High-level Programming for Replicated Data Types

Nicholas V. Lewchenko
University of Colorado Boulder

ABSTRACT

Replicated data types (RDTs) are a useful program model for services that require stable availability in the face of geographic separation and high traffic. Availability in such a system, however, comes at the cost of difficult correctness reasoning. I am working on a domain-specific language for RDT applications which enables automated verification of program safety properties and allows a developer to maximize replica parallelism (and thus availability) without needing to reason about event orderings or inter-replica communications.

1 BACKGROUND

A replicated data store design consists of two parts: an RDT which represents the store's data and primitive operations, and a communication network between the store's replicas through which they synchronize their activity. The choice of network impacts the system's correctness and efficiency. A simple network model such as *causal broadcast*, which propagates all updates that enter replicas and guarantees only a *causal ordering*, allows highly available replicas and low overhead, but makes logical errors in the system (such as double-spending, in the case of a bank system) impossible to prevent. More complex systems that prevent such errors must sacrifice either availability (replicas must coordinate before performing updates) or simplicity/overhead (replicas must arbitrate conflicting updates and roll one back after the fact).

Mixed consistency network models have been proposed which selectively force coordination between replicas for certain updates designated by the developer [2, 3], behaving as a simple causal broadcast system otherwise. Such models gain efficiency while maintaining correctness, but they are difficult to use. The virtue of replicated data stores—that they are written and behave essentially as ordinary sequential systems—is lost on the developer who must now reason about which orderings of events to allow. Worse is that these low-level configurations must be reevaluated for each new application's safety requirements, even if they use common underlying RDTs.

2 APPROACH

I am developing a programming model for replicated applications using mixed-consistency network mechanisms which does not require the developer to think about concurrent executions and which enables automated verification of the program properties which the mechanisms are intended to preserve. In this model, RDT definitions are extended with a set of *query predicates* which can be applied to the store value. Updates which depend on the value of the global store (and may behave inappropriately if their replica is not up to date) must explicitly request that information using a particular query predicate, thus declaring precisely the level of accuracy they require. Developers interact with this new aspect of the RDT rather than directly with the network's consistency mechanisms.

```
type CState = Int
data CEffect = Add Nat | Sub Nat
data CQuery = Eq | Lte | Gte | Any

type Counter = RDT CState CEffect CQuery
```

Figure 1: A definition for the `Counter` RDT

```
type BankOp = Operation Counter

deposit :: Nat -> BankOp
deposit n = emit (Add n)

withdraw :: Nat -> BankOp
withdraw n = do accountValue <- query Lte
                if accountValue >= n
                   then emit (Sub n)
                   else noEffect
```

Figure 2: Operations defined over the `Counter` for a replicated bank account application

As an example, consider a replicated bank account using a *counter* (an integer value supporting add and subtract) as its datatype. In a purely sequential sense, a safe withdrawal operation must check that the account has enough money before it takes effect, to prevent overdrafting. In the example code from Figure 2, we see that `withdraw` accesses the account value using the "`Lte`" query, meaning that it requests a value that is *less than or equal to* the true account value. We can verify that this implementation behaves safely using standard, automatable sequential logic [1]. At this level of the design, we only need to verify the interaction between our invariant (account always ≥ 0) and the query interface provided by the `Counter` (enumerated as `CQuery` in Figure 1). Verifying the relationship between the `Counter`'s interface and the underlying network is performed independently.

For replicated data types which are SMT-representable (such as the `Counter`), the network configurations necessary to execute each supported query, and thus satisfy our earlier assumptions, can be generated automatically. These configurations are not specific to any one application (such as the bank account); they can be reused wherever the RDT is

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

needed. In our example, the network will require replicas to coordinate when executing `withdraw`s (because the `Sub n` emitted from one `withdraw` could invalidate the result of another's `Lte` query), but `deposits` can execute immediately because they make no query.

3 IMPLEMENTATION

I am implementing this programming technique as an embedded monadic DSL in Haskell, and intend to use particular language features available in this environment for increased expressivity and automation. As demonstrated in the example code from Figures 1 and 2, I am studying the representation of store queries and effect emissions as monadic functions. This representation would require the support of *nested* queries, which may not be meaningfully implementable with current mixed-consistency network mechanisms. I hope that in this case, the language development can inspire the complex network protocols needed to realize it.

My adviser and I have developed a formal theory for deciding minimum network consistency requirements on queries and using these results for operational reasoning, automating the process using an SMT solver. I would like to integrate this method of verification directly into the code-writing process by using extensions to Haskell's type system, namely *refinement types* as provided by LiquidHaskell [4]. The primary contribution to which I am eventually aiming this work is a library of RDTs with clear, verified query interfaces, which developers can employ efficiently in unique distributed applications without ever needing to consider the network mechanisms that keep them running safely.

REFERENCES

[1] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

[2] 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

[3] 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 '15)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

[4] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161