ConSysT: Tunable, Safe Consistency meets Object-Oriented Programming (Short paper)

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

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

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

Guido Salvaneschi
Technische Universität Darmstadt
salvaneschi@cs.tu-darmstadt.de

Abstract

Data replication is essential in scenarios like geo-distributed datacenters and edge computing, but it poses a challenge for data consistency. Developers either adopt Strong consistency at the detriment of performance or they embrace Weak consistency and face a higher programming complexity.

We argue that language abstractions should support associating the level of 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 be mixed only in a correct way.

1 Introduction

In scenarios like edge computing, or geo-distributed datacenters, data replication is critical to achieve scalability, low access 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 or performance. For example, Strong consistency models, such as *Sequential Consistency*, do not allow concurrent modifications and require blocking coordination between replicas. 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. Developers are keen towards Weak consistency to boost availability, but Strong consistency is a better choice when the application correctness is at risk. To make things worse, applications often require different consistency models in the same software, 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 of replicated data to infer the guarantees, (b) ensure that data with different consistency models is mixed correctly, and (c) reason about concurrency when mixing consistency models.

```
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.

We propose ConSysT, a language for distributed programming featuring fine-grained, data-centric specification of consistency levels. 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 to ease the adaption by developers.

2 Overview

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

Distribution We consider a system model where 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 a replicated object. Distributed *operations* are performed by calling methods on replicated objects.

Figure 1 shows how to replicate a counter in ConSysT. Therefore, we first start a transaction on the replicated store (Line 5), then we create the replicated Counter object by instantiating the class with `replicate` (Line 7). This returns a reference `Ref` to the replicated object. When creating the replicated object, the developer names the object, `"id"` in the example, so that it can be referred to by other processes. Developers use references `Ref` to perform operations on replicated objects. Operations are prefixed with `ref` to make remote accesses explicit. For example, a `Counter` object has the operation `inc`, which can be performed by calling the respective method on the reference (Line 8).

$$
\begin{array}{rcl}
\textit{Expr} \ni e & ::= & x \mid \mathsf{Ref@}\ell(\rho) \\
\textit{Computation} \ni c & ::= & \mathsf{skip} \mid \mathsf{let}\ x \coloneqq \mathsf{tx}(t)\ \mathsf{in}\ c \mid \mathsf{return}\ e \mid \dots \\
\textit{Transaction} \ni t & ::= & \mathsf{let}\ x \coloneqq \mathsf{replicate}(\rho,\ \ell,\ C,\ \overline{e})\ \mathsf{in}\ t \mid t\,;t \\
 & & \mid \mathsf{let}\ x \coloneqq e.m(e)\ \mathsf{in}\ t \mid \mathsf{return}\ e \mid \dots \\
\textit{Program} \ni P & ::= & c_1,\ \dots,\ c_n \\
D & ::= & \mathsf{class}\ \mathsf{C}_1\ \mathsf{extends}\ \mathsf{C}_2\ \{\overline{F};\ \overline{M}\} \\
M & ::= & \tau_2\ \mathsf{m}(\tau_1\ x)\{\ t\ \} \\
\textit{ConsLevel} \ni \ell & ::= & \mathsf{Sequential} \mid \dots \\
\textit{ConsType} \ni \tau & ::= & \mathsf{C@}\ell
\end{array}
$$

Figure 2. Syntax of the core calculus.

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

Figure 3. Example transition rule.

Consistency How the operation is executed depends on the *consistency level* of the replicated object. The consistency level describes the consistency model – such as sequential, or causal consistency – which is used to keep replicated objects consistent with other replicas. In ConSysT, the developer has fine-grained control over the consistency of replicated data, as every replicated object defines its own consistency level. In the example, we create a replicated `Counter` using the consistency level `Sequential` (Line 7).

Operations performed on a `Sequential` replicated object are propagated using the sequential consistency model. All fields of the replicated object use the same consistency level as the object itself. Yet, a replicated object can also contain a reference to other replicated objects that define their own consistency levels. Thus, ConSysT enables mixing replicated data in two ways: (a) the same operation can contain replicated objects with different consistency levels, or (b) replicated object can be nested, i.e., a replicated object can have a field that is a reference to another replicated object.

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

For the operational semantics, we have 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 a local 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 operations waiting for concurrent transactions. Figure 3 exemplifies the rule for replicate. The rule evaluates expressions $\overline{v}$ and creates a new object o which is then stored in the local store δ. The changes in the local store are written to the global store Δ by the rule for transactions (omitted for brevity).

Correctness When consistency levels are mixed, it is important that the language supports developers in tracking and correctly mixing objects with different consistency levels. In ConSysT, we formalize an information-flow type system for consistency types that uses that subtyping relation to ensure that Strong objects do not depend on Weak objects. Such a flow can degrade consistency guarantees [6]. The type system is parametric in the concrete consistency levels: Consistency 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 then 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 that have 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 consistency levels.

3 Related Work

Mixing consistency has been tackled in several works. Holt et al. [2] use consistency types and the notion that type safety implies consistency safety. However, their type system does not consider control dependencies – hence we adopt an *information-flow type system* in ConSysT. RedBlue Consistency [5] defines two consistency levels to label operations red, i.e. Strong, or blue, i.e. Weak, consistent. Operations have to be labeled red, when they violate application invariants if executed concurrently. MixT [6] is a DSL for transactions over multiple datastores with different consistency levels and different semantics for operations. A type system enforces correct mixing of such 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 directly on *data*, as in ConSysT, another approach is to annotate *operations*. Quelea [8] allows developers to define invariants on functions which result in a consistency guarantee on the ordering relations of operations. Gallifrey [7] is a language for replicated objects. In Gallifrey, developers define restrictions on operations in the form of conditions. Instead, in ConSysT, the data-based approach alleviates the definition of consistency as developers do not need to write special invariants.

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).