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, Technische Universität Darmstadt, Germany

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 consistency 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.

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 [38].

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 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 [29], specialized ADTs that support different consistency levels [19, 25] and function preconditions and invariants [6, 20, 34].

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.

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, Technische Universität Darmstadt, Darmstadt, Germany, salvaneschi@cs.tu-darmstadt.de.

2020. XXXX-XXXX/2020/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ConSysT's integration of consistency and object-oriented programming builds on two major insights. (i) In line with the recent research on distributed systems [5], 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 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.

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.

```java
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.

```java
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.

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 datastore, 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

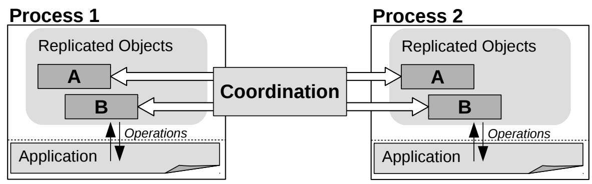

Fig. 2. System model.

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 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`:

```
Ref<Band> band = ...
Ref<ConcertHall> hall = ...
Ref<Concert> concert1 = replicate("concert1",
  new Concert(date("2019-09-25"), hall, band));
```

Method `replicate` returns a reference `Ref` to the new replicated object. Other processes can reference the object by its global name `"concert1"`, 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 whenever a `Concert` object is instantiated. This counter can only be referenced through the field `soldTickets` in a replicated `Concert` object.

```
class Concert {
 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("concert1");
```

The reference points to the replica that is located on the process. Applications can perform operations – method invocation, field access, field modification – on replicated objects as on normal Java objects. 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 [12]. 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:

```
Date d = concert1.ref.date;
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 [5]. 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 [22]. Serializability can be achieved using distributed locking protocols such as two-phase locking [16], 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 [37], and for transactional in single-object consistency by Bailis et al. [5].

In ConSysT, the availability level of a replicated object is defined at object creation. The following code fragment shows the creation of a new replicated `Concert` object with a *High* availability level.

```
<@High Concert>replicate("c1",
  new Concert(date("2019-09-25"), hall, band));
```

The availability level can be also inferred from the type of the reference to the replicated object. The following code snippet exemplifies how availability shows up *in the type* of a replicated object.

```
Ref<@High Concert> concert1 = replicate("c1",
  new Concert(date("2019-09-25"), hall, band));
```

Upon lookup, the availability level of the reference has to match the level of the replicated object:

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

In this case, 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 15). 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 have weak or strong 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.

```
class Concert {
 Date date;
 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@l` where `l` 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 type representing a replicated `Concert` with the `Low` availability level.

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 can reduce the consistency guarantees specified by the programmer, it solves the *consistency leaks* problem (Section 2.1). Historically, information-flow type systems are used to prevent leakage of sensitive data [14], but the concept was recently also employed for consistency models [29].

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 4) 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 5).

```
1 class Concert {
2  Ref<@High ConcertHall> hall;
3  Ref<@Low Counter> soldTickets;
4  void sellOut() {
5   soldTickets.ref.value = hall.ref.maxAudience;
6  } ... }
```

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:

```
Ref<@Low Concert> concert = ...
Ref<@High ConcertHall> hall = concert.ref.hall;
hall.ref.maxAudience -= 100;
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 available objects.

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*:

```
class Concert {
  Ref<@High ConcertHall> hall;
  Ref<@Low Counter> soldTickets;
  Optional<Ticket> buyTicket() {
    if (hall.ref.maxAudience > getSoldTickets()) {
      soldTickets.ref.inc();
      return Optional.of(new Ticket());
    } else ...
  } ... }
```

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.

```
class Concert {
  Date date;
  Ref<@High ConcertHall> hall;
... }
```

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. In particular, the type system prevents the creation of replicated objects if their availability level leads to an illegal information-flow. For example, the method `copyDate` below sets the `date` of a concert to the `date` of another (*High*) concert:

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

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 new counter to `soldTickets`.

```
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 availability level individually. This avoids code duplication: In case availablity levels are associated to classes, implementing a `Counter` (as in Figure 1) would require to implement both a `HighCounter` class and a `LowCounter` class.

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 network 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 [21], featuring mutable fields. Objects are in (replicated) stores and are represented as references in the language. 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., $\overline{\mathtt{f}}$ is a sequence of field identifiers and $\mathtt{f}_i$ is its i-th element. Proof sketches are inline, larger proofs are in Appendix **??**.

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 $e_1, \ldots, e_n$, where expressions define processes running in parallel. Expressions contain

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

Fig. 3. Syntax of the core calculus.

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

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

Fig. 4. Subtyping.

standard local variable identifiers `x`, and let bindings. `f` ranges over identifiers for fields, ρ ranges over locations of replicated objects, `m` ranges over methods, and `C` ranges over class names. `ref<C@l>`(ρ) defines a reference to a (replicated) object at store location $\rho \in \mathit{Loc}$. This captures the method `lookup` in `ConSysT`. Each replica has the same object at the same store location. The location is the unique name of the object. The constructor invocation `new C@l`($\overline{e}$) `at` ρ creates a new object with availability level `l`. In the case of `l` = `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.

A globally available class table $\mathcal{C}$ maps class identifiers `C` to definitions. 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.

Labels `l` define availability levels. Availability types α are class identifiers `C` labeled with an availability level `l`, 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.

4.2 Type system

We define an information-flow type system for the core calculus that tracks the availability guarantees of program data. The type system is inspired by type systems for object-oriented languages with information-flow [18].

Subtyping and global type context. 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

$$\frac{}{\mathit{fields}(\texttt{Object}, \texttt{l}) = \bullet}\ \textsc{A-Fields-1} \qquad \frac{\begin{array}{c}\mathit{fields}(\texttt{C}', \texttt{l}) = \overline{\beta'\ \texttt{f}'} \\ \mathcal{C}(\texttt{C}) = \texttt{class C extends C}'\ \{\overline{F};\ \overline{M}\} \\ \text{if } F_i = \alpha\ \texttt{f} \text{ then } \beta_i = \alpha \\ \text{if } F_i = \texttt{inferred C}''\texttt{f} \text{ then } \beta_i = \texttt{C}''\texttt{@l}\end{array}}{\mathit{fields}(\texttt{C}, \texttt{l}) = \overline{\beta'\ \texttt{f}'}, \overline{\beta\ \texttt{f}}}\ \textsc{A-Fields-2}$$

$$\frac{\mathit{fields}(\texttt{C}, \texttt{l}) = \overline{\alpha\ \texttt{f}} \qquad \alpha_i\ \texttt{f}_i = \beta\ \texttt{f}'}{\mathit{typeOfField}(\texttt{C}, \texttt{l}, \texttt{f}') = \beta}\ \textsc{A-FieldType}$$

$$\frac{\begin{array}{c}\mathcal{C}(\texttt{C}) = \texttt{class C extends C}'\ \{\overline{F};\ \overline{M}\} \\ \beta\ \texttt{m}(\alpha\ \texttt{x})\{\ \texttt{return}\ e\ \} \in \overline{M}\end{array}}{\mathit{method}(\texttt{C}, \texttt{m}) = \beta\ \texttt{m}(\alpha\ \texttt{x})\{\ \texttt{return}\ e\ \}}\ \textsc{A-Method-1} \qquad \frac{\begin{array}{c}\mathcal{C}(\texttt{C}) = \texttt{class C extends C}'\ \{\overline{F};\ \overline{M}\} \\ \beta\ \texttt{m}(\alpha\ \texttt{x})\{\ \texttt{return}\ e\ \} \notin \overline{M} \\ \mathit{method}(\texttt{C}', \texttt{m}) = \beta\ \texttt{m}(\alpha\ \texttt{x})\{\ \texttt{return}\ e\ \}\end{array}}{\mathit{method}(\texttt{C}, \texttt{m}) = \beta\ \texttt{m}(\alpha\ \texttt{x})\{\ \texttt{return}\ e\ \}}\ \textsc{A-Method-2}$$

Fig. 5. Auxiliary definitions for classes.

$$\frac{\begin{array}{c}\Sigma \vdash \overline{M}\ \mathit{is\ ok\ in}\ \texttt{C}\ \mathit{for}\ \texttt{l} \\ \Sigma \vdash \mathcal{C}(\texttt{C}')\ \mathit{is\ ok\ for}\ \texttt{l} \qquad \texttt{l} \neq \texttt{Local}\end{array}}{\Sigma \vdash \texttt{class C extends C}'\ \{\overline{F};\ \overline{M}\}\ \mathit{is\ ok\ for}\ \texttt{l}}\ \textsc{T-Class-1}$$

$$\frac{\begin{array}{c}\Sigma \vdash \overline{M}\ \mathit{is\ ok\ in}\ \texttt{C}\ \mathit{for}\ \texttt{Inconsistent} \\ \Sigma \vdash \mathcal{C}(\texttt{C}')\ \mathit{is\ ok\ for}\ \texttt{Local}\end{array}}{\Sigma \vdash \texttt{class C extends C}'\ \{\overline{F};\ \overline{M}\}\ \mathit{is\ ok\ for}\ \texttt{Local}}\ \textsc{T-Class-2}$$

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

Fig. 6. Method and class typing.

is part of the program definition and statically defines for each location ρ the class and availability level of an object at ρ.

Auxiliary definitions. Figure 5 shows the auxiliary definitions for the static semantics. *fields*(C, l) returns a sequence of all field declarations F for an object of class C instantiated with availability level l. *fields* differentiates between the two kinds of field definitions: If $F = \beta$ f, then *fields* just uses the declared type β for field f. Otherwise $F =$ inferred C f, in which case the type of f is C@l, that is the declared class C labeled with the availability level l of the instantiating object. *typeOfField*(C, l, f) looks up of the type of field f in C, and *method*(C, m) looks up the method declaration of m in C. These two definitions are only defined if C $\in$ dom($\mathcal{C}$).

Typing of classes. Typing for methods and classes is in Figure 6. The typing judgment for classes is $\Sigma \vdash D$ *is ok for* l, where Σ is a global type context, D is the class to be checked, and l is the availability level of the instantiating object. l 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 l 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.

The typing judgment for methods is $\Sigma \vdash M$ *is ok in* C *for* l, where Σ is a global type context, M is the method definition, C is the defining class, and l is the availability level of the 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.

$$\frac{}{\Sigma;\Gamma,\mathtt{x}\mapsto\alpha\vdash\mathtt{x}\,:\,\alpha}\ \textsc{T-Var}\qquad\frac{\Sigma;\Gamma\vdash e_1\,:\,\alpha\qquad\Sigma;\Gamma,\mathtt{x}\mapsto\alpha\vdash e_2\,:\,\beta}{\Sigma;\Gamma\vdash\mathtt{let}\ \mathtt{x}=e_1\ \mathtt{in}\ e_2\,:\,\beta}\ \textsc{T-Let}$$

$$\frac{\Sigma(\rho)<:\mathtt{C@l}}{\Sigma;\Gamma\vdash\mathtt{ref<C@l>}(\rho)\,:\,\mathtt{C@l}}\ \textsc{T-Ref}$$

$$\frac{\begin{array}{ccc}\Sigma(\rho)=\mathtt{C_0@l} & \mathtt{C}<:_D\mathtt{C_0} & \Sigma\vdash\mathcal{C}(\mathtt{C})\ \textit{is ok for}\ \mathtt{l}\\ \Sigma;\Gamma\vdash\overline{e}\,:\,\overline{\alpha} & \textit{fields}(\mathtt{C},\mathtt{l})=\overline{\beta\ \mathtt{f}} & \overline{\alpha}<:\overline{\beta}\end{array}}{\Sigma;\Gamma\vdash\mathtt{new}\ \mathtt{C@l}(\overline{e})\ \mathtt{at}\ \rho\,:\,\mathtt{C@l}}\ \textsc{T-New}\qquad\frac{\begin{array}{c}\Sigma;\Gamma\vdash e_0\,:\,\mathtt{C_0@l_0}\\ \textit{typeOfField}(\mathtt{C_0},\mathtt{l_0},\mathtt{f})=\mathtt{C@l}\end{array}}{\Sigma;\Gamma\vdash e_0.\mathtt{f}\,:\,\mathtt{C@(l_0\sqcup l)}}\ \textsc{T-FieldRead}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma\vdash e_0\,:\,\mathtt{C_0@l_0}\\ \textit{typeOfField}(\mathtt{C_0},\mathtt{l_0},\mathtt{f})=\mathtt{C@l}\\ \Sigma;\Gamma\vdash e\,:\,\alpha\qquad\alpha<:\mathtt{C@l}\end{array}}{\Sigma;\Gamma\vdash e_0.\mathtt{f}=e\,:\,\alpha}\ \textsc{T-FieldWrite}\qquad\frac{\begin{array}{c}\textit{method}(\mathtt{C_0},\mathtt{m})=\beta\ \mathtt{m}(\alpha\ \mathtt{x})\{\ \mathtt{return}\ e'\ \}\\ \Sigma;\Gamma\vdash e_0\,:\,\mathtt{C_0@l_0}\qquad\Sigma;\Gamma\vdash e\,:\,\gamma\qquad\gamma<:\alpha\end{array}}{\Sigma;\Gamma\vdash e_0.\mathtt{m}(e)\,:\,\beta}\ \textsc{T-Invoke}$$

Fig. 7. Typing for expressions.

$$\frac{P=e_1,\dots,e_n\qquad\Sigma;\bullet\vdash\overline{e}\,:\,\overline{\alpha}}{\Sigma\vdash P\ \mathtt{is\ ok}}\ \textsc{T-Prog}$$

Fig. 8. Typing for programs.

Typing of expressions. Figure 7 defines the typing rules for expressions. The 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 $\Gamma : \mathit{Var}\longrightarrow\alpha$, which maps variables to their types.

The type rules T-Var and T-Let are standard. T-Ref states that the type of a reference to a store location ρ has to be a super type of the declared type of the store location $\Sigma(\rho)$. 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 $\mathtt{l}$ and that the parameters are well-typed according to the fields of the class $\mathtt{C}$. The order of the expressions has to match the order of fields. Note that *fields* is only defined when $\mathtt{C}\in\mathrm{dom}(\mathcal{C})$ or $\mathtt{C}=\mathtt{Object}$, so only classes that are available in the class table or `Object` can be instantiated. For T-FieldRead, the resulting type is the declared class of the field with the least-upper bound of the availability levels $\mathtt{l_0}$ of the receiver and $\mathtt{l}$ of the field. For T-FieldWrite, the right hand side has to be a subtype of the declared type of $\mathtt{f}$. The field write expression returns the assigned expression e, thus the return type is α. T-Invoke is similar.

Typing of programs. The typing rule for programs is shown in Figure 8. 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 $\rho\in\mathit{Loc}$. Objects belong to a class $\mathtt{C}$ and hold a sequence of values $\overline{v}$ one for each field of $\mathtt{C}$, in the same order: $o\in\mathit{Obj} ::= \mathtt{obj}\ \mathtt{C}(\overline{v})$.

Objects are stored in *replicas* $\omega : \mathit{Loc}\longrightarrow(\mathit{ALevel}\times\mathit{Obj})$, which hold for each location $\rho\in\mathit{Loc}$ an instance of an object together with its availability level $\mathtt{l}\in\mathit{ALevel}$. Objects local to one replica has availability level `Local`. A (replicated) *store* $\Omega=\omega_1,\dots,\omega_n$ 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.

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

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

Fig. 9. Auxiliary definitions for objects.

Enforcing consistency. As we want to reason about objects in a store with different availability levels and thus consistency guarantees, we define the consistency of a store.

Definition 4.1. A store $\Omega = \overline{\omega}$ is *fully-consistent*, iff $\forall \rho, i, j.\ \omega_i(\rho) = (\texttt{l}, \ldots) = \omega_j(\rho) \land \texttt{l} \neq \texttt{Local}$.

With full consistency, all replicas of the store contain the same objects but local objects. Weaker models relax this constraint. Predicate $\textit{consistent}_{\texttt{l}}(\Omega, k, \rho)$ is satisfied if the object at location ρ in replica $\omega_k \in \Omega$ is consistent to the complete store Ω w.r.t. the availability level $\texttt{l}$. With the predicate, we control when operations with availability $\texttt{l}$ 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:

$$\forall \Omega, k, \texttt{l}, \rho.\ \Omega \text{ fully-consistent} \Rightarrow \textit{consistent}_{\texttt{l}}(\Omega, k, \rho) \tag{1}$$

Predicate $\textit{consistent}_{\texttt{Local}}$ always holds because local operations do not rely on the replicated store – hence the consistency is not relevant. The predicate $\textit{consistent}_{\texttt{l}}$ models the coordination among replicas required for availability level $\texttt{l}$. 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 $\textit{consistent}_{\texttt{l}}$ captures the semantics of the availability level $\texttt{l}$, abstracting over the precise definition of the consistency model, i.e., over details like operations order and message propagation. For example, the definition of sequential consistency requires, amongst others, the order of operations of a single process. Store models that can capture such order can be defined, e.g., as a history of operations [37] and are orthogonal to our work.

4.4 Dynamic semantics

We define a small-step operational semantics for the core calculus with transition relations for processes and programs. The semantics relies on the auxiliary definitions in Figure 9. $\ulcorner o.\texttt{f} \urcorner$ reads the value of a field $\texttt{f}$ from an object o, and $\ulcorner o.\texttt{f} := v \urcorner$ defines an object o with field $\texttt{f}$ set to v. In the rules, the notation $e_1[\texttt{x} \leftarrow e_2]$ indicates the substitution of $\texttt{x}$ with e_2 in e_1.

Local transition rules. We first define the semantics of single processes (Figure 10). *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 judgment $\Omega; k \vdash \langle \omega \mid e \rangle \longrightarrow \langle \omega' \mid e' \rangle$ states that process k with configuration $\langle \omega \mid e \rangle$ makes a step to configuration $\langle \omega' \mid e' \rangle$ when the whole replicated store has the state Ω. k uniquely identifies the process. Transitions of process configurations do not change the store Ω, but only the local replica ω. Ω is only used with the *consistent* predicate to check whether the local replica is consistent.

Rule Context induces a call-by-value, left-to-right evaluation through an evaluation context E. Rule Let replaces the variable $\texttt{x}$ in e_2 with value v as usual. New creates an object store location ρ in ω with class $\texttt{C}$ and availability $\texttt{l}$. Location ρ must be free, i.e., there is no other object in ρ. FieldRead returns the value of the field $\texttt{f}$ of object ρ. The returned value is a reference (references are the only values in the calculus). The premise ensures that the replicated store Ω is consistent with respect to the availability $\texttt{l}'$ of the stored object, i.e., a field can only be read if the requirements

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

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

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

$$\frac{\omega(\rho_0) = (\texttt{l}', \texttt{obj C}'(\overline{v})) \qquad \ulcorner \texttt{obj C}'(\overline{v}).\texttt{f} \urcorner = \texttt{ref<C@l>}(\rho) \qquad \textit{consistent}_{\texttt{l}'}(\Omega, k, \rho_0)}{\Omega; k \vdash \langle \omega \mid \texttt{ref<C}_0\texttt{@l}_0\texttt{>}(\rho_0).\texttt{f} \rangle \longrightarrow \langle \omega \mid \texttt{ref<C@(l}_0 \sqcup \texttt{l)>}(\rho) \rangle}\ \textsc{FieldRead}$$

$$\frac{\begin{array}{c}\omega(\rho_0) = (\texttt{l}', \texttt{obj C}'(\overline{v})) \\ \ulcorner \texttt{obj C}'(\overline{v}).\texttt{f} \coloneqq v \urcorner = o' \qquad \omega' = \omega, \rho \mapsto (\texttt{l}', o') \qquad \textit{consistent}_{\texttt{l}'}(\Omega, k, \rho_0)\end{array}}{\Omega; k \vdash \langle \omega \mid \texttt{ref<C}_0\texttt{@l}_0\texttt{>}(\rho_0).\texttt{f} = v \rangle \longrightarrow \langle \omega' \mid v \rangle}\ \textsc{FieldWrite}$$

$$\frac{\omega(\rho_0) = (\texttt{l}', \texttt{obj C}'(\overline{v})) \qquad \textit{method}(\texttt{C}', \texttt{m}) = \beta\ \texttt{m}(\alpha\ \texttt{x})\{\ \texttt{return}\ e\ \} \qquad \textit{consistent}_{\texttt{l}'}(\Omega, k, \rho_0)}{\Omega; k \vdash \langle \omega \mid \texttt{ref<C}_0\texttt{@l}_0\texttt{>}(\rho_0).\texttt{m}(v) \rangle \longrightarrow \langle \omega \mid e[\texttt{x} \leftarrow v][\texttt{this} \leftarrow \texttt{ref<C}'\texttt{@l}'\texttt{>}(\rho_0)] \rangle}\ \textsc{Invoke}$$

Fig. 10. Operational semantics: Local transitions.

$$\frac{\Omega = \omega_1, \ldots, \omega_n \qquad \Omega; k \vdash \langle \omega_k \mid e_k \rangle \longrightarrow \langle \omega'_k \mid e'_k \rangle}{\ldots, \langle \omega_k \mid e_k \rangle, \ldots \rightsquigarrow \ldots, \langle \omega'_k \mid e'_k \rangle, \ldots}\ \textsc{G-Local}$$

$$\frac{\begin{array}{c}\omega_j(\rho) = (\texttt{l}, o_j) \qquad \omega_k(\rho) = (\texttt{l}, o_k) \qquad \texttt{l} \neq \texttt{Local} \qquad o_j \neq o_k \\ o' = \textit{merge}_{\texttt{l}}(\Omega, o_j, o_k) \qquad \omega'_j = \omega_j, \rho \mapsto (\texttt{l}, o') \qquad \omega'_k = \omega_k, \rho \mapsto (\texttt{l}, o')\end{array}}{\ldots, \langle \omega_j \mid e_j \rangle, \ldots, \langle \omega_k \mid e_k \rangle, \ldots \rightsquigarrow \ldots, \langle \omega'_j \mid e_j \rangle, \ldots, \langle \omega'_k \mid e_k \rangle, \ldots}\ \textsc{G-Converge}$$

$$\frac{\omega_j(\rho) = (\texttt{l}, o_j) \qquad \omega_k(\rho) \text{ undefined} \qquad \texttt{l} \neq \texttt{Local} \qquad \omega'_k = \omega_k, \rho \mapsto (\texttt{l}, o_j)}{\ldots, \langle \omega_j \mid e_j \rangle, \ldots, \langle \omega_k \mid e_k \rangle, \ldots \rightsquigarrow \ldots, \langle \omega_j \mid e_j \rangle, \ldots, \langle \omega'_k \mid e_k \rangle, \ldots}\ \textsc{G-Replicate}$$

Fig. 11. Operational semantics: Global transitions.

of the availability level are met. FieldWrite changes to v the `f` field of the object o in replica ω at ρ 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 ρ_0 and resolves its class to retrieve the method `m`. Again, a consistency check is performed to execute the method.

Global transition rules. 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 11. The program takes a step if one of the processes does a local transition (G-Local).

Rule G-Converge and G-Replicate coordinate the replicas. The function $\textit{merge}_{\texttt{l}}(\Omega, o_1, o_2)$ for each availability level `l` returns the merged state of two objects, given the current store Ω. Like the predicate *consistent*, *merge* is defined differently for each consistency level `l`. If *merge* is undefined for two objects, they can not be merged given the current state of store Ω. This is case when certain consistency or isolation guarantees are not fulfilled, e.g., an operation on object o which is part of a method invocation can only be merged when all operations that are part of the invocation are visible. We require that if *merge* is defined o_1 and o_2 and the returned object have the same class.

$$\forall \Omega, \texttt{l}.\ \textit{merge}_{\texttt{l}}(\Omega, \texttt{obj C}_1(\ldots), \texttt{obj C}_2(\ldots)) = \texttt{obj C}_3(\ldots) \Rightarrow \texttt{C}_1 = \texttt{C}_2 = \texttt{C}_3 \tag{2}$$

In G-Converge, *merge* defines how replicas converge. When two replicas ω_j and ω_k store different objects at location ρ, the objects are merged and the result is stored in both replicas at ρ.

Objects with availability `Local` can not be merged as local objects are not replicated. G-Replicate replicates objects that are not available on a replica ω_k to that replica.

4.5 Properties

In this section we discuss the properties of the core calculus and provide a sketch of the proofs. The full proofs are in Appendix **??**. First, we give the definition of well-defined class tables.

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

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 for the calculus C is well-defined. Next, we relate program expressions to the class table C.

Definition 4.3. An expression e is *well-defined*, iff $\forall \texttt{C}$ in e, $\texttt{C} = \texttt{Object} \vee \texttt{C} \in \text{dom}(C)$.

Similar to well-defined 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 4.4. Object $o = \texttt{obj C}(\overline{v})$ is *well-typed with* `l` *in* Σ, iff

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

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

Definition 4.5. Replica ω *satisfies* Σ, iff (1) $\text{dom}(\omega) \subseteq \text{dom}(\Sigma)$,
and (2) $\Sigma(\rho) = \texttt{C@l} \wedge \omega(\rho) = (\texttt{l}', \texttt{obj C}'(\ldots)) \Rightarrow \texttt{obj C}'(\ldots)$ well-typed with `l` in $\Sigma \wedge \texttt{l} = \texttt{l}' \wedge \texttt{C} = \texttt{C}'$.

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 show the following lemma that relates substitution and type environments.

Lemma 4.6. *If* $\Sigma; \Gamma, \texttt{x} \mapsto \alpha \vdash e : \beta$ *and* $\Sigma; \Gamma \vdash e' : \alpha'$ *and* $\alpha' <: \alpha$, *then* $\Sigma; \Gamma \vdash e[\texttt{x} \leftarrow e'] : \beta'$ *and* $\beta' <: \beta$.

Proof. By induction over derivations of $\Sigma; \Gamma \vdash e : \alpha$. □

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.

Theorem 4.7 (Preservation of expressions). *If* $\Sigma; \Gamma \vdash e : \alpha$ *and* $\Omega; k \vdash \langle \omega \mid e \rangle \longrightarrow \langle \omega' \mid e' \rangle$ *and* ω *satisfies* Σ, *then for some* $\Sigma' \supseteq \Sigma$ *and* $\beta <: \alpha$ *holds* $\Sigma'; \Gamma \vdash e' : \beta$ *and* ω' *satisfies* Σ'.

Proof. By induction over derivations of $\Omega; k \vdash \langle \omega \mid e \rangle \longrightarrow \langle \omega' \mid e' \rangle$ and with Lemma 4.6. □

We have now shown that single processes feature type preservation. The following theorem states preservation of full programs.

Theorem 4.8 (Preservation of programs). *Let* $P = e_1, \ldots, e_n$ *and* $\Omega = \omega_1, \ldots, \omega_n$.
If $\Sigma \vdash e_1, \ldots, e_n$ `is ok` *and* $\langle e_1 \mid \omega_1 \rangle, \ldots, \langle e_n \mid \omega_n \rangle \rightsquigarrow \langle e_1' \mid \omega_1' \rangle, \ldots, \langle e_n' \mid \omega_n' \rangle$ *and* $\overline{\omega}$ *satisfies* Σ, *then for some* $\Sigma' \supseteq \Sigma$ *holds* $\Sigma' \vdash e_1', \ldots, e_n'$ `is ok` *and* $\overline{\omega'}$ *satisfies* Σ'.

PROOF. By induction over derivations of $\langle e_1 \mid \omega_1\rangle, ..., \langle e_n \mid \omega_n\rangle \rightsquigarrow \langle e'_1 \mid \omega'_1\rangle, ..., \langle e'_n \mid \omega'_n\rangle$. Follows by preservation of expressions. G-CONVERGE does not change the type of objects in the replicas ω_i. G-REPLICATE only adds entries to a replica that have been shown to satisfy Σ. □

4.5.2 Progress. Progress specifies that a well-typed program that is not a value, can make a reduction step [39]. 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 as in such case we show that processes waiting for consistency eventually perform a step when the consistency condition is satisfied. We first define unique locations for (sequences of) expressions.

Definition 4.9. $\overline{e}$ has *unique locations*, iff for the sequence $\overline{e'}$ of all subexpressions of $\overline{e}$ holds: $e'_i = \texttt{new}\ \texttt{C}_i\texttt{@l}_i(...)\ \texttt{at}\ \rho_i \wedge e'_j = \texttt{new}\ \texttt{C}_j\texttt{@l}_j(...)\ \texttt{at}\ \rho_j \wedge i \neq j \Rightarrow \rho_i \neq \rho_j$.

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

THEOREM 4.10 (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 $e = \texttt{ref<C}_0\texttt{@l}_0\texttt{>}(\rho_0).\texttt{f}$, then $\rho_0 \notin \text{dom}(\omega)$ or $\neg consistent_{\texttt{l}'}(\Omega, k, \rho_0)$ with $\Sigma(\rho_0) = \texttt{C}'\texttt{@l}'$ for some $\texttt{C}'$ or $\Omega; k \vdash \langle\omega \mid e\rangle \longrightarrow \langle\omega' \mid e'\rangle$ for some e' and ω'.
(2) if $e = (\texttt{ref<C}_0\texttt{@l}_0\texttt{>}(\rho_0).\texttt{f} = v)$, then $\rho \notin \text{dom}(\omega)$, or $\neg consistent_{\texttt{l}'}(\Omega, k, \rho_0)$ with $\Sigma(\rho_0) = \texttt{C}'\texttt{@l}'$ for some $\texttt{C}'$ or $\Omega; k \vdash \langle\omega \mid e\rangle \longrightarrow \langle\omega' \mid e'\rangle$ for some e' and ω'.
(3) if $e = \texttt{ref<C}_0\texttt{@l}_0\texttt{>}(\rho_0).\texttt{m}(v)$, then $\rho \notin \text{dom}(\omega)$ or $\neg consistent_{\texttt{l}'}(\Omega, k, \rho_0)$ with $\Sigma(\rho_0) = \texttt{C}'\texttt{@l}'$ for some $\texttt{C}'$ or $\Omega; k \vdash \langle\omega \mid e\rangle \longrightarrow \langle\omega' \mid e'\rangle$ for some e' and ω'.
(4) else $e \in Value$ or $\Omega; k \vdash \langle\omega \mid e\rangle \longrightarrow \langle\omega' \mid e'\rangle$ for some e' and ω'.

PROOF. By induction on type derivations $\Sigma; \Gamma \vdash e : \alpha$. □

Progress for *expressions* is only defined if the processes are not waiting for other processes to either replicate an object to a location ρ 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.

LEMMA 4.11 (PROGRESS OF PROGRAMS). *Let $P = e_1, ..., e_n$ and $\Omega = \omega_1, ..., \omega_n$. If $\Sigma \vdash P$ $\texttt{is ok}$, and $\overline{e}$ is well-defined and has unique locations, and ω satisfies Σ, and $\forall\texttt{C}.\ merge_1(\Omega, \texttt{obj C}(...), \texttt{obj C}(...))$ defined, then $\forall i.\ e_i \in Value$ or $\langle e_1 \mid \omega_1\rangle, ..., \langle e_n \mid \omega_n\rangle \rightsquigarrow \langle e'_1 \mid \omega'_1\rangle, ..., \langle e'_n \mid \omega'_n\rangle$.*

PROOF. By induction on derivations of $\langle e_1 \mid \omega_1\rangle, ..., \langle e_n \mid \omega_n\rangle \rightsquigarrow \langle e'_1 \mid \omega'_1\rangle, ..., \langle e'_n \mid \omega'_n\rangle$. For rule G-LOCAL, we use progress of expressions, but if it can not be applied, then (1) all expressions are values already or (2) an object of location ρ is not available in a replica, i.e. $\rho \notin \text{dom}(\omega_i)$ for some i, or (3) the predicate *consistent* is not satisfied. If (2), then the rule G-REPLICATE can be applied to define the location ρ in the replica. If (3), then rule G-CONVERGE can be applied to make replicas consistent and satisfy *consistent* because of equation 1. G-CONVERGE can be applied, because *merge* is defined since the classes of the merged objects are equal as follows from ω_i satisfies Σ. □

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 that are given by the isolation, hence violating isolation guarantees.

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 to be indistinguishable.

Definition 4.12. Objects $o_1 = \mathtt{obj}\ \mathtt{C}_1(\overline{v})$ and $o_2 = \mathtt{C}_2\overline{v}'$ are *indistinguishable for* $\mathtt{l}$ *when instantiated as* $\mathtt{l}'$, iff $\mathtt{C}_1 = \mathtt{C}_2$ and with $\mathit{fields}(\mathtt{C}_1, \mathtt{l}_0) = \overline{\mathtt{C@l\ f}}$ for all i holds: $v_i = v_i'$ if $\mathtt{l}_i \sqsubseteq \mathtt{l}$.

Definition 4.13. Replicas ω_1 and ω_2 are *indistinguishable for* $\mathtt{l}$, iff for all $\mathtt{l}' \sqsubseteq \mathtt{l}$ such that $\mathtt{l}' \neq \mathtt{Local}$ holds: $\omega_1(\rho) = (\mathtt{l}', o_1)$ and $\omega_2(\rho) = (\mathtt{l}', o_2)$ and o_1 and o_2 are indistinguishable for $\mathtt{l}$ when instantiated as $\mathtt{l}'$. We write $\omega_1 \approx_{\mathtt{l}} \omega_2$.

Now, we can define non-interference for expressions.

Theorem 4.14 (Non-interference). *If* $\Sigma; \Gamma \vdash e : \alpha$, *and* $\omega_1 \approx_{\mathtt{l}} \omega_2$, *and* ω_1, ω_2 *satisfy* Σ, *and* $\Omega; k \vdash \langle \omega_1 \mid e \rangle \longrightarrow^* \langle \omega_1' \mid v_1 \rangle$, *and* $\Omega; k \vdash \langle \omega_2 \mid e \rangle \longrightarrow^* \langle \omega_2' \mid v_2 \rangle$, *then* $\omega_1' \approx_{\mathtt{l}} \omega_2'$.

Proof. By induction over number of steps in $\Omega; k \vdash \langle \omega_1 \mid e \rangle \longrightarrow^* \langle \omega_1' \mid v_1 \rangle$. □

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 ∼ 4,800 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 [31]. 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 [2] 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.

(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) 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) 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() {
4     return soldTickets.ref.inc(); // Seq Op
5 }}
6 Ref<@Ev Concert> concert = ...
7 concert.ref.incSoldTickets(); // Ev Op
```

Since `concert` is *Ev*, the `incSoldTickets` call (Line 7) 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 two configurations: All-Seq 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. The Mixed configuration uses *High* available data (either *Ev* or *Cau* depending on the use case) and *Seq* availability. The type system ensures the correctness of the mixing. We measure the runtime gain when using Mixed instead of All-Seq.

The benchmarks are executed on nine processes. For each case study, one instance hosts the master replicas. The other machines host follower replicas that perform distributed operations.

Counter. We implemented a replicated integer counter [19, 29]. 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*.

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. 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 [29]: 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*. The master replicas are evenly distributed among followers, each hosting 500 groups, for a total of 4 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*.

IPA Twitter Clone. We reimplemented the Retwis twitter clone from the IPA paper [19] using ConSysT. Retwis supports operations like tweeting, retweeting, following and unfollowing. Users and tweets are *Ev*-replicated objects. A replicated counter with *Seq* availability counts the retweets for each tweet.

Results. We measured the run time for executing a batch of operations 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 regularly manually synchronize the objects. Table 1 shows the run time of as single operation and the number of (*Seq*) objects in each case study. Figure 12 shows the run time of the Mixed configuration relative to the All-Seq baseline.

Table 1. Performance metrics.

Case Study	Runtime (ms)						# of objects	
	Local		Datacenter		Geo-Replicated			
	MIXED	ALL-SEQ	MIXED	ALL-SEQ	MIXED	ALL-SEQ	all	*Seq*
Counter	14.42	90.40	9.24	63.40	28.57	1 642.19	1	0
TicketShop	177.70	184.34	121.08	146.14	5 481.82	7 444.31	4	3
Message Groups	29.38	81.89	10.41	25.60	213.35	1 307.00	13 500	4 500
E-Commerce	524.90	1 376.43	378.89	1 489.69	6 214.13	88 328.66	3 320	2 000
Twitter Clone	20.03	71.52	5.64	18.73	154.16	863.81	27 000	9 000

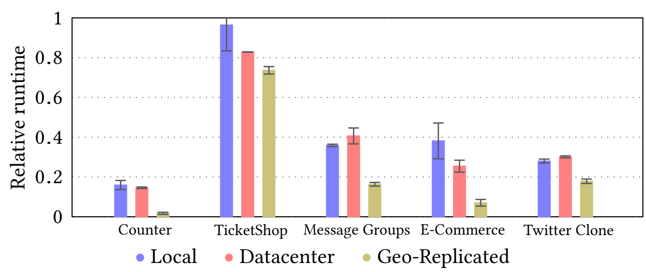

Fig. 12. MIXED compared to ALL-SEQ.

1.0 is the run time of the ALL-SEQ configurations. We executed the benchmarks in three different setups: (i) 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-distributed on the same AWS machines: 3 instances in EU (London), and 2 instances in US East(Ohio), Asia Pacific (Mumbai), and Canada (Central), each.

In *Counter*, the MIXED configuration only takes 1.7 % – 16 % of the time of the baseline. As there is only one replicated object, the counter, all non-blocking synchronization occurs asynchronously in the *Ev* case, improving the run time. In the *Seq* case, however, the followers have to block and synchronize for every operation. 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, which results in a modest speed-up compared to all objects set to *Seq*. For *Message Groups*, we measured two MIXED configurations using *Ev* and *Cau* availability levels, respectively, as *Cau* preserves the order of (causally) related messages. For *Ev* (blue/left bar), the run time is 16 % – 40 % of the baseline. The comparabale high speed-up of *High* operations in the geo-distributed setup comes from the high latency between replicas, which increases the wait time for blocking synchronization. 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. For ALL-SEQ, these operations are blocking and very time-consuming, i.e., 88s for a single operation. 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 (Table 1).

The high absolute run times in some case studies are explained by the fact that the *Seq* availability level requires serializability, which is very slow in a high latency (geo-distributed) setup.

The results we discussed so far represent an extreme case in which applications continuously perform operation, resulting in high contention on locks for *Low*. To measure the effect of contention,

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

```
// Centralized component
Result handleRequest(name: String, request: Request) {
  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. 13. Access to `soldTickets` in TicketShop.

Table 2. Effect of contention.

Delay	Runtime (ms)	
	Mixed	All-Seq
10 ms	0.24	27.43
100 ms	0.36	12.36

we also repeated the *Counter* case study by adding a delay between operations. The results are presented in Table 2. We measured the runtime of an operation if there is a 10ms or a 100ms delay, respectively, between operations. In the Mixed configuration, we measured 0.24ms and 0.36ms per operation for the cases with 10ms and 100ms delay, respectively. In the All-Seq configuration, we measured 27.43ms and 12.36ms for 10ms and 100ms delay. This shows that Mixed provides significant benefits on latency even in the case of low contention.

In summary, mixing availability levels brings a a large speed-up compared to a version with only *Low* available objects.

6.2 Application Design

To evaluate the effects of ConSysT on software design, we analyse three 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 2). Transactions are introduced by `mixt_method` (Line 3). The code iterates over all users

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

(a) MixT.

```
public class Group {
  public Ref<@Seq List<Ref<@Cau User>> users;
    public void addPost(String post) {
      var iterator = users.ref.iterator()
      while (iterator.hasNext())
        iterator.next().ref.inbox.insert(post); } }
```

(b) ConSysT.

Fig. 14. Message Groups.

(Line 4–7) and adds a post to every inbox (Line 6). In MixT, (i) transactions must be written in a DSL embedded into C++ code (Line 3–7), (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 user code. Version *v2* uses ConSysT abstractions to define *Seq* replicated objects. Figure 13a and 13b 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 13a, replicas explicitly synchronize with each other by sending `SerializeOperations` requests to a central component 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.

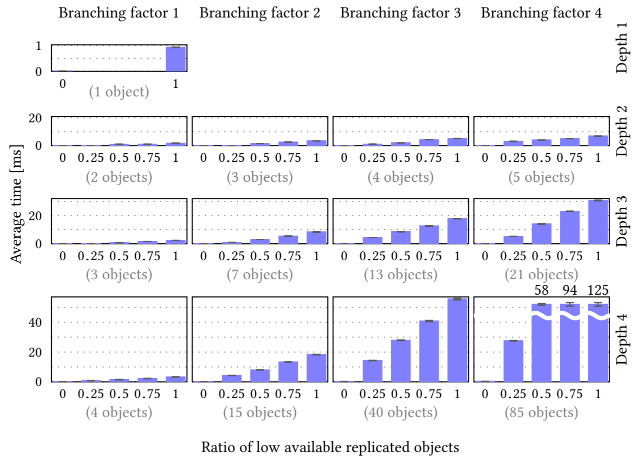

Fig. 15. Microbenchmarks.

6.3 Microbenchmarks

To 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 *Seq* and *Ev* objects between 0 and 100 %, choosing the 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 15 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.

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 datastores. Then we focus on language abstractions to reason about consistency and availability.

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 [37] provide a survey on consistency in non-transactional distributed storage. Adya et al. [1] 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 datastores with acceptable performance, e.g., Google Spanner [13], exploiting the availability of accurate clocks, and Calvin [36], relying on the deterministic execution of transactions to reduce overhead. Other databases, e.g., H-Store [35] and ReactDB [32], make distribution – for its performance effects – explicit in the schema and query language.

Another line of work focuses on data types tailored for weakly consistent models such as eventual consistency. CRDTs [33] are distributed data types with an API that only allows commutative operations, ensuring that replicas eventually converge. Cloud types [9] provide eventually consistent storage and define an abstraction layer over recurring engineering challenges like Web service implementation, communication protocols, and caching. They are similar to CRDTs, but also supports non-commutative operations. Zhao and Haller [42] propose a consistency protocol which combines the performance of mergeable data types like CRDTs, and reliable total order broadcast for strong consistency. The resulting abstractions provide high availability through mergeable data types as well as acceptable latency for strong consistency.

The idea of mixing multiple consistency/availability levels within the same system was pioneered by RedBlue consistency [25]: 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 [26] 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 [40] propose a continuous consistency model to dynamically trade consistency for availability where applications specify a maximum deviation from strong consistency for each replica.

An important research line concerns finding criteria for ensuring that an application exhibits a certain consistency level. For example, Brutschy et al. [8] provide an analysis for serializability in distributed applications. SIEVE [24] 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 [28] relax the synchronization of strong consistent data by predicting how results of strong operations change over time based on the history of the program.

7.2 Abstractions for Consistency and Availability

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

MixT [29] is a C++ DSL for transactions that span multiple datastores each with a different consistency level. The compiler automatically distributes each transaction across the datastores. Similar to ConSysT, MixT adopts an information-flow type system to safely mixing multiple consistency models. MixT is a separate domain-specific language for database transactions. Operations on replicated objects can only be executed inside MixT code. In MixT, only transactions written in the MixT DSL can manipulate replicated objects. The available operations depend on custom operations supported by the specific underlying datastore for a specific datatype. In contrast, ConSysT integrates availability with OO programming and lets developers uniformly operate on replicated objects. Another crucial semantic difference is that 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 [41]. 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 [30] 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. [19] 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 (and MixT) do using 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 [7] is an actor based framework that implements the *virtual actor model* [11], 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 [15] 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. *Correctables* [17] 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 [23] 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 [34] 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 [6], 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 [20] 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 [33] 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. [4] propose delta-state CRDTs that only propagate incremental state changes. Burckhardt et al. [10] 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 [9] 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 [42] combine in a protocol the performance of mergeable data types and reliable total order broadcast for strong consistency. The resulting abstractions provide both high availability through mergeable data types and acceptable latency for strong consistency.

Antidote DB [3] is a database that features geo-replication, CRDTs and highly available transactions. Antidote SQL [27], 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's weak consistency does not use CRDTs. 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.

REFERENCES

[1] 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

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

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

[4] 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

[5] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations (Extended Version). (2013). https://doi.org/10.14778/2732232.2732237 arXiv:1302.0309

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

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

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ 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, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*.

[14] 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

[15] 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

[16] 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

[17] 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.

[18] 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

[19] 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

[20] 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

[21] Atsushi Igarashi. 1999. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM SIGPLAN Notices This ACM Transactions on Programming Languages and Systems* 34, 3 (may 1999), 132–146. https://doi.org/10.1145/503502.503505

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

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

[24] 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

[25] 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

[26] 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

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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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.

[33] Marc Shapiro, Nuno Pregui, Carlos Baquero, and Marek Zawirski. 2011. A Comprehensive Study of Convergent and Commutative Replicated Data Types. (2011).

[34] 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

[35] 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.

[36] 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

[37] 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

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

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

[40] 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

[41] 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

[42] 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