ConSysT: Tunable, Safe Consistency Meets Object-Oriented Programming

Mirko Köhler
Technische Universität Darmstadt
Germany
koehler@cs.tu-darmstadt.de

Nafise Eskandani Masoule
Technische Universität Darmstadt
Germany
n.eskandani@cs.tu-darmstadt.de

Alessandro Margara
Politecnico di Milano
Italy
alessandro.margara@polimi.it

Guido Salvaneschi
Universiät St. Gallen
Switzerland
guido.salvaneschi@unisg.ch

Abstract

Data replication is essential in scenarios like geo-distributed datacenters, but poses challenges for data consistency. Developers adopt Strong consistency at the cost of performance or embrace Weak consistency and face a higher programming complexity. We argue that languages should associate consistency to data types. We present ConSysT, a programming language and middleware that provides abstractions to specify consistency types, enabling mixing different consistency levels in the same application. Such mechanism is fully integrated with object-oriented programming and type system guarantees that different levels can only be mixed correctly.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**.

Keywords: replication, consistency, type systems, Java

ACM Reference Format:
Mirko Köhler, Nafise Eskandani Masoule, Alessandro Margara, and Guido Salvaneschi. 2020. ConSysT: Tunable, Safe Consistency Meets Object-Oriented Programming. In *Proceedings of the 22th ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs (FTfJP '20), July 23, 2020, Virtual, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3427761.3428346

1 Introduction

In scenarios like distributed datacenters, data replication is critical to achieve scalability, low latency and fault tolerance. Keeping replicas consistent in the presence of data modifications poses a challenge to the underlying system and developers. Recently, many consistency models have been proposed each having their own trade-offs between consistency and availability. For example, Strong consistency models, such as *Sequential Consistency*, do not allow concurrent modifications. While Strong consistency reduces programming complexity, it also reduces availability as immediate coordination is required. On the other hand, Weak consistency models defer coordination between replicas, which increases availability, but also complicates reasoning about programs as data can be temporarily inconsistent. As there is no one-size-fits-all solution, the choice of a consistency model for an application becomes complex. Weak consistency boosts availability, but Strong consistency is better when correctness is at risk. To make things worse, applications often require different consistency models, e.g., payment requires Strong consistency, whereas Weak consistency suffices for instant messaging. This is not an easy feat as developers have to (a) know the consistency models to infer the guarantees, (b) ensure that data with different consistency models is mixed correctly, and (c) reason about concurrency when mixing consistencies. We propose ConSysT, a language featuring fine-grained, data-centric specification of consistency levels. The consistency of an application is tuned by changing consistency levels in the program. ConSysT features a static type system to ensure that data with different consistency levels mix safely. It supports (weak) transactions and is integrated into object-oriented programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FTfJP '20, July 23, 2020, Virtual, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8186-4/20/07…$15.00
https://doi.org/10.1145/3427761.3428346

2 Overview

In this section, we introduces ConSysT's core concepts – distribution through replication, consistency, and correctness.

Distribution. In ConSysT, programs are divided into (logically) single-threaded *processes* running in parallel. Replicated data is modelled as *replicated objects* and each process holds its own local copy of an object. Distributed *operations* are performed by calling methods on replicated objects. Figure 1 shows how to replicate a counter in ConSysT. We start

```
1 class Counter {
2   int i;
3   Counter(int i) { this.i = i; }
4   void inc() { i = i + 1; } }
5 transaction(() -> {
6   Ref<@Sequential Counter> counter =
7   replicate("id",Sequential,Counter.class,0);
8   counter.ref().inc(); });
```

Figure 1. Running example.

$$
\begin{array}{rcl}
\mathit{Expr} \ni e & ::= & x \mid \mathsf{Ref@}\ell(\rho) \\
\mathit{Computation} \ni c & ::= & \mathsf{skip} \mid \mathsf{let}\ x := \mathsf{tx}(t)\ \mathsf{in}\ c \mid \mathsf{return}\ e \mid \ldots \\
\mathit{Transaction} \ni t & ::= & \mathsf{let}\ x := \mathsf{replicate}(\rho, \ell, C, \overline{e})\ \mathsf{in}\ t \mid t\,;t \\
 & & \mid \mathsf{let}\ x := e.m(e)\ \mathsf{in}\ t \mid \mathsf{return}\ e \mid \ldots \\
\mathit{Program} \ni P & ::= & c_1, \ldots, c_n \\
D & ::= & \mathsf{class}\ \mathsf{C}_1\ \mathsf{extends}\ \mathsf{C}_2\ \{\overline{F};\ \overline{M}\} \\
\mathit{ConsLevel} \ni \ell & ::= & \mathsf{Sequential} \mid \ldots \\
\mathit{ConsType} \ni \tau & ::= & \mathsf{C@}\ell
\end{array}
$$

Figure 2. Syntax of the core calculus.

a transaction (Line 5), then we create the replicated Counter by using `replicate` (Line 7), which returns a `Ref` to the replicated object. Names, here `"id"`, are used by other processes to refer to the object. Operations are performed using references on references. Operations are prefixed with `ref` to make remote accesses explicit. We perform the operation `inc` by calling the respective method (Line 8).

Consistency. How the operation is executed depends on the *consistency level* of the object. The level describes the consistency model – such as sequential, or causal consistency. In ConSysT, the developer has fine-grained control over the consistency of data, as every replicated object defines its own consistency level. In the example, we create a replicated `Counter` with level `Sequential` (Line 7). Operations performed on a `Sequential` replicated object are propagated using the sequential consistency model. Fields of an object have the same level as the object itself. Thus, ConSysT enables mixing replicated data in two ways: (a) an operation can contain objects with different levels, or (b) replicated objects can be nested, i.e., an object can have a field that is a reference to another replicated object with its own consistency level.

Language. We formalize our language with the syntax outlined in Figure 2. Programs P consist of processes c_n, where each process executes transactions t on a replicated store. Classes and methods are based on Featherweight Java [3]. ConSysT adopts *consistency types* τ to enable static reasoning about consistency levels. For example, the consistent level appears in the type of the replicated `Counter` in its type as `@Sequential` (Line 6 of Figure 1).

We define a replicated store model, where replicas have a transaction local state δ that is merged into a global store Δ [4]. The operational semantics defines transitions for transactions t with variable environment E: $\langle\langle t, E, \Delta, \delta, \kappa_\top\rangle\rangle \twoheadrightarrow \langle\langle t', E', \Delta', \delta', \kappa'_\top\rangle\rangle$. We use continuations $\kappa_\top$ to model waiting for concurrent transactions. Figure 3 shows the rule for replicate. The rule evaluates expressions $\overline{v}$ and creates a new

$$
\frac{E \vdash \overline{e} \Downarrow \overline{v} \qquad o = \mathsf{C}^{\ell}(\overline{v}) \qquad \delta' = \delta \cdot (\rho \mapsto o) \qquad E' = E \cdot (x \mapsto \mathsf{Ref@}\ell(\rho))}{\begin{array}{c}\langle\langle \mathsf{let}\ x := \mathsf{replicate}(\rho, \ell, \mathsf{C}, \overline{e})\ \mathsf{in}\ t, E, \Delta, \delta, \kappa_\top\rangle\rangle \\ \twoheadrightarrow \langle\langle t, E', \Delta, \delta', \kappa_\top\rangle\rangle\end{array}}
$$

Figure 3. Example transition rule.

object o which is stored in the local store δ. The local store is written to the global store Δ by the rule for transactions (omitted for brevity).

Correctness. The language supports developers to mix consistency levels correctly. In ConSysT, we formalize an information-flow type system for consistency types to ensure that Strong objects do not depend on Weak objects. Such a flow can degrade consistency [6]. The type system is parametric in the concrete consistency levels: Levels are ordered in a lattice [1, 9] that defines the subtyping relation for consistency types. We prove that in well-typed programs, Strong values cannot be affected by Weak values (*non-interference*) and that stores can only differ in consistency levels that are weaker than a given level ℓ, i.e., inconsistencies in stores can only appear in weaker consistent objects. We define two stores δ_1 and δ_2 to be *indistinguishable up to a consistency level* ℓ, when all objects in the store with at least the level ℓ are equivalent. The non-interference property then states that two stores that are indistinguishable before the execution of a well-typed program are indistinguishable after the execution of the program, giving us the guarantee that inconsistencies only appear in weaker levels.

3 Related Work

Mixing consistency has been tackled in several works. Holt et al. [2] use types and type safety to imply consistency safety. Their type system does not consider control dependencies. MixT [6] is a DSL for transactions over multiple datastores with different consistency levels. A type system enforces correct mixing of levels. In contrast, ConSysT integrates consistency levels into an object-oriented programming model and does not assume different semantics for datastores. Instead of using consistency levels on *data*, another approach is to annotate *operations*. RedBlue Consistency [5] defines consistency levels red (Strong) and blue (Weak) for operations. Red operations can violate invariants if executed concurrently. In Quelea [8] developers define invariants on functions which guarantee correct ordering of operations. Gallifrey [7] is a language for replicated objects where restrictions are defined as conditions on operations. Instead, in ConSysT, the data-based approach alleviates the definition of conditions.

Acknowledgments

This work has been supported by the LOEWE centre emergenCITY and by the German Reasearch Foundation (DFG), by the DFG projects SA 2918/2-1 and SA 2918/3-1, and by the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *PVLDB*.

[2] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types *(SoCC '16)*. ACM, 15.

[3] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. 1999. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 132–146. https://doi.org/10.1145/320384.320395

[4] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2017. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *Proc. ACM Program. Lang.* 2 (2017), 27:1–27:34.

[5] Cheng Yen Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary *(OSDI '12)*. USENIX.

[6] Matthew Milano and Andrew C. Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions *(PLDI '18)*. ACM.

[7] Matthew Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers. 2019. A Tour of Gallifrey, a Language for Geodistributed Programming *(SNAPL '19)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores *(PLDI '15)*. ACM.

[9] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *CSUR* (July 2016).