1 Title

A Continuous Consistency Library for Distributed Applications

2 Summary

Distributed systems incur significant overhead when synchronizing replica states. Many classes of distributed systems are able to allow replica states to synchronize conditionally, reducing this overhead but resulting in arbitrary state divergence and losing several invariant properties in the process [15]. Yu et al. posit that a system with a non-zero, known, finite *bound* to its inconsistency preserves many valuable properties. A *continuously consistent* distributed system quantifies and repairs inconsistency to preserve its bound [15]. Several promising models for continuous consistency have been defined and explored, but have yet to be broadly adopted. Some implementations do exist, but few outside of distributed databases. This project aims to develop a library to provide an accessible and desirable means of lifting an application to continuous consistency. We describe an approach which provides an API sufficiently expressive for all target applications while minimizing correctness pitfalls.

3 Description of Research

3.1 Introduction

Distributed systems must maintain consistency to collaborate on tasks. This maintenance takes many forms. For example, web services host replicated servers to partition the user space, and compute nodes operate on sub-problems to parallelize work. Maintaining consistency requires nodes to communicate. *Strong consistency* (where states may never diverge) incurs substantial overhead, as a modification to any state must be propagated and reflected everywhere before another modification anywhere may proceed. Conversely, other applications may opt for *optimistic consistency*, allowing nodes to be independently productive and only restore consistency at the last possible moment, incurring minimal overhead [3, 15]. *Bounded inconsistency* generalizes the consistency spectrum, defining the *distance* between nodes in a system in terms of some *metric* which is guaranteed never to be exceeded. Here, strong and optimistic consistency have the most extreme bounds of zero and infinity respectively [15]. The act of maintaining a known, finite consistency bound is called *continuous consistency* as the system must continuously quantify and manage this distance at runtime. This consistency model provides properties **P1-3**:

P1 Metrics may differentiate the consistency restraints of different data [15]. For example, an application strictly preserves a key property, but relaxes others to reduce the overhead overall [4].

P2 Bounds may *change* to optimize the trade-off between consistency and *productivity* [1, 14, 15]. For example, a game server may reduce client update frequency to avoid being overburdened.

P3 Known bounds facilitate statistic analysis of system properties that are tied to consistency [12, 15]. For example, a system predicts the minimum freshness of retrieved data.

These properties are not unique to continuous consistency. **P1-2** is shared with optimistic consistency, as the models have *non-zero* bounds in common. **P3** is shared with strong consistency, as the models have *known* bounds in common. This set of properties makes continuous consistency particularly suitable to the following classes of distributed systems, collectively referred to as our 'target domain':

- **Distributed databases**, **replicated web servers**, and **online-game servers** can leverage **P1** to provide additional availability and performance without sacrificing consistency where it matters, particularly in cases with predictable workloads [4]. **P2** allows replicas to scale quality of service such that system resources are neither over- nor underutilized [1]. Users may be assigned to replicas in response to the replicas' bounds, enabling further optimization of resource utilization [15].
- **Big data analysis** common to **business analytics**, **bioinformatics**, **machine learning** and **information retrieval** prohibits significant synchronization overhead, owing to their reliance on performance [14, 15]. **P1** facilitates trading excessive consistency for the needed performance, but **P3** preserves the ability to reason about of the quality of the data retrieved [15].

Yu et al. define a model for continuous consistency that promises both *generality* and *practicality* such that the model is both widely applicable, and results in intuitive and efficient implementations [15]. The model is built with the *conit* ('consistency unit') as the fundamental primitive, with each conit representing a consistency dependency between data objects in the replicated state. Concretely, the application developer 'protects' the necessary read and write operations with `dependOnConit()` and `affectsConit()` operations, which work together at runtime to emerge as conditional synchronization. Conits represent arbitrary data dependencies in terms of three intuitive metrics: to what extent may write events (1) be reordered (2) disagree on a value (3) be stale. An application's defined conits double as the inconsistency *bound* which characterizes continuous consistency models. Yu et al. argue that this model is sufficiently *general* to express all data dependencies necessary to the target domain. On the other hand, the simple three-metric approach results in a simple and intuitive API surface which maps naturally to the expected implementations. In addition, conits facilitate synchronization of arbitrary granularity, right down to per-write or even pair-wise between replicas.

The conit model and its promising properties have been investigated academically, exploring its applicability in real systems, as well as the benefits [1, 12, 13]. It has also influenced systems such as SHARP [5], TACT [14] and Colyseus [4], most of which are intended for distributed databases. However, the vast majority of applications within our target domains do not make use of bounded consistency (particularly online games [4] and web services [12]). Generally, applications are over-conservative, opting for a consistency model that excessively inhibits productivity. Developers may overlook opportunities to relax consistency in their applications. This is supported by Bermbach et al. in their investigation of the exaggerated inconsistency measured using prevalent data-centric models compared to the inconsistency actually observable by end-users [3]. Ultimately, developers are either unaware of the option, or rather consider the integration of continuous consistency models not to be worthwhile. We believe such models are valuable, and with the right approach, the barriers to their wider adoption can be overcome.

3.2 Our Approach

Based on findings detailed in section 3.1, we believe more developers of distributed applications within our target domains would make use of *continuous consistency* if the cost-to-benefit ratio was more favourable. This work aims to produce a *package* that achieves this by being composed of:

1. A **Guide** for developers to identify opportunities to design their systems such that they make effective use of continuous consistency. This clarifies the *benefits* of using continuous consistency.

2. A **library** written in a general-purpose programming language with an API which closely mirrors the conit model. This library should provide all the functionality necessary for lifting a strongly consistent application in the target domain to a continuously consistent one while abstracting away error-prone consistency minutiae. This decreases the *cost* of using continuous consistency.

An ideal model must be general enough to apply to all applications in its target domain, yet practical enough that the resulting implementation is uncluttered and efficient; these requirements typically conflict. The conit model of Yu et al. promises a balance of these properties suitable for our purposes [15]. We believe the model has not seen wide adoption owing to its focus on applicability for distributed databases, reflected in the authors' choice of API and examples, while the underlying theory is much more widely applicable [15]. Our hope is to facilitate the wider adoption of continuous consistency by making it accessible to the entirety of our target domain.

Our approach relies on facilitating the application of the model in a way that is 'natural' to the particular case, avoiding an intermediate database-like representation. Furthermore, we do not wish to build a run-time system, as many use-cases are performance-sensitive. However, requiring the developer to implement the conit logic 'from scratch' unassisted is error-prone, requiring reasoning about distributed state: a task notoriously difficult for humans [8]. Thus, the library itself should enforce correctness.

Many modern programming languages implement *trait* primitives, which allow the expression of required behavior for an implementor type [11]. If equipped with *generics* and *monomorphization*, the language facilitates 'generic' code expressing the same operations on different data types without redundancy, while still being ultimately compiled down to efficient, specialized binary code for each type ('monomorphized'). These features facilitate a library built out of inter-linked trait constraints [7, 10]. A compiler is able to assist the programmer in enforcing these constraints on the application developer's own types [7]. The developer is then only responsible for filling in bodies of trait functions to map conit logic to their specific domain; for example, a logical synchronization event may map to a message being sent. The library's various components would make extensive use of the *proof of work* pattern[1], further guiding the developer to invoke necessary functions in the correct order. This approach has the developer implement the abstract conit model themselves, with the library's type definitions informing the compiler how to assist in checking that the implementation is correct. This helps to avoid correctness pitfalls and

[1]To ensure function `foo` precedes function `bar`, `foo` returns a new-type wrapper of the empty structure as a *token* which `bar` requires as input. This disappears at compile-time, but imposes an ordering on these functions which is hard to overlook.

achieves a domain-specific implementation without any sort of runtime system. The Rust programming language is suited for this purpose, as it supports the necessary features, offers high performance and a memory-safe static type system. It also inter-operates cheaply with C and C++, freeing developers from having to build their application entirely in Rust [2, 6, 9]. A concrete example of *generic traits* implemented in Rust is given in listing 1.

```
trait Bounded {
  fn test_bound(&self) -> f32;
}

trait Conit {
  fn synchronize(&mut self);
  fn affect<B: Bounded>(&mut B);
}
```

Listing 1: `Conit` exemplifies a Rust trait, with function `affect` generic over type `B` constrained by `Bounded`. The library may offer default function implementations in terms of the type constraints where possible, leaving the domain-specifics for the application developer to fill in. Ultimately, an application binary is *monomorphized*, appearing as if hand-written without traits or generics.

The conit model of Yu et al. serves as a promising starting point for the library's functionality. Alternative continuous consistency models exist that offer different representations [13]. The best choice for our target domains may be one of these alternatives, or even a novel variant or hybrid of multiple models.

3.3 Planning

This section details how project tasks are distributed over the 4 years for one PhD student. 'Start' and 'End' mark intervals from the project's beginning, measured in years.

Start	End	Task
0.0	1.0	A prototype **library** is created to implement the model as specified by Yu et al. [15] similar to TACT and also targeted at distributed databases. The phase ends with a functional test implementation of a distributed database using the library.
1.0	1.5	The set of test implementations is determined to represent the entire target domain defined in §3.1. The **library** is iterated upon toward supporting these implementations and domains in general. The phase ends with the preliminary design of the library API.
1.5	2.5	The **library** is developed concurrently with all test implementations. Necessary changes to test implementations are documented for the first draft of the **guide**. The phase ends when all implementations are functionally complete, but not yet refined.
2.5	4.0	The **library** API is optimized to facilitate the developer experience, and its implementation is optimized for application efficiency. Changes are reflected by iteration on the **guide**. The guide is extended with experimental results to measure the performance implications of using the library. The phase ends when the library's implementation is finalized, and the contents of the guide are finalized and edited.

4 Literature

[1] Raihan Al-Ekram, Ric Holt, and Chris Hobbs. Applying a tradeoff model (tom) to tact. In *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*, pages 351–355. IEEE, 2007.

[2] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. System programming in rust: Beyond safety. *ACM SIGOPS Operating Systems Review*, 51(1):94–99, 2017.

[3] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? an evaluation of amazon S3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, page 1. ACM, 2011.

[4] Ashwin R Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A distributed architecture for online multiplayer games. In *NSDI*, volume 6, pages 12–12, 2006.

[5] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. Sharp: An architecture for secure resource peering. *ACM SIGOPS Operating Systems Review*, 37(5):133–148, 2003.

[6] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66, 2017.

[7] Nicholas D Matsakis and Felix S Klock II. The Rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[8] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. Use of formal methods at Amazon Web Services. *See http://research. microsoft. com/en-us/um/people/lamport/tla/formal-methods-amazon. pdf*, 2014.

[9] Eric Reed. Patina: A formalization of the rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.

[10] Dimitri Sabadie. On dealing with owning and borrowing in public interfaces, Nov 2018.

[11] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.

[12] Swaminathan Sivasubramanian, Michal Szymaniak, Guillaume Pierre, and Maarten van Steen. Replication for web hosting systems. *ACM Computing Surveys (CSUR)*, 36(3):291–334, 2004.

[13] Luís Veiga, André Negrão, Nuno Santos, and Paulo Ferreira. Unifying divergence bounding and locality awareness in replicated systems with vector-field consistency. *Journal of Internet Services and Applications*, 1(2):95–115, 2010.

[14] Kevin Walsh, Amin Vahdat, and Jun Yang. Enabling wide-area replication of database services with continuous consistency. *Unpublished Manuscript*.

[15] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20(3):239–282, 2002.