Consistency Types for Safe and Efficient Distributed Programming

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

Guido Salvaneschi
TU Darmstadt
salvaneschi@cs.tu-darmstadt.de

Abstract

Consistency is a long standing problem in distributed systems. Low consistency levels are considered a necessity for scalability. High consistency is required for critical tasks such as payment and identification. Modern (geo-)distributed systems rely on the data propagation mechanisms and consistency guarantees of the distributed data store they build upon, which makes the implementation of a system that mixes different levels of consistency complex and error prone. In this paper we present preliminary work on ConSysT, a programming language that supports heterogeneous consistency specifications at the type level. In ConSysT, developers assign consistency levels directly to the data and the type system ensures the correct behavior of the application even with computations that mix data at multiple consistency levels. Our vision is that the ConSysT runtime automatically determines the most efficient mechanism to achieve the desired level of consistency among those offered by the underlying data store.

ACM Reference format:
Alessandro Margara and Guido Salvaneschi. 2017. Consistency Types for Safe and Efficient Distributed Programming. In *Proceedings of FTfJP'17, Barcelona, Spain, June 18-23, 2017,* 2 pages.
DOI: 10.1145/3103111.3104044

1 Introduction

Brewer's CAP theorem states the impossibility to simultaneously achieve always-on experience – *availability* – and to read the latest written version of a distributed data store – *linearizable consistency* – in the presence of partial failures – *partitions* [4]. The ubiquity and ever-growing size of distributed systems makes partition tolerance indispensable. Thus the choice remains between the desired level of consistency and availability (or low latency). Higher levels of consistency are not negotiable in monetary transactions, mission critical and security applications. Lower levels of consistency – thus, high availability – has been effectively leveraged in several systems such as social media applications, which may accept (temporary) inconsistency in the content they provide to clients. Eventual consistency has become a de-facto standard in this class of systems [11].

However in practise, the classification of systems into those that require high consistency and those for which low consistency is acceptable is not sharp. In fact, many systems mix functionalities that require high consistency such as payment, with functionalities that are perfectly compatible with lower levels of consistency, such as advertising. To meet these requirements, some work propose mixing data at multiple levels of consistency to provide low latency when possible and high consistency when needed [9].

Programming on top of a weakly consistent data store is arguably difficult, since the developers need to identify possible anomalies that arise from data inconsistency and compensate their effects [2]. Multiple levels of consistency further require reasoning on the possible interactions of data elements with different levels.

We claim that a programming language that is aware of data consistency can support and ease the development of distributed applications. Moving from this premise, this paper introduces ConSysT, a programming language that supports defining computations on data at different consistency levels, thus enforcing high consistency where needed without sacrificing the performance gain of using low level consistency when it is allowed to do so. We sketch a type system that is aware of the consistency levels and ensures that data at different levels are not mixed in a way that breaks the expected semantics of the application.

2 The ConSysT Programming Language

ConSysT is a programming language for distributed systems. Values of type `Shared[T]` are replicated on multiple peers with each peer holding a local copy to increase fault tolerance and availability. When the variable is written, the new value is propagated to the other replicas. The code below shows – in a syntax similar to Scala – three peers sharing the variables `a` and `b`:

```
val a: Shared[Int] = 1     // Peer P1     // Peer P2     // Peer P3
val b: Shared[Int] = 8     a = 5          b = a++        a + b
                           ...            ...            ...
```

Depending on the specific propagation mechanisms adopted, and on the level of consistency they provide, the operation `a + b` on P3 can lead to different results. (1) P3 computes the value `9 = 1 + 8` if it performs the `+` operation before observing the updates on `a` and `b` performed in the other two peers. (2) P3 computes the value `7 = 1 + 6` if P2 receives the update `a = 5` before computing `b = a++`, and P3 observes the update of `b` (`b = 6`) from P2 but not the update `a = 5` from P1. The latter case exemplifies a violation of *causal consistency* because P3 sees the new `a` but not the value of `a` that *caused* it. In ConSysT, this problem can be addressed by defining the values `a` and `b` causally consistent (CC). This definition guarantees that if P3 sees the new value of `b`, it also sees the value of `a` that caused it, ruling out case (2).

```
val a: CC[Int] = 1
val b: CC[Int] = 8
```

Various levels of consistency – and corresponding update propagation mechanisms – exist that provide different balances between the guarantees offered to the developers and the performance cost for ensuring these guarantees.

In ConSysT, the developers can specify consistency at the granularity of individual values, to enforce a high level of consistency only in cases that strictly require so. For instance, the following code snippet defines a variable users as eventually consistent (EC) and a variable dashboardText as causally consistent.

```
val users: EC[Int] = ...
val dashboardText: CC[String] = ...
```

An obvious problem is how different consistency levels interact in the same application, for instance, in the case a function takes in input both dashboardText and users to produce another value. Also, another problem is when a program tries to assign e.g., a CC[Int] value to a EC[Int] value. The ConSysT type system disciplines the interactions between values at different levels of consistency by ensuring that values with low consistency levels do not flow into computations that demand for higher consistency levels.

Core ConSysT We introduce a core calculus for ConSysT. The syntax includes function applications and abstractions, variables, consistency types and base types:

t	$::= t\ t \mid \lambda x : T.\ t \mid v$	Terms
v	$::= \lambda x : T.\ t \mid \ldots$	Values
H	$::= EC[T] \mid SC[T] \mid CC[T] \mid FIFO[T] \mid Shared[T] \mid T$	Types
T	$::= Int \mid String \mid \ldots$	Base Types

The semantics is standard for call by value lambda calculus. For ease of explanation, we hereby consider five levels of consistency: eventual EC[T], FIFO FIFO[T], causal CC[T], sequential SC[T], and no consistency guaranteed by the language Shared[T]. Other consistency levels can be defined to accommodate the needs of applications and the guarantees provided by specific data store implementations. Besides the standard ones, the typing rules for ConSysT are:

$$\frac{\Gamma \vdash t : SC[T]}{\Gamma \vdash t : T} \quad \text{(T-COERCE)}$$

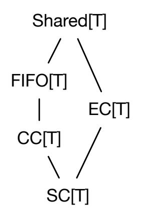
$$\frac{}{SC[T] <: CC[T]} \quad \text{(T-SUB1)}$$

$$\frac{}{CC[T] <: FIFO[T]} \quad \text{(T-SUB2)}$$

$$\frac{}{FIFO[T] <: Shared[T]} \quad \text{(T-SUB3)}$$

$$\frac{}{SC[T] <: EC[T]} \quad \text{(T-SUB4)}$$

$$\frac{}{EC[T] <: Shared[T]} \quad \text{(T-SUB5)}$$



With T-COERCE sequentially consistent values can be treated as a local value. The other rules define subtyping relations that prevent assigning lower consistency levels to higher consistency levels.

Outlook We are currently working on a Scala embedding of ConSysT. Open questions in the design and implementation of ConSysT include:

- Defining complex data types with fields at heterogeneous consistency levels.
- Devising a suitable backend for the implementation. We are currently considering Apache Cassandra as a possible candidate.
- Integrating transactional updates that involve multiple operations and are possibly executed with various levels of isolation.
- Considering restrictions to the operations allowed on certain data types – for example, commutative operations – to guarantee a higher consistency level even with low synchronization overhead.

3 Related Work

Databases and distributed systems Internet-scale distributed systems highlight the tension between performance – availability – and consistency. Motivated by the observation that eventual consistency often behaves as stronger consistency levels since anomalies seldom occur, various NoSQL data stores favor performance over correctness and adopt some form of eventual consistency [11] or causal consistency [3]. Under certain conditions, values are guaranteed to converge in a self-stabilizing manner, despite any failure. This kind of datatypes are known as conflict-free replicated datatypes (CFRDs) [10]. RedBlue consistency [9] exploits the efficiency of eventual consistency to propagate (operations on) CFRDs, and propagates other datatypes with protocols that ensure stronger consistency. SIEVE [8] automatically recognizes CFRDs and selects the least expensive update propagation mechanism. An orthogonal line or research studies protocols to ensure stronger levels of consistency than eventual consistency – for example, bolt-on causal consistency [3] and high-available transactions (HAT) [1] – without incurring in high synchronization costs.

Programming Languages and Type Systems Cloud types [5] are a way to ensure eventual consistency in distributed systems. AJ [6] is a programming language that takes a data centric approach to concurrency. In AJ, object fields are grouped into sets that must be updated atomically. Code fragments, referred to as units of work are associated to atomic sets, and the compiler automatically adds synchronization operations to preserve the consistency of the atomic set. Holt et al. [7] introduce consistency types but with an inheritance hierarchy inverted compared to ours, letting values with low consistency flowing into values with high consistency. In addition to expressing consistency, their types can embed probabilistic guarantees on the latency of propagation or on the maximum difference between the values stored at each replica. DCCT is a language that mixes multiple consistency levels [12]. In contrast to ConSysT– a general purpose language with consistency types – DCCT is a data-oriented language where *actions* – for example, queries – access a distributed storage, and annotations define consistency of values.

References

[1] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. HAT, Not CAP: Towards Highly Available Transactions *(HotOS '13)*. USENIX Association.
[2] Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Commun. ACM* 56, 5 (2013), 55–63.
[3] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency *(SIGMOD '13)*. ACM.
[4] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. (2012).
[5] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency *(ECOOP'12)*. Springer-Verlag.
[6] 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).
[7] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types *(SoCC '16)*. ACM.
[8] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems *(USENIX ATC'14)*. USENIX Association, 281–292.
[9] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary *(OSDI '12)*. USENIX Association.
[10] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types *(SSS'11)*. Springer.
[11] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (2009), 40–44.
[12] Nosheen Zaza and Nathaniel Nystrom. 2016. Data-centric Consistency Policies: A Programming Model for Distributed Applications with Tunable Consistency *(PMLDC '16)*. ACM.