our indirect encoding has two advantages (illustrated in the picture below):

– Removing an element from a set only removes instances that were visibly added before the remove. This is known as observed-remove behavior, as proposed in [13] as a reasonable semantics for eventually consistent sets (see examples on the left and in the middle below).
– If the user deletes an entity, that entity disappears automatically from all sets that contain it (see example on the right below).



```
// operations on entities

create_E(e) {
    σ_E(e) := σ_E(e)[e ↦ ok] ;
}
delete_E(e) {
    σ_E(e) := σ_E(e)[e ↦ ⊤] ;
    propagate() ;
}
all_E {
    return {e ∈ E | σ_E(e) = ok} ;
}

// auxiliary functions

propagate() {
    while exists E, e such that
      σ_E(e) ≠ ⊤ and deleted(e)
    do
      σ_E(e) := σ_E(e)[e ↦ ⊤] ;
}
deleted(i) {
    match i with
    A[i_1, ..., i_n]:
        return (exists j such that deleted(i_j)) ;
    E[uid, i_1, ..., i_n]:
        return σ_E(i) = ⊤
              or (exists j such that deleted(i_j)) ;
    else // string or int
        return false ;
}
isdefault(σ) {
    if σ ∈ Σ^CInt
        return get# σ = 0 ;
    else if σ ∈ Σ^CString
        return get# σ ="" ;
    else if σ ∈ Σ^CSet⟨ι⟩
        return elems# σ = ∅ ;
}

// operations on properties

query_p(i, q) {
    if (deleted(i))
      return ⊥ ;
    else
      return σ_p(i).q ;
}
update_p(i, u) {
    if (not deleted(i))
      σ_p(i).u ;
}
entries_p {
    return all i ∈ ι
      where (not isdefault(σ_p(i))
        and (not deleted(i))
}

// fork and join functions

fork() {
    var σ' = σ ; // copy the state
    foreach property p : ι → ω
      foreach i ∈ ι
        (σ_p(i), σ'_p(i)) := f^ω(σ_p(i)) ;
    return σ' ;
}

join(σ') {
    foreach property p : ι → ω
      foreach i ∈ ι
        σ_p(i) := j^ω(σ_p(i), σ'_p(i)) ;
    foreach entity E(k_1 : ι_1, ..., k_n : ι_n)
      foreach e ∈ E
        σ_E(e) := max(σ_E(e), σ'_E(e))) ;
    propagate() ;
}

max(s_1, s_2) uses the order ⊥ < ok < ⊤
```

Fig. 14. Complete fork-join automaton

6 Related Work

At the heart of our work is the idea of using revision diagrams and fork-join automata to achieve eventual consistency, which was introduced in [1]. In this paper we extend and concretize this idea, by (1) devising a composable way to

construct schema from basic cloud types, which eliminates the need for user-defined conflict resolution code, (2) giving examples of concrete programs and cloud types, (3) devising primitives that are sufficient to recover stronger synchronization. Moreover, we provide a formal syntax and semantics that connects a small, but sufficiently expressive programming language with a detailed operational system model.

Eventual consistency is motivated by the impossibility of achieving strong consistency, availability, and partition tolerance at the same time, as stated by the CAP theorem [5]. Eventual consistency across the literature uses a variety of techniques to propagate updates (e.g. general causally-ordered broadcast [14,15], or pairwise anti-entropy [10]). For a general high-level comparison of our work with various notions of eventual consistency appearing in the literature, we refer to the discussion in [1].

Most closely related to our work are conflict-free replicated data types (CRDTs) [14] and Bayou's weakly consistent replication [16].

- CRDTs are very similar to our cloud types, insofar that they separate the use of eventually consistent data types from their implementation. In fact, CRDTs can serve as cloud types (as exemplified by the observed-remove set proposed in [13]). However, we are not aware of prior work on how to compose individual CRDTs into a larger schema. Furthermore, CRDTs only support commutative operations, whereas our approach supports non-commutative operations while still achieving eventual consistency. Furthermore, we support stronger synchronization primitives like *flush* when necessary, in the same framework.
- In Bayou [16], and in the original Concurrent Revisions work[2], conflict resolution is achieved by explicit merge functions written by the user. In contrast, this paper uses conflict resolution that is *automatically inferred* from the structure of the type declarations.

Research on *persistent data types* [8] is related to our definition of cloud types insofar it concerns itself with efficient implementations of data types that permit retrieval and mutations of past versions. However, it does not concern itself with apects related to transactions or distribution.

Prior work on *operational transformations* [15] can be understood as a specialized form of eventual consistency where updates are applied to different replicas in different orders, and modified in such a way as to guarantee convergence. This specialized formulation can provide highly efficient broadcast-based real-time collaboration, but poses significant implementation challenges [7].

There is of course a large body of work on transactions. Most academic work considers strong consistency (serializable transactions) only, and is thus not directly applicable to eventual consistency. Nevertheless there are some similarities, such as:

- [6] provides insight on the limitations of serializable transactions, and proposes similar workarounds as used by eventual consistency (timestamps and commutative updates). However, transactions remain tentative during disconnection.

– Snapshot isolation [4] relaxes the consistency model, but transactions can still fail, and can not commit in the presence of network partitions.
– Automatic Mutual Exclusion [9], like our work, uses *yield* statements to separate transactions.

7 Conclusion

Providing good programming abstractions for cloud storage, synchronization, and disconnected operation appears crucial to accelerate the production of useful and novel applications on today's and tomorrow's mobile devices. In this paper, we provided a sound foundation upon which to build such programming abstractions through the use of automatically synchronized cloud data types that can be composed into a larger data schema using indexed arrays and entities. The design we presented allows implementing all the difficult parts of such a system (the cloud service, the local persistence, the caching, the conflict resolution, and the synchronization) once and for all, while guaranteeing eventual consistency. An application programmer declares only the data schemas and focuses on writing code performing operations on the data, as well as identifying points in his program where synchronization is desired.

References

1. Burckhardt, S., Leijen, D., Fähndrich, M., Sagiv, M.: Eventually Consistent Transactions. In: Seidl, H. (ed.) Programming Languages and Systems. LNCS, vol. 7211, pp. 67–86. Springer, Heidelberg (2012)
2. Burckhardt, S., Leijen, D.: Semantics of Concurrent Revisions. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 116–135. Springer, Heidelberg (2011)
3. Chen, P.P.-S.: The entity-relationship model toward a unified view of data. ACM Trans. Database Syst. 1, 9–36 (1976)
4. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. 30(2), 492–528 (2005)
5. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33, 51–59 (2002)
6. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. SIGMOD Record 25, 173–182 (1996)
7. Imine, A., Rusinowitch, M., Oster, G., Molli, P.: Formal design and verification of operational transformation algorithms for copies convergence. Theoretical Computer Science 351, 167–183 (2006)
8. Kaplan, H.: Persistent data structures. In: Handbook on Data Structures and Applications, pp. 241–246. CRC Press (1995)
9. Martin, A., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: Principles of Programming Languages (POPL), pp. 63–74 (2008)
10. Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Demers, A.: Flexible update propagation for weakly consistent replication. Operating Systems Review 31, 288–301 (1997)

11. Saito, Y., Shapiro, M.: Optimistic replication. ACM Computing Surveys 37, 42–81 (2005)
12. Shapiro, M., Kemme, B.: Eventual consistency. In: Encyclopedia of Database Systems, pp. 1071–1072. Springer (2009)
13. Shapiro, M., Preguica, N., Baquero, C., Zawirski, M.: A comprehensive study of convergent and commutative replicated data types. Technical Report Rapport de recherche 7506, INRIA (2011)
14. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-Free Replicated Data Types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 386–400. Springer, Heidelberg (2011)
15. Sun, C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms, and achievements. In: Conference on Computer Supported Cooperative Work, pp. 59–68 (1998)
16. Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., Hauser, C.: Managing update conflicts in bayou, a weakly connected replicated storage system. SIGOPS Oper. Syst. Rev. 29, 172–182 (1995)
17. Tillmann, N., Moskal, M., de Halleux, J., Fähndrich, M.: Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In: ONWARD 2011 at SPLASH (also available as Microsoft TechReport MSR-TR-2011-49) (2011)