ConSysT: Tunable, Safe Consistency meets Object-Oriented Programming

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

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

Guido Salvaneschi
Technische Universität Darmstadt
Darmstadt, Germany
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.

Keywords distributed systems, replication, consistency, type systems, Java

ACM Reference Format:
Mirko Köhler, Nafise Eskandani, Alessandro Margara, and Guido Salvaneschi. 2020. ConSysT: Tunable, Safe Consistency meets Object-Oriented Programming. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In scenarios like local networks, edge computing, or geo-distributed datacenters, data replication is critical to achieve scalability, low access latency and fault tolerance. However, keeping replicas consistent in the presence of data modifications poses a challenge to the underlying system and developers. Recently, many Weak and Strong 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 the availability and performance as immediate and blocking coordinated between replicas is needed. On the other hand, Weak consistency models defer coordination between replicas. This solution increases availability and performance, but it 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 and performance, but Strong consistency is a better choice when the application correctness is at risk. To make things worse, applications often require data with different consistency models in the same software, e.g., payment requires Strong consistency, whereas Weak consistency suffices for instant messaging.

To this end, developers are required to program with multiple consistency models in the same application. Yet, this solution 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 within a single transaction. an object-oriented language for distributed applications where consistency can be specified at the granularity of *replicated objects*.

We propose ConSysT[1], a language and middleware 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 transactions under different models and it 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.

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

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. `Operations` are performed by calling methods on replicated objects.

In the following, we use a replicated counter as introductory example. Replicated objects are created by instantiating classes with the method `replicate`. This method returns a reference `Ref` to a replicated object. For example, we can instantiate a replicated counter just by replicating the `Counter` class (Line 6).

```
1 class Counter {
2     int i;
3     Counter(int i) { this.i = i; }
4     void inc() { i = i + 1; } }
5 Ref<Counter> counter =
6   replicate("id", Counter.class, 42);
```

When creating the replicated object, the developer names the object, `"id"` in the example, so that it can be referred to by other processes.

```
Ref<Counter> counter = lookup("id");
```

Developers use references 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.

```
1 counter.ref().inc();
```

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. For example, we can create a replicated `Counter` using the consistency level `Sequential`.

```
1 Ref<Counter> counter =
2   replicate("id", Sequential, Counter.class, 42);
```

Operations performed on a `Sequential` replicated object are propagated using the sequential consistency model. Operations can call other operations during their execution. These nested operations have transactional isolation guarantees, thus, equating operations to transactions.

All fields of the replicated object use the same consistency level as the object itself. Yet, a replicated object can also contain a references 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.

Correctness When consistency levels are mixed, it is important that the language supports developers in tracking and correctly mixing objects with different consistency levels. For that, ConSysT adopts *consistency types* 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`.

```
1 Ref<@Sequential Counter> counter =
2   replicate("id", Sequential, Counter.class, 42);
```

The consistency levels are ordered in a lattice [1, 7] that defines thes subtyping relation for consistency types. ConSysT employs a information-flow type-system for consistency types that uses that subtyping relation to ensure that Strong consistent data does not depend on Weak consistent data. Such a flow can degrade consistency guarantees [4].

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 [3] 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 [4] 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 [6] allows developers to define invariants on functions which result in a consistency guarantee on the ordering relations of operations. Gallifrey [5] is a language for replicated objects. In Gallifrey, developers define restrictions on operations in the form of conditions. The advantage of operation-based approaches is that consistency levels are specified even more fine-grained: operations with different consistency levels can be used on the same data. Instead, in ConSysT, the data-based approach alleviates the definition of consistency as developers do not need to write special invariants. We plan to leverage the disadvantages of the data-based approach by adding a declassification operation that allows to use a replicated object under different consistency levels. This combines the advantages of both approaches.

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] 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.

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

[5] 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.

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

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