# Data Consistency and Convergence

Department of Computer Science, University of California, Riverside

**Abstract**

*The key paper that we will be reviewing is "Putting consistency back into Eventual consistency". This paper along with the related supporting papers suggests new approaches to solve the data consistency and convergence problems in an operational distributed systems model. By reviewing these papers we try to prove that it is not necessary to give up data integrity and convergence for achieving low-latency and high throughput.*

## 1. Putting consistency back into Eventual consistency

### 1.1 Introduction

When designers of complex distributed software applications have to choose between replication protocols, they usually have to make a choice between low-latency, eventually consistent operations or strong consistency but lower performance. This paper proposes an alternative consistency model called Explicit Consistency. It strengthens eventual consistency with guarantees to preserve specific invariants. Explicit consistency identifies which operations would be unsafe under concurrent executions through static analysis. It also allows programmers to choose between either invariant violation-avoidance or invariant repair techniques. The paper also presents a middleware called Indigo that provides Eventual consistency on top of causal consistency data store. Indigo guarantees strong application invariants while providing similar latency to an eventual consistent application.

### 1.2 Contributions:

- Explicit consistency: new consistency model for data convergence focused on application semantics and not on order of operations.
- A methodology to obtain a reservation system for enforcing explicit consistency, based on the set of defined invariants associated with the application.
- Indigo, a middleware system implementing explicit consistency on top of causal consistency geo-replicated data store.

### 1.3 Explicit Consistency

The system is allowed to reorder execution of operations at different replicas, provided the specified invariant is maintained. It is also possible to avoid cross-datacenter coordination, even for crucial operations that could possibly violate invariants. Explicit consistency can be summarized as follows:

- The information to detect the set of operations that may lead to invariant violation when executed concurrently is provided by the developer. This combined with the invariants performs statistical analysis to infer the set of operations when executed concurrently may lead to invariant violations.
- Secondly, a mechanism needs to be decided to handle the steps, one is *invariant repair* in which operations are allowed to execute concurrently and conflict resolution rules are applied to restore the invariant. The other is *violation avoidance* which restricts concurrency to avoid invariant from being violated.
- The third step is to instrument the application code to use the

conflict-repair or conflict-avoidance mechanisms.

**1.4 Evaluation:**

There are two sample applications that the explicit consistency is tested on - the first is the add counter application which models the information maintained by the system that manages the ad impressions in online applications. This information needs to be geo-replicated. The second is tournament management in a distributed multi-player game. Indigo framework is compared against three baseline approaches - causal consistency, strong consistency and RedBlue. The results show that Indigo achieves throughput and latency similar to causal consistency. The throughput of RedBlue is only slightly less than that of Indigo. The results also show that Indigo exhibits lower latency than RedBlue. The peak throughput for Indigo decreases while latency keeps constant. This is a result of an increase in the number of updated objects as the number of invariants increase which has an impact on the number of operations. Therefore, in conclusion the results show that applications have performance similar to weak consistency for most operations, while being able to enforce application invariants.

## 2. Cloud Types for Eventual Consistency

**2.1 Introduction**

The research paper proposes the use of specialized cloud data types in order to make eventual consistency more programmable. This is basically an abstraction to implementation details like caches, protocols, servers, networks, etc.

The paper focuses on introducing special data types known as the cloud types that synchronizes the data instantaneously across all devices on local as well as cloud storage. These include simple types like cloud strings, cloud integers, etc as well as structured types like cloud entities, cloud arrays, etc. These are chosen in a way that they automatically work under concurrent conditions.

The system implements revision diagrams to maintain eventual consistency. In this model, cloud stores the main revision while the devices store local revision which are synchronized periodically.

**2.2 Contributions**

- A data model that integrates support for eventually consistent data in the programming language.
- Construction of data schema from basic cloud types and development of advanced types from the simple ones.
- Formal syntax and semantics to enable support for an operational distributed system model with a programming language.

**2.3 Similarities and differences**

**2.3.1 Eventual Consistency:**

If the client is not expecting a server response, it forks a new revision for local usage after which it sends the current revision to the server. If there is a revision from the server, current local revision is merged into it. If the client is awaiting a response from the server, it does nothing.

The client repeatedly calls yield and the message is delivered eventually as opposed to the explicit consistency introduced in the key paper.

**2.3.2 Convergence:**

In the paper, the consistency model is relaxed using snapshot isolation but still it is possible that transactions might fail and couldn't commit in the presence of network partitions

while on the other hand, Indigo, in the key paper, guarantees strong application invariants and convergence owing to its reservation system.

### 2.3.4 Conclusion

The paper provides a basis for programming level abstraction of complicated parts of a system like the local persistence, the conflict resolution, the cloud service, the caching, and the synchronization while maintaining eventual consistency. It delineates the use of cloud data types which can be synchronized automatically and can also be used to create large data schemas.

## 3. Transaction chains: achieving serializability with low latency in geo-distributed storage systems

### 3.1 Introduction

The research paper introduces Lynx, a geo-distributed storage system which offers transaction chains, materialized join views, secondary indexes, and geo-replication. It utilises static analysis to determine if each hop can execute separately while preserving serializability. Also, Lynx does not replicate all data in all datacenters and so, it can have many data centers with low replication cost.

### 3.2 Contributions

- Concurrent execution preserves serializability.
- Applications achieve low-latency for chains that execute piecewise by returning control after first hop.
- Application queries are faster.
- Scalable with machines and datacenters.

### 3.3 Similarities and differences

#### 3.3.1 Transaction chains

The data is distributed amongst multiple servers. A transaction T is encoded in the chain as a sequence of hops T = [p1, p2, ….,Pk] with each hop pi executing at one server which can be at different datacenters and may repeat. It is recommended to execute a chain piecewise, i.e., hops must be treated as separate transactions and executed one after the other. Even application latency can be improved using these as it just needs to wait for the chain's first hop to complete.

#### 3.3.2 Serializability

An a priori known set of transactions is present with the web applications which enables a global static analysis of the application to determine what chains can be executed piecewise while preserving serializability.

### 3.4 Short-commings

- A programmer needs to explicitly divide the transactions into a chain such that only its first hop contains a user-initiated abort and it needs to be static, i.e. the shards it accesses at each hop must be known before the execution.
- In order to achieve low-latency, the application should proceed after the completion of the first few hops.
- Messages using external channels are not consistent.

### 3.5 Conclusion

In geo-distributed storage systems, lynx provides serializability with low-latency. The main contribution of the paper is to express transactions as chains with multiple numbers of hops. Also, features like join views, geo-replication and secondary indexes can also be implemented using chains.

## 4. Transactional storage of geo-replicated systems

The novel contribution of this paper is the introduction of Walter, a key-value store that supports transactions and replicates data across distant datacenters. The main idea behind Walter is a new property called Parallel Snapshot Isolation (PSI) which allows Walter to provide stronger guarantees. PSI provides a balance between consistency and latency. Within a site, PSI hosts transactions according to consistent snapshot and total ordering. Across sites, PSI only enforces causal ordering but not global ordering allowing the application to replicate asynchronously. To implement PSI, Walter relies on two techniques - preferred sites and counting sets. Preferred site is the site where objects can be written more efficiently. It means that these are the sites where writes to the object can be committed without checking other sites for conflicts. A counting set is like a set except each element has an integer count. Transactions with counting sets can be committed without having to check for conflicts as they are commutative data types.

**4.1 Contributions:**

Parallel Snapshot Isolation, an isolation property well-suited for geo-replicated web applications. PSI provides a strong guarantee within a site; across sites, PSI provides causal ordering and precludes write-write conflicts.

Walter, a geo-replicated transactional key-value store that provides PSI. Walter can avoid common write-write conflicts without cross-site communication using two simple techniques: preferred sites and csets.

Walter is used to build two applications and demonstrate the usefulness of its transactional guarantees.

**4.2 Evaluation**

The performance and usefulness of PSI is evaluated with two sample web applications: WaltSocial - a social networking application and ReTwis which is a clone of twitter running on redis. It is found that for WaltSocial, aggregate throughput is at 40Kops per transaction. The 99.9 percentile latency of all operations is below 50ms. Then the performance of ReTwis is then benchmarked. The performance can be seen similar to Walter. The throughput of write operation for Walter is 4712 ops/s compared to 5470 ops/s for Redis.

**4.3 Similarities and differences**

The base paper proposes a new consistency replication model called explicit consistency to enforce data integrity in eventual consistency while this paper utilizes an improved version of snapshot isolation provided popular commercial databases like Oracle called Parallel snapshot isolation by allowing different sites to have different commit orderings. It avoids the need for conflict resolution by preserving the property of snapshot isolation that committed transactions have to write-write conflicts. I believe PSI provides strong guarantees that are more suitable to web applications while explicit consistency is better suitable for static applications like bank account, student management or tournaments.

**5. Serializability for Eventual Consistency: Criterion, Analysis, and Applications**

**5.1 Introduction**

This paper enlists the details of a new serializability criterion that generalizes conflict serializability to eventually consistent semantics, while at the same time taking into account high-level operations. This model ticks of all the features, battling the challenges. It presents a model that not only looks for commutativity, but also absorption, a rich semantic filled vital combination of the two, focusing on a targeted mannered execution, rather than generalized. The executional test came from two domain apps: (1) Analysis of 33 small mobile

applications that use a weakly consistent cloud type (TouchDevelop) (2) Implementation with database benchmark TPC-C.

**5.2 Contributions**

- A well-structured serializability client criterion for weakly consistent and high-level data-based clients.
- Time bound program execution limit based polynomial algorithm to check the correctness and complexity.
- Implementation and absorption of two use cases: TouchDevelop and Raik with TPC-C. An exhaustive description that indicates unacknowledged errors that did not get recognized before, thus elevating scalability and correctness.

**5.3 Similarities and differences**

As opposed to the key paper, this paper introduces a new look at eventual consistency, focussing on serializability. The given paper leverages the concepts of commutativity and absorption to generalize conflict serializability to high level data types.

**5.4 Evaluation**

The conducted experiments depict that there has been a vital reduction in terms of serializability violations than commutativity races, thus reporting for 21 of the 33 applications for the latter and only 8 for former: leading to a 75% decline. Moreover, there was no evidence on a false alert dealing with the serializability violations. This denotes that serializability violation is the cause of the replication system returning inconsistent values to the application. If we observe the four applications mentioned in the paper, we acknowledge that these data inconsistencies had negligible and nullified effect on the overall application execution.

**References**


[1] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, et al. Putting Consistency back into Eventual Consistency. European Conference on Computer Systems (EuroSys), ACM Sigops / EuroSys, Apr 2015.

[2] Cloud Types for Eventual Consistency Sebastian Burckhardt1 , Manuel F¨ahndrich1 , Daan Leijen1 , and Benjamin P. Wood2 1 Microsoft Research 2 University of Washington

[3] Transaction chains: achieving serializability with low latency in geo-distributed storage systems Yang Zhang* , Russell Power* , Siyuan Zhou* , Yair Sovran* , Marcos K. Aguilera‡ , Jinyang Li* *New York University ‡Microsoft Research Silicon Valley 3 October 2013 Minor revision 29 October 2013

[4] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP ’11, pages 385–400, New York, NY, USA, 2011.

[5] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, pages 276–291, New York, NY, USA, 2013. ACM.