Rethinking Safe Consistency in Distributed Object-Oriented Programming

MIRKO KÖHLER, Technische Universität Darmstadt, Germany
NAFISE ESKANDANI, Technische Universität Darmstadt, Germany
PASCAL WEISENBURGER, Technische Universität Darmstadt, Germany
ALESSANDRO MARGARA, Politecnico di Milano, Italy
GUIDO SALVANESCHI, Universität St. Gallen, Switzerland

Large scale distributed systems require to embrace the trade off between consistency and availability, accepting lower levels of consistency to guarantee higher availability. Existing programming languages are, however, agnostic to this compromise, resulting in consistency guarantees that are the same for the whole application and are implicitly adopted from the middleware or hardcoded in configuration files.

In this paper, we propose to integrate availability in the design of an object-oriented language, allowing developers to specify different consistency and isolation constraints in the same application at the granularity of single objects. We investigate how availability levels interact with object structure and define a type system that preserves correct program behavior. Our evaluation shows that our solution performs efficiently and improves the design of distributed applications.

CCS Concepts: • **Software and its engineering → Distributed programming languages**; • **Information systems** → *Remote replication.*

Additional Key Words and Phrases: replication, consistency, type systems, Java

ACM Reference Format:
Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. 2020. Rethinking Safe Consistency in Distributed Object-Oriented Programming. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 188 (November 2020), 30 pages. https://doi.org/10.1145/3428256

1 INTRODUCTION

Replication is fundamental in large-scale distributed systems to improve data availability and data access latency. By replicating data at multiple sites, distributed components can interact with the closest (e.g., local) replica without incurring the cost of remote data access. However, synchronizing replicas to ensure a globally consistent view implies onerous performance costs that can hamper rather than improve availability and latency. For this reason, weakly consistent models have been widely adopted to limit synchronization overhead at the expense of correctness [Vogels 2009].

Still, some operations, for instance those related to security, monetary transactions, and accountability, have higher correctness requirements and need strong consistency. One option is to build the application on top of a highly consistent data store and accept the overhead of strong

Authors' addresses: Mirko Köhler, Technische Universität Darmstadt, Darmstadt, Germany, koehler@cs.tu-darmstadt.de; Nafise Eskandani, Technische Universität Darmstadt, Darmstadt, Germany, n.eskandani@cs.tu-darmstadt.de; Pascal Weisenburger, Technische Universität Darmstadt, Darmstadt, Germany, weisenburger@cs.tu-darmstadt.de; Alessandro Margara, Politecnico di Milano, Milano, Italy, alessandro.margara@polimi.it; Guido Salvaneschi, Universität St. Gallen, St. Gallen, Switzerland, guido.salvaneschi@unisg.ch.

2475-1421/2020/11-ART188
https://doi.org/10.1145/3428256

consistency. Alternatively, developers can exploit weakly consistent data stores and implement strong consistency manually for some operations. Because of the interaction of data at different consistency levels, however, this is a complex and error-prone task. To address these issues, recent research investigates APIs for data stores that enable setting different consistency models for different data, offering high availability and low latency when possible, and strong consistency when needed. These solutions are based on various approaches, including embedded DSLs for transactions [Milano and Myers 2018], specialized ADTs that support different consistency levels [Holt et al. 2016; Li et al. 2012] and function preconditions and invariants [Balegas et al. 2015; Houshmand and Lesani 2019; Sivaramakrishnan et al. 2015].

We propose ConSysT, a language where consistency and availability are integrated with the object-oriented programming model. We enable mixing objects with different availability characteristics and investigate how availability interacts with object abstractions, such as fields, references, object nesting, encapsulation, and mutable state.

ConSysT's integration of consistency and object-oriented programming builds on two major insights. (i) In line with the recent research on distributed systems [Bailis et al. 2013], ConSysT considers consistency and isolation concerns *together*. (Single-object) consistency constrains how updates are propagated across replicas and become visible to the other components of the distributed application. Isolation determines how concurrent invocations of methods (blocks of operations) on replicated objects interleave. Consistency and isolation together concur in determining data availability and access latency. (ii) ConSysT adopts a type system that ensures that the program does not violate the correctness constraints set by the developer to enable integrating availability and object-oriented programming in a safe way. Our evaluation shows that ConSysT enables developers to pay a performance overhead only where consistency is needed. It improves the design of distributed applications and prevents consistency programming errors. In summary, this paper makes the following contributions:

- We propose ConSysT, a distributed language that combines multiple availability levels with object-oriented programming, enabling mixing objects with different consistency and isolation guarantees.
- We design a type system that ensures that consistency constraints are not violated due to unsafe mixing. A core calculus for ConSysT and the correctness proofs for the type system are in the supplementary material.
- We provide a reference implementation of ConSysT as a Java extension and a middleware for replicated objects.
- We evaluate ConSysT with case studies and benchmarks, showing that it improves software design and increases performance over traditional coarse-grained consistency selection mechanisms.

The implementation of ConSysT is publicly available, together with the case studies and the benchmarks discussed in this paper.[1]

The paper is structured as follows. Section 2 outlines the context of our work. Section 3 presents the design of ConSysT and Section 4 introduces its formalization. Section 5 describes the implementation. Section 6 presents the evaluation. Section 7 discusses related work and Section 8 concludes.

2 CONSISTENCY AND ISOLATION

This section discusses consistency and isolation in distributed systems using TicketShop, a distributed shop application for concert tickets, as a running example. In TicketShop, the objects modeling concerts are replicated to the machines of various local retailers enabling them to retrieve and display data related to concerts with low delay whenever possible.

[1] https://consyst-project.github.io/

```
class Band {
 String bandName;
}
class ConcertHall {
 int maxAudience;
}
class Counter {
 int value = 0;
 void inc() { value++; }
}

class Concert {
 Date date;
 ConcertHall hall;
 Band band;
 Counter soldTickets;

 int getSoldTickets() {
  return soldTickets.value;
 }

 Optional<Ticket> buyTicket() {
  if (hall.maxAudience >
   getSoldTickets()) {
   soldTickets.inc();
   return Optional.of(new Ticket());
  } else {
   return Optional.empty();
}}}
```

(a) Java implementation.

```
class Band {
 String bandName;
}
class ConcertHall {
 int maxAudience;
}
class Counter {
 int value = 0;
 void inc() { value++; }
}

class Concert {
 Date date;
 Ref<@Low ConcertHall> hall;
 Ref<@High Band> band;
 Ref<@Low Counter> soldTickets;

 int getSoldTickets() {
  return soldTickets.ref.value;
 }

 Optional<Ticket> buyTicket() {
  if (hall.ref.maxAudience >
   getSoldTickets()) {
   soldTickets.ref.inc();
   return Optional.of(new Ticket());
  } else {
   return Optional.empty();
}}}
```

(b) ConSysT implementation.

Fig. 1. TicketShop: example of a distributed application with replicated objects.

Figure 1 shows part of the implementation of TicketShop in Java (Figure 1a) and in our ConSysT language (Figure 1b). For now, we refer to the Java implementation, while the following sections will introduce the features of ConSysT. Figure 1a defines a `Band` class that, for simplicity, only includes a `bandName` field with the name of the band, a `ConcertHall` class, with a `maxAudience` field indicating the capacity of the room, and a `Counter` with an integer value as well as an `inc()` method to increment the value by one. A `Concert` has a date, a hall, a band, and the number of tickets sold so far. Method `buyTicket` lets a customer buy a ticket as long as there are tickets available, that is, if the number of sold tickets is lower than the capacity of the concert hall. If that is the case, then the number of sold tickets is increased by one and a new ticket is returned. Otherwise the method returns no ticket.

2.1 Issues with Replicated Objects

Now assume that `Band`, `ConcertHall`, `Counter`, and `Concert` objects are replicated at multiple machines, which can interact with their copies of the objects simultaneously. This simple application is enough to demonstrate pitfalls of working with replicated data, as explained in the following.

Fine-Grained Consistency. If the number of tickets sold for a concert (`soldTickets`) is replicated to multiple processes, the application might diverge from the expected behavior if different replicas can see the operations performed on `soldTickets` in different orders. For example, assume that only one ticket for a concert is left, but there are two users that buy a ticket at two different replicas at the same time. The system has to ensure that only one ticket is sold, and that all replicas agree on which user gets the ticket, i.e., who tries to buy the ticket first. In other words, the program requires (the order of operations performed on) `soldTickets` to be *consistent* across all replicas.

However, such strong consistency is not needed for all replicated objects. For example, in the case of the `band` field, it is acceptable if a change to the band is not propagated to other replicas immediately. It is only required that it is propagated *eventually*. Unfortunately, if the consistency level is hardcoded in the data store, to ensure correctness, programmers need to adopt the highest consistency level for all values, facing significant performance costs also for values that could be replicated with a lower consistency level.

Isolation. Even under the assumption that `soldTickets` is *consistent*, method `buyTicket()` remains problematic. There is no guarantee that the value returned by `getSoldTickets()` is still up-to-date when `soldTickets.inc()` is called. Other processes can cause concurrent changes of `soldTickets` between these two calls. The example shows that the application does not behave correctly without some form of *isolation* of method invocations, which defines how concurrent invocations can interleave with each other. In our example, method `buyTicket()` executes correctly only if no concurrent update to `soldTickets` is allowed. Also, the example shows that consistency alone is hardly useful to define correct application and it is only meaningful to programmers when combined with some form of isolation.

Consistency Leaks. The `buyTicket()` method presents yet another potential source of errors in the comparison between `getSoldTickets()` and `maxAudience`. As described above, the `soldTickets` value has to be consistent across all replicas, but such requirement might not hold for `maxAudience`, i.e., `maxAudience` can differ between replicas, possibly allowing some of them to sell tickets while others cannot. In fact, mixing a strongly consistent value like `getSoldTickets()` and a weakly consistent value like `maxAudience` lowers the guarantees that strong values provide.

2.2 Executive Summary

In summary, to address the issues above, we need:

(1) A mechanism to define consistency at the granularity of individual objects to ensure correctness for strong consistent objects and accept lower guarantees to enhance availability for weak consistent objects.
(2) As strong consistency provides little guarantees in practice if not combined with some form of isolation, we need a mechanism to specify isolation of multiple operations performed on replicated objects.
(3) Since objects with different consistency guarantees can coexist, the system has to ensure that strong guarantees are not violated by wrongfully mixing data with different levels of consistency.

In the rest of the paper, we propose a language that supports replicated objects with different consistency and isolation properties, and a type system that prevents their violation.

3 CONSYST REPLICATED OBJECTS

This section presents the design of ConSysT by reimplementing the TicketShop example of Section 2. We present the system model in Section 3.1, and then we incrementally introduce the features of ConSysT: replicated objects (Section 3.2), availability levels (Section 3.3) and the availability type system (Section 3.4).

3.1 System Model

Figure 2 shows an overview of our system model. Distributed applications run on multiple *processes*, which are (logically) single-threaded and can be deployed on different machines. Applications create and access *replicated objects*. Conceptually, each process keeps its own replicas of all replicated objects in the system. Applications interact with replicated objects by performing *operations*: method

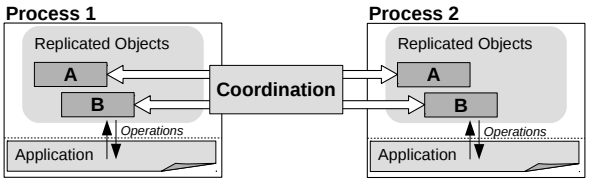

Fig. 2. System model.

invocations, field accesses (reads) and modifications (writes). Each replicated object has associated consistency and isolation guarantees. When an application performs operations involving one or more replicated objects, the system runs a coordination phase that depends on the specific guarantees of the objects.

3.2 Replicated Objects

ConSysT introduces a `replicate` method to create new replicated objects. For example, the following code snippet shows how the TicketShop application creates a new replicated object of type `Concert`:

```
1 Ref<Band> band = ...
2 Ref<ConcertHall> hall = ...
3 Ref<Concert> concert1 = replicate("c1", new Concert(date(...),hall,band));
```

Method `replicate` returns a reference `Ref` to the new replicated object. Other processes can reference the object by its global name `"c1"`, specified at creation time.

An overload of `replicate` creates replicated objects without a name. Such objects can only be referenced through fields of other replicated objects. For example, the following code snippet creates a new counter that can only be referenced through the variable `soldTickets`.

```
Ref<Counter> soldTickets = replicate(new Counter(0));
```

Replicating an object creates a deep copy of the original object, ensuring referential integrity. References to replicated objects are preserved. The `lookup` method performs a lookup of an existing replicated object by its global name. For example, a client can reference an existing concert by looking up its name:

```
Ref<Concert> concert1 = lookup("c1");
```

The reference points to the replica that is located on the process. Applications can perform operations – method invocation, field access/modification – on replicated objects as in normal Java. For example, a client can buy a ticket by invoking `buyTicket()` on the replicated `Concert` object:

```
Optional<Ticket> t = concert1.ref.buyTicket();
```

The interaction with a replicated object is always denoted by `ref`. In this way, ConSysT makes performing operations on replicated objects explicit and avoids distribution transparency [Collet et al. 2007]. As in Java, method invocations are synchronous. Accessing a field is similar to a method invocation. The date field of the concert can be accessed with `ref` as well:

```
1 Date d = concert1.ref.date;
2 concert1.ref.date = date("2019-09-28");
```

The result of a field access or method call is a copy of the value in the replicated object. Mutating object `d` of the code snippet does not mutate field `date` stored in the replicated object `concert1`. The same holds for arguments in method invocations, which are copied before the method is executed. As before, references to replicated objects are preserved. Copy semantics ensure that replicated objects do not hold references to objects that are local to some process, while they can hold references to other replicated objects.

3.3 Availability of Replicated Objects

ConSysT associates each replicated object with an *availability level*, and supports mixing replicated objects with different availability levels. For simplicity, in the rest of this section we consider two availability levels, *High* and *Low* as representative of two classes: *High* for levels that do not require blocking coordination among processes; *Low* requires blocking coordination to implement consistency and isolation.

For example, a *High* level could provide eventual consistency with Item Cut Isolation [Bailis et al. 2013]. Eventual consistency applies changes locally (with no blocking coordination) and eventually propagates them to all replicas. It resolves conflicts with some deterministic strategy to ensure convergence. Item Cut Isolation guarantees that when an operation reads a value multiple times, all reads return the same value. It can be implemented without blocking coordination by locally caching read data until the operation is finished. A *Low* availability level could provide sequential consistency with serializable isolation, which together guarantee that all replicas observe the effects of operations as if they were executed in some serial order without interleaving. Sequential consistency requires some form of coordination, e.g., a centralized component to serialize all accesses to a given object or a consensus algorithm, such as Paxos [Lamport et al. 2001]. Serializability can be achieved using distributed locking protocols such as two-phase locking [Eswaran et al. 1976], which prevent concurrent modifications of replicated objects, or using optimistic concurrency control, which restores a previous version of the data in the case concurrent operations corrupt the state.

3.3.1 Availability Levels and Object Structure. ConSysT guarantees that data with different levels is not combined in a way that violates the respective availability guarantees. ConSysT is oblivious of the specific semantics of each level. From the perspective of the programming model, the system only needs to know the partial order relation that defines when an availability level l_1 is stronger than another level l_2, i.e., that l_1 offers at least all the guarantees of l_2 (and possibly more). In general, availability levels form a lattice. We define the availability level `Local` (for objects that are not replicated) and `Inconsistent` as the $\bot$ and $\top$ of the lattice. Lattices for single-object consistency models have been given by Viotti and Vukolić [2015], and for transactional and single-object consistency by Bailis et al. [2013].

In ConSysT, the availability level of a replicated object is defined at object creation. The following code snippet exemplifies how availability shows up *in the type* of a replicated object.

```
1 Ref<@High Concert> concert1 = replicate("c1",new Concert(date(...),hall,band));
2 Ref<@High Concert> concert1 = lookup("c1");
```

In these cases, the ConSysT runtime checks dynamically if the availability levels of references to the same replicated objects match and throws an error in case of a mismatch.

ConSysT replicated objects can hold fields that are `Refs` to other replicated objects and hence can have different availability types. For simplicity, the availability level of non-`Ref` fields of a replicated object is the same as the availability level of the object that holds the field. The rationale of this approach is that fields that are not `Refs` themselves are local to the object and one needs to access them through a remote reference of the containing object. Figure 1b reports the definition of the `Concert` class using availability type annotations. To ensure that the field `soldTickets` is strongly consistent (i.e. low available), as required in the ticket shop application, the developer annotates the reference to `soldTickets` with the *Low* availability level (Line 16). The field `band` has a *High* availability level to prevent incurring latency overhead. `hall` cannot be *High* as will be explained in Section 3.4.2. `date` inherits the same availability level that is used to instantiate the `Concert` object. In summary, by explicitly defining the availability level, ConSysT ensures that the guarantees for the annotated objects are satisfied. This approach solves the *Fine-grained consistency* problem

from Section 2.1: the programmer can make accurate decisions about which data has *High* or *Low* consistency levels.

3.3.2 Operations on Replicated Objects. ConSysT classifies operations based on the availability of the receiving object. *High operations* and *Low operations* are performed on *High* and *Low* replicated objects, respectively. In the following example, the operation that updates the field `date` is *High* as `concert1` is a reference to a *High* object:

```
Ref<@High Concert> concert1 = concert1.ref.date = date("2019-09-28");
```

In the rest, without loss of generality, *High* operations are not isolated whereas *Low* operations are executed with serializable isolation. For example, method `buyTicket()` has to be executed in isolation to prevent concurrency bugs. This is accomplished by invoking it on a *Low* object:

```
Ref<@Low Concert> concert = concert.ref.buyTicket();
```

Note that the isolation level of a method always depends on the receiver object. *Low* operations that occur during the execution of a *High* method run at the *Low* availability level, which provides stronger guarantees. Our approach to isolation is similar to the `synchronize` keyword in Java where an object manages its own concurrent modifications. With the design above, we solve the isolation problem from Section 2.1 by providing isolation guarantees together with consistency guarantees for objects.

References to replicated objects (annotated with their availability level) can be passed as method parameters or return values. To demonstrate, consider, the method `copyDate` below, which sets the `date` field to the date of another (*High* available) concert. The availability annotation ensures that only references that have at least the consistency guarantees of the availability level *High* are passed as method parameter.

```
void copyDate(Ref<@High Concert> concert) { date = concert.ref.date; }
```

3.4 Availability Type System

Availability levels in ConSysT not only define runtime semantics, but are also tracked by the type system as *availability types*. Availability types are pairs `C@a` where `a` is the availability level and `C` is a data type, e.g., a class. We use the notation `Ref<@l C>` to define concrete availability type in ConSysT. For example, `Ref<@Low Concert>` is a replicated `Concert` with `Low` availability.

The ConSysT type system tracks the availability types of replicated objects and ensures that data at different availability levels is not mixed mindlessly, as unconstrained mixing can violate consistency or isolation guarantees and lead to inconsistent replicas. In particular, ConSysT employs an *information-flow* type system to ensure that no information from *High* available data sources leaks into *Low* available data sinks (i.e., variables, fields, method parameters). Leakage from *High* sources to *Low* sinks occurs (a) when passing *High* values into *Low* data sinks either by assignment or as method parameter or return value, (b) when the control flow of a program is changed such that *Low* operations are affected by conditions on *High* data (also called *implicit flows*), (c) when instantiating objects at an availability level that would permit the leakage, and (d) when *Low* available data is accessed through *High* references. As the information-flow type system rejects all flow that reduces consistency guarantees specified by the programmer, it solves the *consistency leaks* problem (Section 2.1). Historically, information-flow type systems were used to prevent leakage of sensitive data [Denning 1976], but the concept was recently also employed for consistency models [Milano and Myers 2018]. In our work, we adapt information-flow control to enforce correct mixing of availability levels for replication in an *object-oriented* language.

3.4.1 Violation of Guarantees Through Explicit Information-Flow. In the following example, a *High* object is assigned to a *Low* field. Method `sellOut` in class `Concert` (Line 3) sells all available tickets for a concert by setting the field `value` of `soldTickets` to the maximum audience size of the `hall`. In the following code snippet, this is achieved with an assignment from `maxAudience` to `value` (Line 3).

```
1 Ref<@High ConcertHall> hall;
2 Ref<@Low Counter> soldTickets;
3 void sellOut() {  soldTickets.ref.value = hall.ref.maxAudience; }
```

Without the intervention of the type system, the program above might lead to an undesired state. For instance, assume that Process A reduces the size of the concert hall and then sells all tickets:

```
1 hall.ref.maxAudience -= 100;
2 concert.ref.sellOut();
```

When another Process B observes the invocation of `sellOut`, it may not have observed the assignment to `maxAudience` yet. In this case, `value` has the same value on Process A and Process B, but `maxAudience` is 100 lower for Process A. The intention of invoking `sellOut` is that a process is expected to stop selling tickets. But due to an unexpected order of operations, Process B can still sell 100 tickets.

To summarize, this unexpected behavior occurs because in `sellOut` there is an information-flow from the *High* available field `maxAudience` to the *Low* available field `value`, and there is no guarantee that the assignment to the `maxAudience` field takes place before the invocation of `sellOut`. For this reason, the ConSysT type systems rejects programs that contain assignments from *High* to *Low*.

3.4.2 Restricting Implicit Information-Flow. In addition to preventing direct data flow from *High* to *Low* available data, the type system also prohibits *implicit* information-flow. Consider the following code snippet, where `Concert` is defined such that `soldTickets` is *High*, and the other fields are *Low*:

```
1 Ref<@High ConcertHall> hall;
2 Ref<@Low Counter> soldTickets;
3 Optional<Ticket> buyTicket() {
4  if (hall.ref.maxAudience > getSoldTickets()) {
5   soldTickets.ref.inc();
6   return Optional.of(new Ticket());
7  } else { ... }
8 }
```

With this definition, the method `buyTicket` has a problematic implicit information-flow from the *High* reference `hall` to the *Low* reference `soldTickets`. The *Low* operation `soldTickets.ref.inc()` is called in the body of an if-statement with a condition that accesses the *High* value `maxAudience`. Hence, the invocation of `inc()` depends on a *High* value, which is an illegal information-flow and thus prohibited by the type system. In Figure 1b, `hall` is already defined as *Low*, which does not result in an illegal information-flow.

3.4.3 Restricting Object Creation. In ConSysT, the availability level of a field without an availability type is the same as the level of the containing object. For example, `Concert` has a field `date`: when `Concert` is instantiated as a *High* object, `date` is *High* as well.

```
1 class Concert {
2   Date date;
3   Ref<@High ConcertHall> hall;
4   void copyDate(Ref<@High Concert> concert) { date = concert.ref.date; }
5 }
```

Hence, the ConSysT type system considers whether the availability level of fields can violate the information-flow depending on the availability level of the containing object. Attaching to a field the same availability level as the field's object is not standard in information-flow control. In ConSysT, it stems from the fact that the data stored in the field without an availability type, i.e., `date`

in the example, is stored together with the containing object and thus replicated using the same availability level.

The type system prevents the creation of replicated objects if their availability level leads to an illegal information-flow. For example, the method `copyDate` in class `Concert` above sets the `date` of a concert to the `date` of another (*High*) concert: If `Concert` is instantiated as a *Low* object, the `date` field is *Low* as well. In this case, `copyDate` assigns the *High* value `concert.ref.date` to the *Low* field `date`, which is an illegal information-flow from *High* to *Low*. For this reason, ConSysT disallows creating replicated objects with an availability level such that the object is not compatible with the execution of any of its methods.

3.4.4 Restricting Information-Flow for References. The availability level of references that are accessed as a field of another replicated object *o* can not have stronger consistency guarantees than *o*. To clarify look at the following code snippet. The *Low* field `soldTickets` is accessed through a *High* reference to a concert.

```
Ref<@High Concert> concert = ...
Ref<@High Counter> counter = concert.ref.soldTickets;
```

Even though the field is declared as *Low*, `concert.ref.soldTickets` returns a *High* reference, because `concert` is *High*. If that would not be the case, then the following anomaly could happen. Assume that Process A assigns a counter to `soldTickets` with `concert.ref.soldTickets = /*...*/`. This assignment is a *High* operation, and such may not be visible to Process B that reads `soldTicket` afterwards. Process B would then have an outdated reference to the counter of `concert`, which violates the consistency guarantees of *Low* availability. ConSysT captures this in the type system by setting the availability level of a field access to the highest level between the declared availability level of the field and the level of the receiver object.

Discussion. In ConSysT, availability levels are attached *to objects*. An alternative design is to associate availability *to classes*: The availability of an object follows directly from its class. Although ConSysT enables to instantiate the same class with different availability levels, ConSysT's approach does not hinder reasoning about the availability level of an object, as the level of an object is reflected in its type and developers can reason about it at compile time. Additionally, annotating objects improves flexibility because each object can be assigned an individual availability level. This avoids code duplication: In case availability levels are associated to classes, implementing a `Counter` (as in Figure 1) would require to implement both a `HighCounter` and a `LowCounter`.

An alternative design is to attach availability levels *to operations* [Balegas et al. 2015; Houshmand and Lesani 2019; Sivaramakrishnan et al. 2015]. For example, in a bank account that can not have a negative balance, making a deposit can be a *High* operation but withdrawing has to be a *Low* operation. The reason is that replicas have to coordinate to ensure that two concurrent withdrawings do not result in a negative balance to determine which one is actually executed. In ConSysT, the availability level is attached to the bank account and all operations performed on the bank account have the same availability level. However, it is possible to declare the bank account *High* and have a distinct *Low* object `WithdrawHelper` that coordinates the withdraw operations.

```
class BankAccount {
 int balance;
 Ref<@Strong WithdrawHelper> w;
 void deposit(int amount) { balance += amount; }
 void withdraw(int amount) { w.ref.withdraw(this, amount); }
}

class WithdrawHelper {
 void withdraw(BankAccount acc, int amount) { acc.balance -= amount; }
}
```

$$
\begin{array}{rll}
P ::= & e_1, ..., e_n & \text{(Programs)} \\
Loc \ni \ell & & \text{(Locations)} \\
\\
Expr \ni e ::= & \texttt{x} \mid \texttt{let x} = e_1 \texttt{ in } e_2 & \\
& \mid \texttt{ref<C@a>}(\ell) \mid \texttt{new C@a}(\overline{e}) \texttt{ at } \ell & \\
& \mid e_0.\texttt{f} \mid e_0.\texttt{f} = e \mid e_0.\texttt{m}(e) & \\
Value \ni v ::= & \texttt{ref<C@a>}(\ell) &
\end{array}
\qquad
\begin{array}{rl}
ClassDef \ni D ::= & \texttt{class C}_1 \texttt{ extends C}_2 \{\overline{F};\ \overline{M}\} \\
F ::= & \alpha \texttt{ f} \mid \texttt{inferred C f} \\
M ::= & \beta \texttt{ m}(\alpha \texttt{ x})\{ \texttt{return } e \} \\
\\
ALevel \ni \texttt{a} ::= & \texttt{Local} \mid \texttt{Inconsistent} \mid ... \\
\alpha, \beta, \gamma ::= & \texttt{C@a}
\end{array}
$$

Fig. 3. Syntax of the core calculus.

3.5 Handling Failures

In distributed system, network partitions may occur anytime, e.g., due to node or network failures. *Low* operations require blocking coordination among nodes and cannot complete in case of network partitions because some nodes are not reachable. *High* operations, on the other hand, only need access to the local replica and defer coordination, thus *High* operations are not blocked by partitions.

ConSysT provides different approaches to notify developers about failures of *High* and *Low* operations. As *Low* operations can block until the *synchronous* coordination among replicas completes, ConSysT provides (optional) timeouts. The process can either continue waiting or can relinquish the guarantees of *Low* available objects and perform the operation on the local replica. In the language, timeouts result in exceptions which can only occur when accessing replicated objects via `refs` (Section 3.1). *High* operations, instead, are not directly affected by network partitions – coordination is *asynchronous*, i.e., non-blocking. Performing an operation on a *High* object never results in a timeout. Yet, in ConSysT, developers can manually synchronize *High* objects, leading to a timeout exceptions visible to the developer, just as with *Low* operations. Thus, also in the case of *High* available objects, developers may introduce different failure handling strategies on top of manual synchronization.

As failures can occur any time, it is possible that operations are only partially applied, i.e., an operation has only been applied on some replicas, or only a part of a nested operation has been applied on one replica (due to network partitions). Whether partial application of an operation is allowed depends on the availability level. For *High* objects, operations are allowed to be partially applied, as *High* availability allows inconsistencies between replicas. For *Low* objects, operations are not allowed to be partially applied. In this case, too, the impossibility of coordination leads to blocking, which is handled using timeouts, as discussed above.

4 FORMALIZATION

We provide a core calculus for ConSysT based on Featherweight Java [Igarashi et al. 2001], featuring mutable fields, objects are in (replicated) stores and only their references are in the syntax. Processes hold a replica and communicate by synchronizing replicas. We equip the calculus with a type system that tracks availability levels and prove its correctness. In the notation, overbars specify sequences, and subscripts project elements of a sequence, e.g., $\overline{\texttt{f}}$ is a sequence of field identifiers and $\texttt{f}_i$ is its i-th element.

4.1 Syntax

The syntax of the core calculus is in Figure 3. A programs P in the calculus consists of a sequence of expressions $e_1, ..., e_n$, where expressions define processes running in parallel. Expressions contain standard local variable identifiers `x`, and let bindings. `f` ranges over identifiers for fields, ℓ ranges over locations of replicated objects, `m` ranges over methods, and `C` ranges over class names. `ref<C@a>`(ℓ) defines a reference to a (replicated) object at store location $\ell \in Loc$. This captures the method `lookup` in ConSysT. Each replica has the same object at the same store location. The location is the

$$\frac{C(\texttt{C}) = \texttt{class C extends C}' \{...\}}{\texttt{C} <:_D \texttt{C}'}\ \textsc{TSub-Cls} \qquad \frac{}{\texttt{C} <:_D \texttt{C}'}\ \textsc{TSub-Refl}$$

$$\frac{\texttt{C} <:_D \texttt{C}' \qquad \texttt{C}' <:_D \texttt{C}''}{\texttt{C} <:_D \texttt{C}''}\ \textsc{TSub-Trans} \qquad \frac{\texttt{C}_1 <:_D \texttt{C}_2 \qquad \texttt{a}_1 \sqsubseteq \texttt{a}_2}{\texttt{C}_1\texttt{@a}_1 <: \texttt{C}_2\texttt{@a}_2}\ \textsc{ASub}$$

Fig. 4. Subtyping.

unique name of the object. The constructor invocation `new C@a(`$\overline{e}$`) at` ℓ creates a new object with availability level `a`. In the case of `a = Local` a new local object is created, otherwise it creates a replicated object. The latter case captures the method `replicate` in `ConSysT`. Further, the language supports field access, field modification and method invocation. The only value are references.

Class definitions D have a name $\texttt{C}_1$, a super class $\texttt{C}_2$, and contain a sequence of field and method declarations $\overline{F}$; $\overline{M}$. A field declaration F is a reference with an availability level α `f`. Alternatively, the availability level for `inferred C` is inferred from the containing object. Methods M take one parameter `x` and return the value of an expression e. Return values and the parameter type are both labeled with availability levels. The empty class `Object` is the top of the class hierarchy. We assume that there is no field overriding. Methods on the other hands can be overridden. In the following, we assume a globally available class table $C : \texttt{C} \rightarrow \mathit{ClassDef}$ that maps class identifiers `C` to class definitions $\mathit{ClassDef}$.

Labels `a` define availability levels. Availability types α are class identifiers `C` labeled with an availability level `a`, which corresponds to the notation `Ref<@l C>` used in `ConSysT`. Non-replicated objects have the special availability level `Local` – the bottom of the availability lattice.

In the following, we use the symbols introduced here to implicitly universally quantify over the specific syntax elements. We use i, j, and n to universally quantify over integers.

4.2 Type System

We define an information-flow type system for the core calculus that tracks the availability guarantees of program data. For brevity, we show only selected rules. The type system is inspired by type systems for object-oriented languages with information-flow [Hicks et al. 2006].

Basic Definitions. The subtyping relation is in Figure 4. For classes, $\texttt{C} <:_D \texttt{C}'$ is, as usual, according to the class hierarchy. We assume that $<:_D$ is anti-symmetric and the class hierarchy is acyclic. Subtyping for availability types is defined as $\alpha <: \alpha'$, which requires that the classes are subtypes and that the availability levels are ordered by $\sqsubseteq$. We define a type context $\Sigma : \mathit{Loc} \mapsto \alpha$ that defines an availability type for each location of the replicated store. The context is part of the program definition and statically defines for each location ℓ the class and availability level of an object at ℓ.

We introduce auxiliary definitions used in the type system. $\mathit{fields}(\texttt{C}, \texttt{a})$ returns a sequence of all field declarations F for an object of class `C` instantiated with availability level `a`. $\mathit{typeOfField}(\texttt{C}, \texttt{a}, \texttt{f})$ looks up the type of field `f` in `C`, and $\mathit{method}(\texttt{C}, \texttt{m})$ looks up the method declaration of `m` in `C`.

Typing of Classes. Typing for methods and classes is in Figure 5. The typing judgment for classes is $\Sigma \vdash D$ *is ok for* `a`, where Σ is a global type context, D is the class to be checked, and `a` is the availability level of the instantiating object. `a` defines the restriction of object instantiation based on the availability (Section 3.4.3). The rule T-Class-1 specifies that it is ok to instantiate a class for an object with availability level `a` if the super class and all methods are ok under that availability level. Rule T-Class-2 defines typing for local classes. Fields of local classes can contain values with any availability level. For methods, the typing judgment is $\Sigma \vdash M$ *is ok in* `C` *for* `a`, where Σ is a global type context, M is the method definition, `C` is the defining class, and `a` is the availability level of the

$$\frac{\Sigma \vdash \overline{M} \textit{ is ok in } \mathtt{C} \textit{ for } \mathtt{a} \qquad \Sigma \vdash C(\mathtt{C}') \textit{ is ok for } \mathtt{a} \qquad \mathtt{a} \neq \mathtt{Local}}{\Sigma \vdash \mathtt{class\ C\ extends\ C'}\ \{\overline{F};\ \overline{M}\} \textit{ is ok for } \mathtt{a}}\ \text{T-Class-1}$$

$$\frac{\Sigma \vdash \overline{M} \textit{ is ok in } \mathtt{C} \textit{ for } \mathtt{Inconsistent} \qquad \Sigma \vdash C(\mathtt{C}') \textit{ is ok for } \mathtt{Local}}{\Sigma \vdash \mathtt{class\ C\ extends\ C'}\ \{\overline{F};\ \overline{M}\} \textit{ is ok for } \mathtt{Local}}\ \text{T-Class-2}$$

$$\frac{\begin{array}{c}\Sigma; \bullet, \mathtt{x} \mapsto \alpha, \mathtt{this} \mapsto \mathtt{C@a} \vdash e : \gamma \qquad \gamma <: \beta \qquad C(\mathtt{C}) = \mathtt{class\ C\ extends\ C'}\ \{\ldots\} \\ \text{if } \mathit{method}(\mathtt{C}', \mathtt{m}) = \beta'\ \mathtt{m}(\alpha'\ \mathtt{x}')\{\ \mathtt{return}\ \ldots\ \} \text{ then } \alpha' = \alpha \wedge \beta' = \beta\end{array}}{\Sigma \vdash \beta\ \mathtt{m}(\alpha\ \mathtt{x})\{\ \mathtt{return}\ e\ \} \textit{ is ok in } \mathtt{C} \textit{ for } \mathtt{a}}\ \text{T-Method}$$

Fig. 5. Method and class typing.

$$\frac{\Sigma(\ell) <: \mathtt{C@a}}{\Sigma; \Gamma \vdash \mathtt{ref<C@a>}(\ell)\ :\ \mathtt{C@a}}\ \text{T-Ref} \qquad \frac{\begin{array}{c}\Sigma(\ell) = \mathtt{C_0@a} \qquad \mathtt{C} <:_D \mathtt{C_0} \qquad \Sigma \vdash C(\mathtt{C}) \textit{ is ok for } \mathtt{a} \\ \Sigma; \Gamma \vdash \overline{e} : \overline{\alpha} \qquad \mathit{fields}(\mathtt{C}, \mathtt{a}) = \overline{\beta\ \mathtt{f}} \qquad \overline{\alpha} <: \overline{\beta}\end{array}}{\Sigma; \Gamma \vdash \mathtt{new\ C@a}(\overline{e})\ \mathtt{at}\ \ell\ :\ \mathtt{C@a}}\ \text{T-New}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash e_0\ :\ \mathtt{C_0@a_0} \qquad \alpha <: \mathtt{C@a} \\ \mathit{typeOfField}(\mathtt{C_0}, \mathtt{a_0}, \mathtt{f}) = \mathtt{C@a} \qquad \Sigma; \Gamma \vdash e : \alpha\end{array}}{\Sigma; \Gamma \vdash e_0.\mathtt{f} = e\ :\ \alpha}\ \text{T-FieldWrite} \qquad \frac{P = e_1, \ldots, e_n \qquad \Sigma; \bullet \vdash \overline{e} : \overline{\alpha}}{\Sigma \vdash P\ \mathtt{is\ ok}}\ \text{T-Prog}$$

Fig. 6. Typing for expressions and programs.

instantiating object. The rule T-Method checks that the methods body is well-typed and that in case of overriding, the types of the methods match.

Typing of Expressions and Programs. Figure 6 shows an extract of the typing rules for expressions and programs. The expression type judgment $\Sigma; \Gamma \vdash e : \alpha$ states that expression e has availability type α with the global type context Σ and the local type environment $\Gamma : \mathit{Var} \to \alpha$, which maps variables to their types.

T-Ref states that the type of a reference to a store location ℓ has to be a super type of the declared type of the store location $\Sigma(\ell)$. T-New is for object creation. It requires that the consistency level of the object and the consistency level declared by the store are equal, and that the class of the object is a subclass of the class defined by the store. Further, the rule ensures that the class can be instantiated for the consistency level a and that the parameters are well-typed according to the fields of the class C. The order of the expressions has to match the order of fields. *fields* is only defined when $\mathtt{C} \in \mathrm{dom}(C)$ or $\mathtt{C} = \mathtt{Object}$, such that only classes in the class table or Object can be instantiated. For T-FieldWrite, the right hand side has to be a subtype of the declared type of f. The field write expression returns the assigned expression e, thus the return type is α.

T-Prog is the typing rule for programs. A program is well-typed, when all expressions representing the processes are well-typed. The type judgment requires a global type context, which has to be given with the program definition.

4.3 Store Model

We now introduce the replicated store used by the core calculus. Replicated stores contain *objects* $o \in \mathit{Obj}$ at locations $\ell \in \mathit{Loc}$. Objects belong to a class C and hold a sequence of values $\overline{v}$ one for each field of C, in the same order: $o \in \mathit{Obj} ::= \mathtt{obj\ C}(\overline{v})$.

Objects are stored in *replicas* $\omega : \mathit{Loc} \to (\mathit{ALevel} \times \mathit{Obj})$, which hold for each location $\ell \in \mathit{Loc}$ an instance of an object together with its availability level $\mathtt{a} \in \mathit{ALevel}$. Objects local to one replica

has availability level `Local`. A (replicated) *store* $\Omega = \omega_1, ..., \omega_n$ is a sequence of replicas ω_k. For a program P, the store is defined such that each expression $e_k \in P$ has its own replica ω_k.

Defining Consistency. As we want to reason about objects in a store with different availability levels and thus consistency guarantees, we provide two definitions *consistent* and *merge* for each availability level. The *consistent* predicate determines when an operation on a data store is possible, e.g., to restrict concurrent modifications in *Low* available consistency models. The *merge* operator is for (1) merging the state of two replicas that have diverged due to concurrent execution (conflict resolution), and (2) restricting *when* data can be merged to enforce consistency constraints. For example, in eventual consistency, operations and merges have no restriction as they can happen any time. For this approach to the definition of availability levels, we were inspired by Kaki et al. [2017], who propose an operational semantics where transactional isolation levels are expressed through a predicate. First, we define the consistency of a store.

Definition 4.1. A store $\Omega = \overline{\omega}$ is *fully-consistent*, iff $\mathtt{a} \neq \mathtt{Local} \wedge \omega_i(\ell) = (\mathtt{a}, ...) \Rightarrow \omega_i(\ell) = \omega_j(\ell)$.

With full consistency, all replicas of the store contain the same objects except for local objects.

Ensuring Consistency. Predicate $consistent_{\mathtt{a}}(\Omega, k, \ell)$ is satisfied if the object at location ℓ in replica $\omega_k \in \Omega$ is consistent to the complete store Ω w.r.t. the availability level `a`. With the predicate, we control when operations with availability `a` are performed, i.e., if the predicate does not hold, a process has to wait to progress. We require that If the store is fully-consistent the predicate holds – for any availability level:

$$\Omega \text{ fully-consistent} \Rightarrow consistent_{\mathtt{a}}(\Omega, k, \ell) \tag{1}$$

Predicate $consistent_{\mathtt{Local}}$ always holds because local operations do not rely on the replicated store – hence the consistency is not relevant. The predicate $consistent_{\mathtt{a}}$ models the coordination among replicas required for availability level `a`. When the predicate is defined using the store Ω, the system needs coordination among replicas as the current state of other replicas has to be checked. In the formalization, the predicate $consistent_{\mathtt{a}}$ captures the semantics of the availability level `a`, abstracting over the precise definition of the consistency model, i.e., over details like operations order and message propagation. As the availability levels are ordered in a lattice, we require that this ordering is also reflected in the *consistent* predicate. In particular, for two availability levels `a` and `a′`, if `a` has a lower availability than `a′`, then $consistent_{\mathtt{a}'}$ has to hold whenever $consistent_{\mathtt{a}}$ holds:

$$\mathtt{a} \sqsubseteq \mathtt{a}' \wedge consistent_{\mathtt{a}}(\Omega, k, \ell) \;\Rightarrow\; consistent_{\mathtt{a}'}(\Omega, k, \ell) \tag{2}$$

For example, the *consistent* predicate allows one to define different availability levels. In the case of eventual consistency, concurrent updates are always allowed, thus the predicate is always satisfied.

$$consistent_{Ev}(\Omega, k, \ell) \;\equiv\; true$$

In the case of sequential consistency, the store is only consistent for a location ℓ if all replicas agree on the object at that location. That means, that in the case of a concurrent operation, a replica has to wait until the effect of that operation is visible on all replicas before continuing.

$$consistent_{Seq}((\omega_1, \ldots, \omega_n), k, \ell) \;\equiv\; \forall k'.\omega_k(\ell) = \omega_{k'}(\ell)$$

Converging to Consistent Stores. Next, we define the operator *merge* that converges a store to a consistent state. Formally, we define $merge_{\mathtt{a}}(\Omega, o_1, o_2)$ for each availability level `a`, which returns an object which is the result of a merge of two objects, given the current store Ω. Like the predicate *consistent*, *merge* is defined differently for each availability level `a`.

$$\frac{\mathit{fields}(\mathtt{C}, \mathtt{a}) = \overline{\alpha\ \mathtt{f}} \qquad \text{if } \mathtt{f}' = \mathtt{f}_i \text{ then } v' = v_i}{\ulcorner \mathtt{obj}\ \mathtt{C}(\overline{v}).\mathtt{f}' \urcorner = v'} \text{ A-ReadField}$$

$$\frac{\mathit{fields}(\mathtt{C}, \mathtt{a}) = \overline{\alpha\ \mathtt{f}} \qquad \text{if } \mathtt{f}' = \mathtt{f}_i \text{ then } v' = v_i'' \qquad \text{if } \mathtt{f}' \neq \mathtt{f}_i \text{ then } v_i = v_i''}{\ulcorner \mathtt{obj}\ \mathtt{C}(\overline{v}).\mathtt{f}' := v' \urcorner = \mathtt{obj}\ \mathtt{C}(\overline{v''})} \text{ A-WriteField}$$

Fig. 7. Auxiliary definitions for objects.

In the following, we state requirements that the definition of $merge_{\mathtt{a}}$ has to satisfy for each availability level a. First, the *merge* is required to be commutative:

$$merge_{\mathtt{a}}(\Omega, o_1, o_2) = merge_{\mathtt{a}}(\Omega, o_2, o_1) \tag{3}$$

We require that if *merge* is defined o_1 and o_2 and the returned object have the same class.

$$merge_{\mathtt{a}}(\Omega, \mathtt{obj}\ \mathtt{C}_1(\ldots), \mathtt{obj}\ \mathtt{C}_2(\ldots)) = \mathtt{obj}\ \mathtt{C}_3(\ldots) \Rightarrow \mathtt{C}_1 = \mathtt{C}_2 = \mathtt{C}_3 \tag{4}$$

If *merge* is undefined for two objects, they can not be merged given the current state of store Ω. This is the case when certain consistency guarantees of the specified availability level are not fulfilled. However, we require that *merge* can always be applied to *some* object as long as the store $\Omega = (\omega_1, \ldots, \omega_n)$ is not fully-consistent.

$$\begin{aligned}&(\omega_1, \ldots, \omega_n) \text{ is not fully-consistent} \Rightarrow \\ &\quad \exists \ell, i, j, \mathtt{a}. \omega_i(\ell) \neq \omega_j(\ell) \wedge merge_{\mathtt{a}}(\Omega, \omega_i(\ell), \omega_j(\ell)) \text{ is defined}\end{aligned} \tag{5}$$

4.4 Dynamic Semantics

We define a small-step operational semantics for the core calculus as transition relations for processes (local transitions) and for programs (global transitions).

We define a small-step operational semantics for the core calculus as transition relations for processes and programs.

Auxiliary Definitions. The semantics relies on the auxiliary definitions in Figure 7. $\ulcorner o.\mathtt{f} \urcorner$ reads the value of a field f from an object o, and $\ulcorner o.\mathtt{f} := v \urcorner$ defines an object o with field f set to v. In the rules, the notation $e_1[\mathtt{x} \leftarrow e_2]$ indicates the substitution of x with e_2 in e_1.

Local Transition Rules. We first introduce the operational semantics for single processes. *Process configurations* $\langle \omega \mid e \rangle$ capture the states of processes, i.e., the state of the process' local replica ω and an expression e that defines the current execution. The transition is defined with the judgment $\Omega; k \vdash \langle \omega \mid e \rangle \rightarrow \langle \omega' \mid e' \rangle$, stating that when the whole replicated store has the state Ω, a (uniquely identified) process k with configuration $\langle \omega \mid e \rangle$ makes a step to configuration $\langle \omega' \mid e' \rangle$. Transitions of process configurations do not change the global store Ω, but only the local replica ω. Ω is only used to check whether the local replica is consistent with the *consistent* predicate.

Rule Context induces a call-by-value, left-to-right evaluation through an evaluation context E. Rule Let replaces the variable x in e_2 with value v as usual. New creates an object store location ℓ in ω with class C and availability a. Location ℓ must be free, i.e., there is no other object in ℓ.

The rules for operations – method invocations, field access/modification – read from and/or write to the local replica. Depending on the availability level of the operation, the transition can only be made if the store is consistent w.r.t. the *consistent* predicate. In case the transition can not occur, the process waits until the store satisfies the consistency requirements (i.e., the *consistent* predicate holds). FieldRead returns the value of the field f of object ℓ. The returned value is a reference (references are the only values in the calculus). The premise ensures that the replicated

(Context) $E ::= E[\cdot] \mid \texttt{let x} = E \texttt{ in } e \mid \texttt{new C@a}(\overline{v}, E, \overline{e}) \texttt{ at } \ell \mid E.\texttt{f} \mid E.\texttt{f} = e \mid v.\texttt{f} = E \mid E.\texttt{m}(e) \mid v.\texttt{m}(E)$

$$\frac{\Omega; k \vdash \langle \omega \mid e \rangle \rightarrow \langle \omega' \mid e' \rangle}{\Omega; k \vdash \langle \omega \mid E[e] \rangle \rightarrow \langle \omega' \mid E[e'] \rangle}\ \textsc{Context} \qquad \frac{}{\Omega; k \vdash \langle \omega \mid \texttt{let x} = v_1 \texttt{ in } e_2 \rangle \rightarrow \langle \omega \mid e_2[\texttt{x} \leftarrow v_1] \rangle}\ \textsc{Let}$$

$$\frac{\ell \notin \mathrm{dom}(\omega) \qquad \omega' = \omega, \ell \mapsto (\texttt{a}, \texttt{obj C}(\overline{v}))}{\Omega; k \vdash \langle \omega \mid \texttt{new C@a}(\overline{v}) \texttt{ at } \ell \rangle \rightarrow \langle \omega' \mid \texttt{ref<C@a>}(\ell) \rangle}\ \textsc{New}$$

$$\frac{\omega(\ell_0) = (\texttt{a}', \texttt{obj C}'(\overline{v})) \qquad \ulcorner \texttt{obj C}'(\overline{v}).\texttt{f} \urcorner = \texttt{ref<C@a>}(\ell) \qquad \mathit{consistent}_{\texttt{a}'}(\Omega, k, \ell_0)}{\Omega; k \vdash \langle \omega \mid \texttt{ref<C}_0\texttt{@a}_0\texttt{>}(\ell_0).\texttt{f} \rangle \rightarrow \langle \omega \mid \texttt{ref<C@(a}_0 \sqcup \texttt{a)>}(\ell) \rangle}\ \textsc{FieldRead}$$

$$\frac{\begin{array}{c}\omega(\ell_0) = (\texttt{a}', \texttt{obj C}'(\overline{v})) \\ \ulcorner \texttt{obj C}'(\overline{v}).\texttt{f} := v \urcorner = o' \qquad \omega' = \omega, \ell \mapsto (\texttt{a}', o') \qquad \mathit{consistent}_{\texttt{a}'}(\Omega, k, \ell_0)\end{array}}{\Omega; k \vdash \langle \omega \mid \texttt{ref<C}_0\texttt{@a}_0\texttt{>}(\ell_0).\texttt{f} = v \rangle \rightarrow \langle \omega' \mid v \rangle}\ \textsc{FieldWrite}$$

$$\frac{\omega(\ell_0) = (\texttt{a}', \texttt{obj C}'(\overline{v})) \qquad \mathit{method}(\texttt{C}', \texttt{m}) = \beta\ \texttt{m}(\alpha\ \texttt{x})\{\ \texttt{return } e\ \} \qquad \mathit{consistent}_{\texttt{a}'}(\Omega, k, \ell_0)}{\Omega; k \vdash \langle \omega \mid \texttt{ref<C}_0\texttt{@a}_0\texttt{>}(\ell_0).\texttt{m}(v) \rangle \rightarrow \langle \omega \mid e[\texttt{x} \leftarrow v][\texttt{this} \leftarrow \texttt{ref<C}'\texttt{@a}'\texttt{>}(\ell_0)] \rangle}\ \textsc{Invoke}$$

Fig. 8. Operational semantics: Local transitions.

$$\frac{\Omega = \omega_1, ..., \omega_n \qquad \Omega; k \vdash \langle \omega_k \mid e_k \rangle \rightarrow \langle \omega_k' \mid e_k' \rangle}{..., \langle \omega_k \mid e_k \rangle, ... \rightsquigarrow ..., \langle \omega_k' \mid e_k' \rangle, ...}\ \textsc{G-Local}$$

$$\frac{\begin{array}{c}\omega_j(\ell) = (\texttt{a}, o_j) \qquad \omega_k(\ell) = (\texttt{a}, o_k) \qquad \texttt{a} \neq \texttt{Local} \qquad o_j \neq o_k \\ o' = \mathit{merge}_{\texttt{a}}(\Omega, o_j, o_k) \qquad \omega_j' = \omega_j, \ell \mapsto (\texttt{a}, o') \qquad \omega_k' = \omega_k, \ell \mapsto (\texttt{a}, o')\end{array}}{..., \langle \omega_j \mid e_j \rangle, ..., \langle \omega_k \mid e_k \rangle, ... \rightsquigarrow ..., \langle \omega_j' \mid e_j \rangle, ..., \langle \omega_k' \mid e_k \rangle, ...}\ \textsc{G-Converge}$$

$$\frac{\omega_j(\ell) = (\texttt{a}, o_j) \qquad \omega_k(\ell) \text{ undefined} \qquad \texttt{a} \neq \texttt{Local} \qquad \omega_k' = \omega_k, \ell \mapsto (\texttt{a}, o_j)}{..., \langle \omega_j \mid e_j \rangle, ..., \langle \omega_k \mid e_k \rangle, ... \rightsquigarrow ..., \langle \omega_j \mid e_j \rangle, ..., \langle \omega_k' \mid e_k \rangle, ...}\ \textsc{G-Replicate}$$

Fig. 9. Operational semantics: Global transitions.

store Ω is consistent with respect to the availability $\texttt{a}'$ of the stored object, i.e., a field can only be read if the requirements of the availability level are met. FieldWrite changes to v the f field of the object o in replica ω at ℓ Like FieldRead, the rule can only be applied if *consistent* holds for the store. A field write returns the assigned value. Invoke looks up the receiver object at ℓ_0 and resolves its class to retrieve the method m. Again, a consistency check is performed to execute the method.

Global Transition Rules. The global transition rules consider all processes in the system in parallel. For a program P, a *program configuration* is a sequence of process configurations $\overline{\langle \omega \mid e \rangle}$ such that $P = \overline{e}$. The replicated store Ω is the sequence of all replicas $\Omega = \overline{\omega}$.

The global transition relation is in Figure 9. The program can make a global step when any process can make a local step, when state of differing local replica converge, or when new replicated objects are created. The program takes a step if one of the processes does a local transition (G-Local). In G-Converge, the *merge* operator defines how replicas converge. When two replicas ω_j and ω_k store different objects at location ℓ, the objects are merged and the result is stored in both replicas at ℓ. Objects with availability Local can not be merged as local objects are not replicated. When

merge is defined for objects at the same address, then the state of two replicas can be merged, and if they differ, making their state equal. Together with equation 5, this rule ensures to always find an object in the store that can be merged as long as the store is not fully-consistent, which ensures that the store is eventually fully-consistent. G-Replicate replicates objects that are not available on a replica ω_k to that replica. As before, `Local` objects are not replicated.

Extending the Store. At the end of Section 4.3, we defined eventual and sequential consistency. There are availability levels, such as causal consistency, that make assertions over the dependencies of operations and can not be specified with our current store model Store models that can capture such dependencies can be defined, e.g., as a history of operations [Viotti and Vukolić 2015]. In our work, we keep the store model intentionally simple to reason about the relation between the type system and replication. In the following we sketch how the store model can be extended to define causal consistency.

First, we extend the store to a multiversion store. The store contains multiple versions v of an object $\omega : Loc \times v \rightarrow (ALevel \times Obj)$. Whenever an operation is performed, a new version of the objects involved in the operation is added to the store. Further, the store keeps track of operation dependencies. We extend the store with a *happened-before relation* [Lamport 1978] for operations. This relation orders operations when they are sequential in a process or when an operation is visible to another one. Each replica holds its own version of the happened-before relation for all operations that are visible to that replica. When a process performs an operation it appends the operation to its own happened-before relation. When a process observes the effect of an operation a of another process, i.e., when an object is merged, then it observes all operations that are seen by a. That means that merging does not only merge objects but also the happened-before relations. Merging is only possible if all operations that are visible to the merged object are visible in the other replica as well.

4.5 Properties

In this section we discuss the properties of the core calculus. First, we give the definition of well-defined class tables and expressions.

Definition 4.2. Class table C is *well-defined*, iff the following hold: (1) $\texttt{Object} \notin \mathrm{dom}(C)$, (2) $\forall \texttt{C} \in \mathrm{dom}(C).\ C(\texttt{C}) = \texttt{class C extends} \ldots$, and (3) $\forall \texttt{C}$ anywhere in C, $\texttt{C} \in \mathrm{dom}(C) \vee \texttt{C} = \texttt{Object}$.

An expression e is *well-defined*, iff $\forall \texttt{C}$ in e. $\texttt{C} = \texttt{Object} \vee \texttt{C} \in \mathrm{dom}(C)$.

For class tables, the definition states that (1) `Object` is not a valid identifier for a class, (2) that the class definition at $C(\texttt{C})$ has the name `C`, and that (3) all class identifiers `C` that appear anywhere in C are `Object` or an element of C. We assume that the global class table C is well-defined in the following. Similar to class tables, an expression e is well-defined if all class identifiers `C` that appear in e are `Object` or an element of C. Next, we establish a relation between replicas ω and global type environments Σ. For that, we define a well-typed object:

Definition 4.3. Object $o = \texttt{obj C}(\overline{v})$ is *well-typed with* `a` *in* Σ, iff

$$\mathit{fields}(\texttt{C}, \texttt{a}) = \overline{\beta\ \texttt{f}} \Rightarrow \exists \overline{\alpha}.\ \Sigma; \bullet \vdash \overline{v} : \overline{\alpha} \wedge \overline{\alpha} <: \overline{\beta}.$$

Objects are well-typed, when the values of the fields are subtypes of the declared fields for an object with availability level `a`. We now define the relation between replicas and type environments.

Definition 4.4. Replica ω *satisfies* Σ, iff (1) $\mathrm{dom}(\omega) \subseteq \mathrm{dom}(\Sigma)$, and (2) $\Sigma(\ell) = \texttt{C@a} \wedge \omega(\ell) = (\texttt{a}', \texttt{obj C}'(\ldots)) \Rightarrow \texttt{obj C}'(\ldots)$ well-typed with `a` in $\Sigma \wedge \texttt{a} = \texttt{a}' \wedge \texttt{C} = \texttt{C}'$.

This property states that (1) all locations of the replica are in the type environment, and that (2) all objects in the store are well-typed with their availability level and that the type of the object adheres to the type declared in the environment.

4.5.1 Preservation. We show that the transition relation preserves typing. We start with formalizing preservation of expressions, i.e., when a well-typed expression e makes a step with a ω that satisfies Σ, then the resulting expression is well-typed and the resulting store satisfies a superset of Σ. Then, we define preservation for full programs.

Theorem 4.5 (Preservation of expressions). *If* $\Sigma; \Gamma \vdash e : \alpha$ *and* $\Omega; k \vdash \langle \omega \mid e \rangle \rightarrow \langle \omega' \mid e' \rangle$ *and* ω *satisfies* Σ*, then for some* $\Sigma' \supseteq \Sigma$ *and* $\beta <: \alpha$ *holds* $\Sigma'; \Gamma \vdash e' : \beta$ *and* ω' *satisfies* Σ'*.*

Theorem 4.6 (Preservation of programs). *Let* $P = e_1, ..., e_n$ *and* $\Omega = \omega_1, ..., \omega_n$*.*
If $\Sigma \vdash e_1, ..., e_n$ `is ok` *and* $\langle e_1 \mid \omega_1 \rangle, ..., \langle e_n \mid \omega_n \rangle \rightsquigarrow \langle e'_1 \mid \omega'_1 \rangle, ..., \langle e'_n \mid \omega'_n \rangle$ *and* $\overline{\omega}$ *satisfies* Σ*, then for some* $\Sigma' \supseteq \Sigma$ *holds* $\Sigma' \vdash e'_1, ..., e'_n$ `is ok` *and* $\overline{\omega'}$ *satisfies* Σ'*.*

4.5.2 Progress. Progress specifies that a well-typed program can make a reduction step [Wright and Felleisen 1994]. We use a weaker notion of progress *for expressions* as processes may not progress if they are waiting for a consistent store. *For full programs*, we use a standard definition of progress. In such case, processes waiting for consistency eventually perform a step when the consistency condition is satisfied. We first define unique locations for (sequences of) expressions.

Definition 4.7. $\overline{e}$ has *unique locations*, iff for the sequence $\overline{e'}$ of all subexpressions of $\overline{e}$ holds: $e'_i = \texttt{new C}_i\texttt{@a}_i(...)\ \texttt{at}\ \ell_i \wedge e'_j = \texttt{new C}_j\texttt{@a}_j(...)\ \texttt{at}\ \ell_j \wedge i \neq j \Rightarrow \ell_i \neq \ell_j$.

The property ensures that only one object is created for each store location ℓ. As locations are infinite, the property does not reduce expressiveness. Next, we define progress for expressions.

Theorem 4.8 (Progress of expressions). *If* $\Sigma; \bullet \vdash e : \alpha$ *and* e *is well-defined and has unique locations, and* ω *satisfies* Σ*, then*

(1) if $e = \texttt{ref<C}_0\texttt{@a}_0\texttt{>}(\ell_0).\texttt{f}$*, then* $\ell_0 \notin \mathrm{dom}(\omega)$ *or* $\neg consistent_{\texttt{a}'}(\Omega, k, \ell_0)$ *with* $\Sigma(\ell_0) = \texttt{C}'\texttt{@a}'$ *for some* $\texttt{C}'$ *or* $\Omega; k \vdash \langle \omega \mid e \rangle \rightarrow \langle \omega' \mid e' \rangle$ *for some* e' *and* ω'*.*
(2) if $e = (\texttt{ref<C}_0\texttt{@a}_0\texttt{>}(\ell_0).\texttt{f} = v)$*, then* $\ell \notin \mathrm{dom}(\omega)$*, or* $\neg consistent_{\texttt{a}'}(\Omega, k, \ell_0)$ *with* $\Sigma(\ell_0) = \texttt{C}'\texttt{@a}'$ *for some* $\texttt{C}'$ *or* $\Omega; k \vdash \langle \omega \mid e \rangle \rightarrow \langle \omega' \mid e' \rangle$ *for some* e' *and* ω'*.*
(3) if $e = \texttt{ref<C}_0\texttt{@a}_0\texttt{>}(\ell_0).\texttt{m}(v)$*, then* $\ell \notin \mathrm{dom}(\omega)$ *or* $\neg consistent_{\texttt{a}'}(\Omega, k, \ell_0)$ *with* $\Sigma(\ell_0) = \texttt{C}'\texttt{@a}'$ *for some* $\texttt{C}'$ *or* $\Omega; k \vdash \langle \omega \mid e \rangle \rightarrow \langle \omega' \mid e' \rangle$ *for some* e' *and* ω'*.*
(4) else $e \in Value$ *or* $\Omega; k \vdash \langle \omega \mid e \rangle \rightarrow \langle \omega' \mid e' \rangle$ *for some* e' *and* ω'*.*

Progress for *expressions* is only defined if the processes are not waiting for other processes to either replicate an object to a location ℓ or for the store to be consistent. In the case of *programs*, whether objects are replicated and the store is consistent eventually, depends on the specific definition of $merge$. As $merge$ is a partial function, rule G-Converge may not be applicable, leaving the store inconsistent, and possibly even leading to deadlocks. In the definition of progress for programs we assume that $merge$ is defined when applied on objects with the same class.

Theorem 4.9 (Progress of programs). *Let* $P = e_1, ..., e_n$ *and* $\Omega = \omega_1, ..., \omega_n$*. If* $\Sigma \vdash P$ `is ok`*, and* $\overline{e}$ *is well-defined and has unique locations, and* ω *satisfies* Σ*, and* $\forall \texttt{C}.\ merge_{\texttt{a}}(\Omega, \texttt{obj C}(...), \texttt{obj C}(...))$ *defined, then* $\forall i.\ e_i \in Value$ *or* $\langle e_1 \mid \omega_1 \rangle, ..., \langle e_n \mid \omega_n \rangle \rightsquigarrow \langle e'_1 \mid \omega'_1 \rangle, ..., \langle e'_n \mid \omega'_n \rangle$*.*

If $merge$ is defined for objects of the same class, the state of two replicas can be merged, if they differ, making their state equal. When the state of all replicas is equal, the store is fully-consistent. On a fully-consistent store $consistent$ is always satisfied, thus the processes waiting for consistency in the operational semantics can further progress. In practice, the requiring that

merge is always defined for two objects of the same class means that two objects can be merged any time disregarding any limitations.

4.5.3 Non-interference. Non-interference in the availability types system states that operations on High objects can not affect operations on Low objects. We first define what it means for stores and objects to be indistinguishable.

Definition 4.10. Objects $o_1 = \texttt{obj}\ \texttt{C}_1(\overline{v})$ and $o_2 = \texttt{C}_2\overline{v}'$ are *indistinguishable for* $\texttt{a}$ *when instantiated as* $\texttt{a}'$, iff $\texttt{C}_1 = \texttt{C}_2$ and with $\mathit{fields}(\texttt{C}_1, \texttt{a}_0) = \overline{\texttt{C@a f}}$ for all i holds: $v_i = v'_i$ if $\texttt{a}_i \sqsubseteq \texttt{a}$.

Replicas ω_1 and ω_2 are *indistinguishable for* $\texttt{a}$, iff for all $\texttt{a}' \sqsubseteq \texttt{a}$ such that $\texttt{a}' \neq \texttt{Local}$ holds: $\omega_1(\ell) = (\texttt{a}', o_1)$ and $\omega_2(\ell) = (\texttt{a}', o_2)$ and o_1 and o_2 are indistinguishable for $\texttt{a}$ when instantiated as $\texttt{a}'$. We write $\omega_1 \approx_{\texttt{a}} \omega_2$.

Now, we can define non-interference for expressions.

Theorem 4.11 (Non-interference). *If* $\Sigma; \Gamma \vdash e : \alpha$, *and* $\omega_1 \approx_{\texttt{a}} \omega_2$, *and* ω_1, ω_2 *satisfy* Σ, *and* $\Omega; k \vdash \langle \omega_1 \mid e \rangle \rightarrow^* \langle \omega'_1 \mid v_1 \rangle$, *and* $\Omega; k \vdash \langle \omega_2 \mid e \rangle \rightarrow^* \langle \omega'_2 \mid v_2 \rangle$, *then* $\omega'_1 \approx_{\texttt{a}} \omega'_2$.

Non-interference states that if a well-typed expression is evaluated twice with different stores that are only identical in their *Low* objects, in the resulting stores, *Low* objects are also identical. Hence, in a well-typed expression *High* values have no effect on *Low* objects.

5 IMPLEMENTATION

The implementation of ConSysT includes the type checker and the middleware that supports replicated objects with different availability levels. Overall, the implementation consists of ~ 5,500 lines of Scala and Java code.

Type Checker. A valid ConSysT program is a valid Java program. ConSysT adopts the Java type system for basic types, and implements availability types as Java type annotations using the Checker Framework [Papi et al. 2008]. ConSysT allows the definition of a lattice of annotations, and type checking is based on the subtype relation induced by the lattice and by an analysis of information flow to prevent implicit flows.

Middleware. Within each process (JVM), replicated objects are standard Java objects. We use Akka [Akka 2009] to implement the middleware that synchronizes the replicated objects.

We currently support three availability levels: eventual (*Ev*) and causal (*Cau*) belong to the *High* availability levels and do not require blocking coordination, whereas sequential (*Seq*) belongs to *Low* availability levels and requires coordination for consistency and isolation.

Seq provides sequential consistency and serializable isolation through a lock associated to each replicated object and managed by a *master* replica for that object. Processes contact master replicas to acquire the lock on one or more objects – the *lock set* – prior to performing operations. For field accesses, the lock set includes the receiver object. For method invocations, it includes all replicated objects (which we approximate via static analysis) accessed within the method execution. A two phase locking (2PL) protocol locks all the objects in the lock set before performing an operation. Crucially, this approach guarantees *pessimistic* concurrency control *without deadlocks*, which avoids aborting and retrying methods execution. As a result, methods can safely perform side effects – preserving the semantics of Java programs.

The protocol for *Seq* replicated objects works as follows. Consider a process p on a replica k performing an operation on the object o. Then the following steps are executed: (1) p tries to acquire the lock for all objects in the lock set of the operation by contacting their masters. The operation is repeated for all masters until the locks for all objects is granted. (2) Then, the masters return to p

the latest state of each object in the lock set. (3) Replica k updates its local state for all the objects in the lock set. (4) The operation is performed locally on replica k. (5) The new state of all changed objects is sent to and acknowledged by the masters. The locks in the lock set are released. (6) In the end, the system returns the result of the operation to the calling process.

Ev and *Cau* levels provide eventual and causal consistency, respectively, with no isolation, and require non-blocking coordination. Operations on a *Ev* or *Cau* object o are performed on the local replica, and eventually propagated to other replicas. In the case of *Ev*, the propagation has two steps: (i) operations are re-executed on a master replica for o, which is responsible for merging the operations on o from all replicas; (ii) the state of o in local replicas is synchronized with the state in the master. This synchronization step is called periodically or can be triggered manually. The order in which operations are applied is decided by the master replica. For example, assume operation m is performed on the replica o_1 of the object o, and operation n on the replica o_2, then ConSysT decides an order for n and m, e.g., first m and then n. To ensure eventual consistency, o_2 reverts to the state before executing n, then it applies m and n in the order decided by the master replica. As clients can always perform operations on their local replica and the middleware might asynchronously re-execute them in a different order, the protocol is non-blocking, In the case of *Cau*, operations are distributed to all other replicas where they are applied when all causal dependencies are satisfied. *Cau* associates metadata information (vector clocks) to operations such that replicas can detect whether the dependencies for an operation are fulfilled.

As *Ev* operations are synchronized by re-executing them on the master and *Cau* re-executes operations on other replicas, executing *Ev* or *Cau* operations that contain other operations requires a special treatment. In the remainder of this section, we only refer to *Ev* for simplicity. When the *Ev* operation is re-executed, then nested operations on other objects would be also re-executed, resulting in multiple executions. To address this issue, the ConSysT runtime adopts a multiversion cache. For example, in the following code snippet the invocation of `incSoldTickets` is *Ev*, and contains an operation on another object:

```
1 class Concert { ...
2   Ref<@Seq Counter> soldTickets;
3   void incSoldTickets() { return soldTickets.ref.inc(); /*Seq Op*/ }
4 }
5
6 Ref<@Ev Concert> concert = concert.ref.incSoldTickets(); /*Ev Op*/
```

Since `concert` is *Ev*, the `incSoldTickets` call (Line 6) occurs directly on the current replica, executing the `inc` method on `soldTickets`. Eventually, the `incSoldTickets` method is executed on other replicas and `inc` is performed again even though the user called `inc` only once. Instead, the multiversion cache ensures that the call to `inc` is cached and not repeated. In summary, when an operation a is nested in another operation b that may be re-executed, the result of a is cached to avoid re-executing a. In our implementation, b can be re-executed if it is either *Ev* or *Cau*. The availability level of a is irrelevant in this case.

6 EVALUATION

Our evaluation is based on case studies to validate the performance and the design of ConSysT. Additionally, we conducted microbenchmarks to evaluate the performance of different availability levels.

6.1 Performance

We evaluated each case study in three configurations: All-Seq, All-Weak and Mixed. In All-Seq, every replicated object has *Seq* availability, forbidding concurrent access to shared data,

and ensuring that the application is correct w.r.t. replication. This is the only possible choice to preserve correctness if the middleware does not allow mixing data with different availability levels. The Mixed configuration uses *High* available data whenever possible while still ensuring that application invariants are preseved. The type system ensures the correctness of mixing *High* and *Low* available data. The All-Weak configuration only uses *High* available data, even if this violates some application invariants. Comparing this configuration to the others is a measurement for the cost of correctness. We measure the latency of a single operation gain when using All-Weak and Mixed instead of All-Seq. The benchmarks are executed on nine processes.

Counter. We implemented a replicated integer counter [Holt et al. 2016; Milano and Myers 2018]. One instance keeps the master replica of the counter, and the follower replicas perform increment operations. In the Mixed configuration, the counter is replicated using *Ev* and there are no other *Seq* objects.

TicketShop. The case study is the TicketShop example from Section 2. In the Mixed configuration, the availability levels are assigned as in Figure 1b with *Seq* and *Ev* representing *Low* and *High*, respectively. This configuration has 75% *Seq* objects. One instance hosts the master replica of the TicketShop and eight follower replicas continuously perform operations on the shop: retrieving a band name (58%), retrieving the data of the concert (22%), and buying a ticket (20%). The operations are randomly chosen according to a Zipf distribution.

MixT Message Groups. We reimplemented this application from the MixT paper [Milano and Myers 2018]: users join message groups and post messages to all group users. Replicas continuously execute operations according to a Zipf distribution: post to a group (22%), add a user to a group (20%), retrieve the inbox of a user (58%). In Mixed, accessing the inbox and the users list is a *Ev* operation. Other operations are *Seq*. Out of the 13,500 replicated objects in this case study, 33,3% are *Seq* in the Mixed configuration. The master replicas are evenly distributed among followers, each hosting 500 groups, for a total of 4.5 K groups. Each group initially contains a single user.

E-Commerce. The case study is an online shopping application. A server holds the master replica of the products and of the users. These are implemented as a Java `LinkedList` and as a `HashMap` replicated using ConSysT. Followers randomly execute operations from a set of seven requests (from a Zipf distribution): searching the product database (38%), querying product information (19%) adding items to the cart (13%), adding balance (10%), logging in (8%), checking out (6%), and logging out (5%). In the Mixed configuration, the product and user databases are *Ev*, the users are *Seq*. The case study has 3 320 objects, with ~ 60% *Seq* objects in Mixed.

IPA Twitter Clone. We reimplemented the Retwis twitter clone from the IPA paper [Holt et al. 2016] using ConSysT. Retwis supports operations like tweeting, retweeting, following and unfollowing. The case study uses 27 000 replicated objects. Users and tweets are *Ev*-replicated objects. A replicated counter with *Seq* availability counts the retweets for each tweet. 9 000 objects are *Seq* in the Mixed configuration.

Results. We measured the latency for executing an operation for each case study. In each case study, followers are continuously executing concurrent operations simulating a high load on the system. For operations on *High* available objects, we synchronize objects approximately once for every 20 operations. Figure 10 shows the latency of the All-Weak and Mixed configurations relative to the All-Seq baseline. 1.0 is the latency of the All-Seq configurations. We executed the benchmarks in three different setups: (i) *locally* on a single machine (3.1 GHz Intel Core i5-7200U processor and 16 GiB memory, Linux Mint 19.2), (ii) in a single AWS *datacenter* on machines with 2.5 GHz Intel Xeon processors and 2 GiB memory running Ubuntu Server 18.04, and (iii) *geo-replicated*

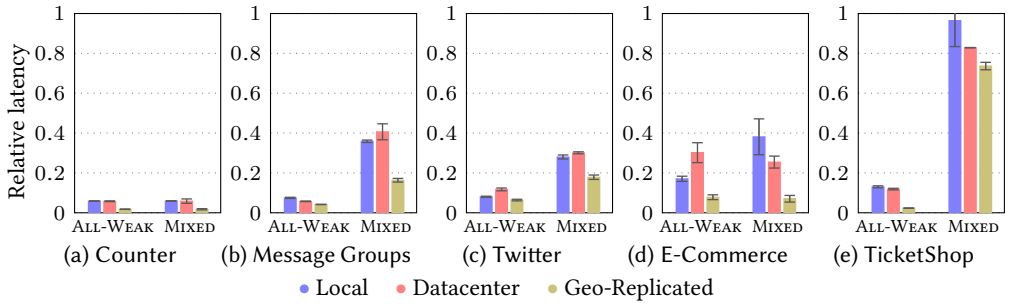

Fig. 10. Relative latency of one operation compared to All-Seq.

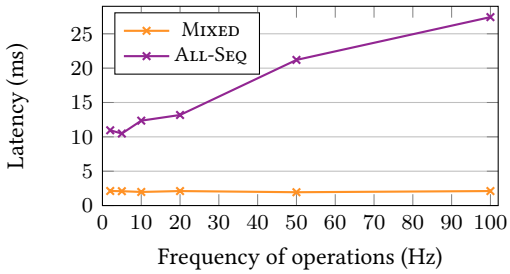

Fig. 11. Effect of contention.

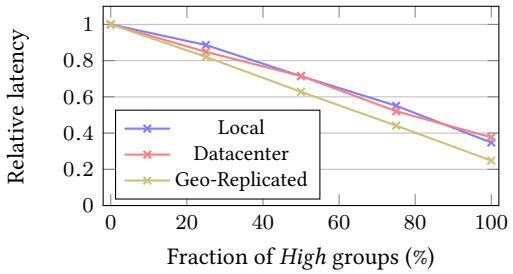

Fig. 12. Relative latency depending on *High* groups.

on the same AWS machines: 3 instances in EU (London), and 2 instances in US East(Ohio), Asia Pacific (Mumbai), and Canada (Central), each.

All our experiments clearly show that increasing the number of *High* available objects improves latency. Moreover, in many applications, the Mixed configuration performs relatively close to the best possible scenario for performance (All-Weak) without sacrificing application correctness. This proves the benefits of a system that mixes data at multiple availability levels, as ConSysT does. Moreover, the benefits increase in the case of geo-replication.

In *Counter*, the All-Weak configuration only takes $6\% - 16\%$ of the time of the baseline; indeed, there is only one replicated object, the counter, and all operations occur asynchronously in the All-Weak case. Since there is a single replicated object, the Mixed configuration is identical to All-Weak. In the *Seq* case, however, the followers have to block and synchronize for every operation. For *Message Groups*, the latency of Mixed is $16\% - 40\%$ compared to the baseline. The large speed-up of the Mixed configuration in the geo-distributed setup comes from the high latency between replicas, which increases the wait time for blocking synchronization. In *Twitter clone*, the Mixed configuration takes $18\% - 30\%$ of the run time compared to the baseline. Although the case study uses a single instance to host all master replicas, the speed-up is similar to Message Groups, which has the same percentage of *Seq* objects. Similarly, in *E-Commerce*, the Mixed configuration takes $7\% - 38\%$ of the run time of All-Seq. The reason is that for All-Seq the distributed data structures are very inefficient. A lookup in a distributed map or in a distributed list requires multiple (nested) operations, which are blocking in the case of All-Seq. In *TicketShop*, the Mixed configuration requires $74\% - 96\%$ of the time of All-Seq. The reason for the relatively small speed-up is that there is only one *High* object (and three *Low* objects). Thus, even in the Mixed configuration most operations are accessing *Seq* objects.

Effect of Contention. The results we discussed so far represent an extreme case in which applications continuously perform operations, resulting in high contention on locks for the All-Seq

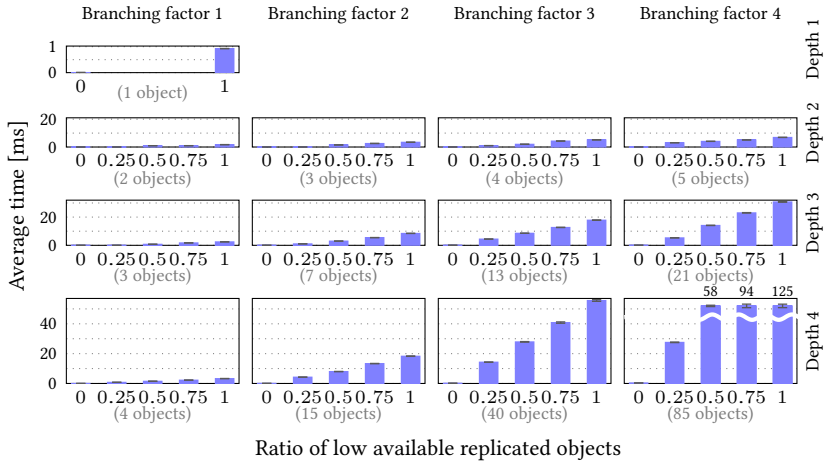

Fig. 13. Microbenchmarks.

configuration. This leads to an unusual high latency in some benchmarks as the replicated are highly contended and operations have to wait before being performed. To measure the protocol overhead apart from the queue time, i.e. the time that an operation is queued due to an overloaded system, we repeated the *Counter* case study with a fixed frequency of operations in the datacenter setup. The *Counter* case study is suited for this benchmark as the operations are not heavy and thus we can measure the overhead of the protocol. The results are presented in Figure 11. In the Mixed configuration, the latency lies between 1.94ms and 2.11ms per operation. In the All-Seq configuration, we see that the delay is highly influencial to the run time. In the case, where operations are send with 100Hz the latency is 27.43ms per operation, because of high contention. If the contention is reduced, then the latency goes down to 10.66ms per operation. This high number comparably to the mixed configuration is because of the blocking execution of strong operations. In summary, mixing availability levels (Mixed configuration) brings a large speed-up compared to the All-Seq configuration even with low contention.

Effect of Weak Objects. We further study how the percentage of *High* available objects impact on performance. We focus on the *MessageGroups* case study and we execute it with a different percentage of *High* available message group objects (Figure 12). In all setups – local, datacenter and geo-replicated – the latency decreases linearly with an increase of *Low* groups. In the geo-replicated case the performance gain of *High* is significantly higher than in the other setups, due to the impact of network latency on synchronization.

6.2 Microbenchmarks

To further compare the performance charateristics of each availability level, we implemented a microbenchmark that measures the performance of an operation under different levels. We create synthetic objects structures where objects include references to other objects and we change two dimensions: branching factor and depth. In the simplest configuration with an object structure of depth 1, we only have a single root object. A depth of n means that every leaf object is reachable over a path of length n from the root object. A branching factor of m means that every (non-leaf) object holds references to m other replicated objects. The rows represent an increasing depth of the object structure (from top to bottom) and the columns represent an increasing branching factor (from left to right). We vary the ratio of *Low* and *High* objects between 0 and 100 %, choosing the

```
1  RemoteList<Handle<user,causal>,linearizable> users;
2  mixt_method(add_post) (post) mixt_captures(users) (
3   var it = users,
4   while (it.isValid()) {
5    it->v.inbox.insert(post),
6    it = it->next
7   }
8 )
9
```

(a) MixT.

```
1 public Ref<@Seq List<Ref<@Cau User>> users;
2 public void addPost(String post) {
3  var it = users.ref.iterator();
4  while (it.hasNext())
5   it.next().ref.inbox.insert(post);
6 }
```

(b) ConSysT.

Fig. 14. Message Groups.

availability level for the objects in the structure randomly according to their ratio. The setup is an Intel Core i7-5600U, 2.6–3.2 GHz, 8 GiB.

Figure 13 presents the results for increasing depth (top to bottom) and increasing branching factor (left to right). The plots show the average time for updating all leaf objects (thereby traversing the complete object structure accessing every object once) and the 99.9% confidence intervals for different ratios of Low objects. After warmup, we run four iterations on three forked JVMs for 10 seconds each and take the average. The experiments show that running time scales linearly in the number of *Low* available objects.

6.3 Application Design

To evaluate the effects of ConSysT on software design, we analyze three of the case studies and compare different variants.

MixT Message Groups. Figure 14a shows the code for posting messages in MixT's Message Groups. MixT transactions can span multiple databases with different levels. In MixT, consistency levels are associated to the databases in which the objects are stored, e.g., the list of users is in a database that is strongly consistent (linearizable) and every user entry is in a database that is causally consistent (Line 1). Transactions are introduced by `mixt_method` (Line 2). The code iterates over all users (Line 3–8) and adds a post to every inbox (Line 5). In MixT, (i) transactions must be written in a DSL embedded into C++ code (Line 2–8), (ii) replicated objects can only be accessed and interact with each other inside transactions and (iii) transactions cannot be nested. In ConSysT (Figure 14b) instead, (i) no separate DSL for transactions is needed, and availability levels reconcile with standard OO programming as (ii) replicated objects can be used and interact with each other everywhere in the program and (iii) operations on replicated objects can be nested, i.e., they can call other operations on replicated objects.

IPA Twitter Clone. In the IPA version of the twitter clone there are three weakly consistent sets (`retweets`, `followers`, `followees`) which are instances of the ADT `IPASet`. In the IPA system, only predefined ADTs, such as `IPASet` support consistency levels, hindering code reuse because developers need to implement a new ADT (and the Cassandra query relative to each ADT operation) for each object they want to associate a consistency level with. ConSysT instead enables developers to associate availability levels to objects from existing Java classes. In the ConSysT reimplementation, the objects above are modeled using a standard set form the Java collections library. The specific type is a set of weakly consistent references to replicated objects (`Set<Ref<@Weak ...>>`).

TicketShop. We compare two versions of TicketShop. In version *v1*, the runtime does not provide any out-of-the-box consistency guarantee: coordination across replicas needs to be implemented in

```
Ref<@Ev Counter> soldTickets;
int getSoldTickets() {
 RequestHandler handler = soldTickets.getReplSys().acquireHandlerFromMaster();
 SerialResult<Counter> res = handler.request(soldTickets.getBoundName(),
  new SerializeOperations(soldTickets.getLocalOperations()));
 handler.close();
 return result.object.value;
}
```

```
// Centralized component
Result handleRequest(String name,Request req) {
 if (request instanceof SerializeOperations) {
  Ref<?> object = replicatedObjects.get(name);
  synchronized (object) {
   ((SerializeOperations) request).getOperations()
   .forEach(object::applyOperation);
  }
  return new SerializationResult(object);
 } else { ... }
}
```

(a) Manual synchronization.

```
Ref<@Seq Counter> soldTickets;
int getSoldTickets() {
 return soldTickets.ref.value;
}
```

(b) ConSysT.

Fig. 15. Access to `soldTickets` in TicketShop.

user code. Version *v2* uses ConSysT abstractions to define *Seq* replicated objects. Figure 15a and 15b show an excerpt of both versions, focusing on the code defining the accesses to the `soldTickets` field. Version *v1* manually enforces the sequential consistency semantics on top of weaker models (such as *Ev* in ConSysT). For instance, in Figure 15a, replicas explicitly synchronize with each other by sending `SerializeOperations` requests to a central master that orders the accesses to replicated objects. In *v2*, ConSysT's availability types make the developer's intent explicit providing the runtime the information to execute the application correctly, hiding the complexity of distributed consensus. The full TicketShop example in Figure 1b performs three accesses to *Seq*-available values (`soldTickets.ref.value`, `hall.ref.maxAudience`, `soldTickets.ref.inc()`), each requiring additional ~ 20 LOC. Also, in ConSysT the developer does not need to implement error-prone synchronization on the master side.

7 RELATED WORK

The trade-off between availability and consistency with transactional guarantees has been the subject of many studies. We first discuss recent trends in distributed database systems and replicated data stores. Then we focus on language abstractions to reason about consistency and availability, application-level specification for consistency, and mergable replicated datatypes.

7.1 Consistency and Availability

In the design space of availability in distributed systems, authors proposed hierarchical models – lattices – that account for various combinations of consistency and isolation. For example, Viotti and Vukolić [2015] provide a survey on consistency in non-transactional distributed storage. Adya et al. [2000] present implementation-independent specifications of isolation levels, including existing ANSI levels. Recent works study algorithms and techniques for strong consistency and transactional isolation in (geo-)replicated data stores with acceptable performance, e.g., Google Spanner [Corbett et al. 2013], exploiting the availability of accurate clocks, and Calvin [Thomson

et al. 2012], relying on the deterministic execution of transactions to reduce overhead. Other databases, e.g., H-Store [Stonebraker et al. 2007] and ReactDB [Shah and Vaz Salles 2018], make distribution – for its performance effects – explicit in the schema and query language.

The idea of mixing multiple consistency/availability levels within the same system was pioneered by RedBlue consistency [Li et al. 2012]: programmers label operations that do not commute or potentially violate invariants as strongly consistent and the remaining ones as weakly consistent. The runtime then adopts different protocols to synchronize replicas. Along the same line, the Olisipo coordination service [Li et al. 2018] provides finer-grained consistency specifications for geo-replicated systems. For example, it supports synchronizing specific groups of operations with each other, but not with operations in other groups. Yu and Vahdat [2001] propose a continuous consistency model to dynamically trade consistency for availability where applications specify a maximum deviation from strong consistency for each replica. The Pileus NoSQL storage system [Terry et al. 2013] allows developers to specify SLAs that combine both latency and consistency for each *get* operation.

An important research line concerns finding criteria for ensuring that an application exhibits a certain consistency level. For example, Brutschy et al. [2017] provide an analysis for serializability in distributed applications. SIEVE [Li et al. 2014] combines static and dynamic analysis to determine when it is necessary to use strong consistency to preserve invariants and when it is safe to use a weaker consistency model. Predictive Treaties [Magrino et al. 2019] relax the synchronization of strong consistent data by predicting how results of strong operations change over time based on the history of the program. Sorteria [Nair et al. 2020] is a tool for deciding whether a distributed object can be implemented with high availability and still satisfy eventual consistency. It uses a proof system that checks if the object satisfies program invariants in the face of concurrency.

7.2 Abstractions for Consistency and Availability

Several approaches adopt programming language techniques to support multiple availability levels.

MixT [Milano and Myers 2018] is a C++ DSL for transactions that span multiple data stores each with a different consistency level. The compiler automatically distributes each transaction across the data stores. Similar to ConSysT, MixT adopts an information-flow type system for safely mixing multiple consistency models. Yet, MixT exhibits fundamental differences for developers compared to ConSysT. MixT is a *separate* domain-specific language for database transactions and operations on replicated objects can only be executed inside MixT code. In MixT, only transactions written in the MixT DSL can manipulate replicated objects – which have a direct mapping to the database store. The available operations depend on custom operations supported by the specific underlying data store for a specific datatype. In contrast, ConSysT integrates availability with OO programming and lets developers uniformly operate on replicated objects. Additionally, the execution of a MixT transaction can abort to rollback the associated database transaction. In this case, side effects and changes to the program state would lead to inconsistencies. ConSysT supports the traditional semantics of object-oriented languages, without rollbacks.

DCCT is a data-oriented query language that mixes multiple consistency levels [Zaza and Nystrom 2016]. In DCCT, *actions* (e.g., queries), access a distributed storage and annotations define the consistency of values. In contrast to ConSysT, which integrates availability levels into an OO programming model, DCCT is based on relational queries. CAPtain.js [Myter et al. 2018] is a JavaScript library with two abstract data types, *consistents* and *availables* laying at the opposite extremes in the design space according to the CAP theorem: the former ensures strong consistency by sacrificing availability, the latter ensures availability, but only provides eventual consistency. In contrast to ConSysT, which features availability types, in CAPtain.js, conflicts among consistency levels generate runtime exceptions.

The idea that type safety should imply consistency safety has been introduced by Holt et al. [2016] in the IPA system. Using subtyping, IPA prevents that values with high consistency flow into values with low consistency, which however, does not rule out control dependencies as ConSysT uses an *information flow* type system (Section 3.4). In contrast to ConSysT, which enables associating availability levels to any (nested) Java object (including those from the standard library), IPA only supports consistency for predefined ADTs that implement the necessary operations on the Cassandra backend.

Geo [Bernstein et al. 2017] is an actor based framework that implements the *virtual actor model* [Bykov et al. 2011], i.e., it automates placement, discovery, recovery, and load-balancing of actors. Instead of associating availability to data, as ConSysT does, Geo exposes availability levels via different operations. Geo, however, does not provide any guarantee for operations on multiple actors, while ConSysT supports isolation for methods that involve several objects. AJ [Dolby et al. 2012] is a concurrent language where object fields are grouped into sets that must be updated atomically. Code fragments, referred to as *units of work* are associated to atomic sets. The compiler automatically adds synchronization operations to preserve the consistency of the set. The features above result in a consistency model that, similar to ours, is *data* centric: update consistency is associated to data sets. Yet, AJ does not target availability in distributed systems. CScript [De Porre et al. 2020] is a programming language that allows to define replicated objects that are available or consistent. The language does not allow to mix consistency models to avoid losing consistency guarantees. *Correctables* [Guerraoui et al. 2016] are an abstraction for incremental consistency. Correctables capture successive refinements of the result of an operation on a replicated object: applications receive both preliminary results (fast but possibly inconsistent) as well as final (consistent) results that become available later, thus enabling speculative execution. Correctables are complementary to our approach. They enable computations with multiple consistency levels but do not support the interaction among levels, as we do. Hercules [Lebeck et al. 2019] is a system in which shared data is stored on a central (replicated) server and clients cache their local copy. Shared data can be accessed either at the local copy as weak consistent data, or the server as strong consistent data.

7.3 Application-Level Consistency Specification

Consistency can be inferred from user-defined program invariants. Quelea [Sivaramakrishnan et al. 2015] is a declarative programming model for eventually consistent data stores. It defines a contract language to specify fine-grained consistency properties of methods and it automatically generates a consistency protocol to enforce the contract. In Indigo [Balegas et al. 2015], consistency is specified via user-defined conditions on operations. The system statically checks whether two operations would violate a condition when executed concurrently and, in case, it introduces coordination to decide a serial execution order. Similarly, Houshmand and Lesani [2019] design a language for replicated objects which lets developers define conditions on methods. The system uses such conditions to find pairs of methods that are conflicting when executed concurrently and synthesizes synchronization protocols for them. In contrast to these approaches, ConSysT does not require programmers to specify application invariants on methods. Instead, developers define the availability of objects via availability types. We believe this approach is more intuitive for OO programmers, which think of their programs in terms of objects rather than functions. Further, ConSysT releases developers from writing potentially complex invariants on object state. The trade-off is that we can not infer synchronization at the granularity of methods, i.e., execute two methods in an object that have no conflict without coordination, whereas ConSysT infers synchronization on the granularity of objects.

7.4 Mergeable Data Types

Certain data types have been tailored for weakly consistent models such as eventual consistency. CRDTs [Shapiro et al. 2011] are distributed data types that are (strong) eventual consistent. To ensure consistency, CRDTs require either that all operations are commutative (operation-based), or that there is commutative merge operation (state-based). As state-based CRDTs may require to propagate potentially large state, Almeida et al. [2018] propose delta-state CRDTs that only propagate incremental state changes. Burckhardt et al. [2014] propose a framework to specify the semantics of eventually consistent replicated data types and to prove the correctness of their implementation. The authors also develop a notion of *optimality* for proving lower bounds on the worst-case proportion of metadata needed to resolve conflicts. Cloud types [Burckhardt et al. 2012] provide eventually consistent storage and define an abstraction layer over recurring engineering challenges like the details of Web service implementation, communication protocols, and caching. Cloud types are similar to CRDTs, but they also support non-commutative operations when given a merging strategy.

Zhao and Haller [2021] propose a language with a consistency type system that supports non-interference for strongly and weakly consistent data. It builds upon the observable atomic consistency protocol [Zhao and Haller 2018] which combines mergeable data types and reliable total order broadcast. The resulting abstractions provide both high availability through mergeable data types and acceptable latency for strong consistency.

Antidote DB [Akkoorath and Bieniusa 2016] is a database that features geo-replication, CRDTs and highly available transactions. Antidote SQL [Lopes 2018], supports consistency annotations specifying how concurrent updates to a column are synchronized. A *no concurrency* annotation indicates that objects are synchronized whenever an update occurs on that column, resembling ConSysT's *Low* level, which also does not allow concurrent updates.

ConSysT does not use CRDTs for weak consistency. CRDTs restrict the possible operations (commutativity), or restrict the replicated state (mergeable state). ConSysT instead provides consistency through its middleware without restrictions in the operations or the shared state. This result is achieved by either immediate synchronization for *Low* availability, or by agreeing on an operation order when synchronizing for *High* availability. CRDTs achieve consistency without additional synchronization whereas ConSysT allows writing programs without any restriction.

8 CONCLUSION

We presented ConSysT, a language for distributed systems that integrates object-oriented programming and data availability (consistency and isolation) in a coherent design. The ConSysT type system, featuring *availability types*, ensures that programs do not incur into errors due to the combination of incompatible availability levels. We formalize ConSysT with a core calculus and prove the type system correct. Our evaluation shows that ConSysT performs efficiently and demonstrates the design benefits of our solution.

ACKNOWLEDGMENTS

This work has been supported by the by the LOEWE initiative (Hesse, Germany) within the emergenCITY centre, by the German Research Foundation (DFG) within projects SA 2918/2-1 and SA 2918/3-1, and it has ben funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297.

REFERENCES

A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*. 67–78. https://doi.org/10.1109/ICDE.2000.839388

Akka. 2009. Akka toolkit for Java and Scala. Retrieved 2019-04-06 from https://akka.io

Deepthi Devaki Akkoorath and Annette Bieniusa. 2016. Antidote: the highly-available geo-replicated database with strongest guarantees. *SyncFree Technology White Paper* (2016).

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (2018), 162 – 173. https://doi.org/10.1016/j.jpdc.2017.08.003

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proceedings of the VLDB Endowment* 7, 3 (Nov. 2013), 181–192. https://doi.org/10.14778/2732232.2732237

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 6, 16 pages. https://doi.org/10.1145/2741948.2741972

Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. 2017. Geo-distribution of Actor-based Services. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 107 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133931

Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for eventual consistency: criterion, analysis, and applications. *ACM SIGPLAN Notices* 52, 1 (2017), 458–472. https://doi.org/10.1145/3093333.3009895

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-642-31057-7_14

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848

Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2038916.2038932

Raphaël Collet et al. 2007. *The limits of network transparency in a distributed programming language.* Ph.D. Dissertation.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

Kevin De Porre, Florian Myter, Christophe Scholliers, and Elisa Gonzalez Boix. 2020. CScript: A Distributed Programming Language for Building Mixed-Consistency Applications. *J. Parallel and Distrib. Comput.* 144 (2020), 109 – 123. https://doi.org/10.1016/j.jpdc.2020.05.010

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. 2012. A Data-centric Approach to Synchronization. *ACM Trans. Program. Lang. Syst.* 34, 1 (May 2012), 4:1–4:48. https://doi.org/10.1145/2160910.2160913

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (Nov. 1976), 624–633. https://doi.org/10.1145/360363.360369

Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 169–184.

Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. 2006. Trusted declassification: High-level policy for a security-typed language. *Proceedings of the 2006 workshop on Programming languages and analysis for security - PLAS '06* (2006), 65. https://doi.org/10.1145/1134744.1134757

Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing - SoCC '16*. ACM Press, New York, New York, USA, 279–293. https://doi.org/10.1145/2987550.2987559

Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. *Proc. ACM Program. Lang.* 3, POPL, Article 74 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290387

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2017. Alone Together: Compositional Reasoning and Inference for Weak Isolation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 34. https://doi.org/10.1145/3158115

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

Niel Lebeck, Jonathan Goldstein, and Irene Zhang. 2019. *Hercules: A Multi-View Cache for Real-Time Interactive Apps.* Technical Report.

Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 281–292. http://dl.acm.org/citation.cfm?id=2643634.2643664

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. 2018. Fine-grained consistency for geo-replicated systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 359–372. https://www.usenix.org/conference/atc18/presentation/li-cheng

Pedro S. Lopes. 2018. Antidote SQL: SQL for Weakly Consistent Databases. http://hdl.handle.net/10362/68859

Tom Magrino, Jed Liu, Nate Foster, Johannes Gehrke, and Andrew C. Myers. 2019. Efficient, Consistent Distributed Computation with Predictive Treaties. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 36, 16 pages. https://doi.org/10.1145/3302424.3303987

Matthew Milano and Andrew C. Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 226–241. https://doi.org/10.1145/3192366.3192375

Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2018. A CAPable Distributed Programming Model. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2018)*. ACM, New York, NY, USA, 88–98. https://doi.org/10.1145/3276954.3276957

Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 544–571.

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 201–212. https://doi.org/10.1145/1390630.1390656

Vivek Shah and Marcos Antonio Vaz Salles. 2018. Reactors: A Case for Predictable, Virtualized Actor Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, 259–274.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, Vol. 50. ACM Press, New York, New York, USA, 413–424. https://doi.org/10.1145/2737924.2737981

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 1150–1160.

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 309–324. https://doi.org/10.1145/2517349.2522731

Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

Paolo Viotti and Marko Vukolić. 2015. Consistency in Non-Transactional Distributed Storage Systems. *Comput. Surveys* 49, 1 (jun 2015), 1–34. https://doi.org/10.1145/2926965 arXiv:1512.00168

Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (2009), 40–44.

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994).

Haifeng Yu and Amin Vahdat. 2001. The Costs and Limits of Availability for Replicated Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/502034.502038

Nosheen Zaza and Nathaniel Nystrom. 2016. Data-centric Consistency Policies: A Programming Model for Distributed Applications with Tunable Consistency. *First Workshop on Programming Models and Languages for Distributed Computing on - PMLDC '16* (2016), 2–5. https://doi.org/10.1145/2957319.2957377

Xin Zhao and Philipp Haller. 2018. Observable Atomic Consistency for CvRDTs. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/3281366.3281372

Xin Zhao and Philipp Haller. 2021. Consistency types for replicated data in a higher-order distributed programming language. In *The Art, Science, and Engineering of Programming*, Vol. 5.