# Having Your Cake and Eating it Too: Combining Strong and Eventual Consistency

Paweł T. Wojciechowski
Institute of Computing Science
Poznań University of Technology
60-965 Poznań, Poland
pawel.t.wojciechowski@cs.put.edu.pl

Konrad Siek
Institute of Computing Science
Poznań University of Technology
60-965 Poznań, Poland
konrad.siek@cs.put.edu.pl

## Abstract

Given the limitations imposed on distributed systems that are necessary to maintain strong consistency guarantees there is a growing interest in relaxed consistency models. Such models are often sufficient for particular applications, but allow more freedom to improve scalability and availability. Eventual consistency is a particularly useful approach, where the correct state spreads throughout the system over time, so that at any point any element of the system may be inconsistent, but all elements will eventually converge upon a consistent state. On the other hand relaxing properties may be unacceptable in the general case: a slightly stale shopping cart is one thing, but inconsistent payment processing is quite another.

In this paper we try to balance strong and eventual consistency by proposing a general-purpose pessimistic distributed transactional memory that allows eventually consistent transactions to run alongside consistent ones. While the former maintain read-isolation (i.e., read from a consistent snapshot), they do not interfere with the latter's safety properties. The relaxed-consistency transactions are later followed by their consistent counterpart so that the user view and global state eventually agree. Our contribution is to show that we can significantly relax synchronization (to the point of eliminating it completely from eventually consistent transactions) while retaining useful properties, but without imposing additional constraints about system architecture or data operations, common to other relaxed consistency approaches. All this, without affecting those transactions that execute in consistent mode.



*PaPEC '14*, April 13–16, 2014, Amsterdam, Netherlands.

ACM 978-1-4503-2716-9/14/04. . . $15.00.
http://dx.doi.org/10.1145/2596631.2596637

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming

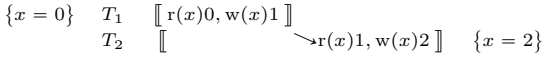
***Keywords*** Transactional memory, Eventual consistency

## 1. Introduction

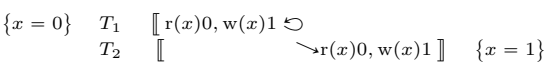
*Transactional Memory* (TM) [8] is an increasingly popular approach to concurrent programming that aims to make it more intuitive as well as efficient by employing and refining the *transaction* abstraction known from database systems. TM also found a use in distributed systems, from universal applications [5, 14] to specialized ones, like geo-distributed key-value stores [4]. While the distributed setting requires a consideration of new problems like partial failures, it also provides an opportunity to use eventual consistency in trade for availability or scalability.

A distributed e-commerce system is one example where a TM application could use eventually consistent transactions to perform stock inquiries and sales data analyses while consistent transactions finalized purchases. The efficiency gain (in terms of response time) should be particularly visible when operating in high-contention environment where a strongly consistent transaction accessing a wide scope of variables is likely to be forced to repeatedly abort and retry (optimistic TM) or wait a long time (pessimistic TM). If the consistency requirement can be temporarily suspended, such a transaction can execute out of order and receive an estimated, and often sufficient result much sooner.

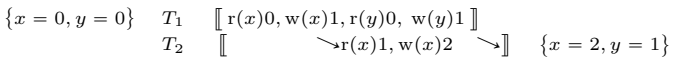
In this paper, we propose a distributed TM that allows eventually consistent transactions within the framework of versioning algorithms [11, 15, 16]—a family of general-purpose pessimistic concurrency control algorithms with strong consistency guarantees. Eventually consistent transactions are designated by the programmer to execute without waiting. They maintain *read-isolation* (i.e., read from a consistent snapshot) and are prevented from modifying the global system state. They are followed by their "proper" consistent counterparts which provide convergence of the final result.




(a) Conflicting transactions—pessimistic execution.




(b) Aborting transaction.




(c) Conflicting transactions—early release.

Figure 1: Versioning algorithm examples.

## 2. Versioning Algorithms

The general idea behind versioning algorithms is that they are pessimistic, so transactions avoid conflicts and never roll back (rather than aborting and retrying when one occurs, as in optimistic TM). This approach is capable of dealing with high contention and has an easier time with issues like irrevocable operations (that cannot be aborted or re-executed) than the optimistic approach. As soon as transactions start, they get a version number for each variable they will access (some prior knowledge obtained via static analysis [10] or typing [15] is needed). Then, the versions are used in conjunction with a *local counter* (also per variable) to defer the transaction's accesses to each variable until preceding transactions finish accessing them. Broadly, a transaction can access a variable if it has a version number equal to the local counter for that variable. Once a transaction commits, aborts, or releases a variable, local counters are incremented. (We present only a rudimentary algorithm since delving into the complexities of a more refined mechanisms introduces no new insights towards the work presented here.) Our system model is one where clients run transactions that access shared variables, each of which is located on one of several, independent remote servers (as opposed to being replicated on multiple nodes of a single "logical" server).

We elaborate on the *modus operandi*, using the examples in Fig. 1. In Fig. 1a transactions $T_1$ and $T_2$ both try to update a variable $x$ (execute a read $\mathrm{r}(x)v$ and a write $\mathrm{w}(x)u$). Since $T_1$ starts (denoted $[\![$) before $T_2$, it has a lower version number for $x$ than $T_2$ (since they are consecutive, they differ by one). $T_1$ can access $x$ once its local counter is equal to the transaction's version of $x$. Once $T_1$ finishes (commits, $]\!]$), the local counter is incremented, so $T_2$ can then start accessing $x$. In effect $T_2$ defers access until $T_1$ completes. Note then, that transactions are synchronized per variable, so that if transactions do not access the same variables, they will execute in parallel. In Fig. 1b, a similar situation occurs, except $T_1$ does not commit, but aborts (denoted $\circlearrowleft$). Since $T_2$ waits for $T_1$ to complete, $T_2$ observes the state after $T_1$ rolls back. A similar case is shown in Fig. 1c, except that here $T_1$ accesses two variables $x$ and $y$. Versioning algorithms allow early release, so the algorithm may determine which operation on $x$ is last (see [11]) or the programmer can use a release operation. In such a case, $T_1$ increases the local counter for $x$ after the last operation on $x$ ($\mathrm{w}(x)1$) instead of on commit. In effect $T_2$ can access $x$ while $T_1$ is still running. This allows some versioning algorithms to increase their efficiency. However, since it is possible for $T_1$ to rollback after releasing $x$, $T_2$ may be required to defer committing until $T_1$ commits. If $T_1$ aborts, $T_2$ can also be forced to abort.

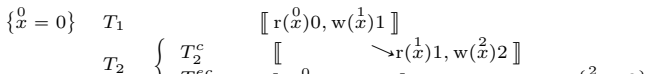



Figure 2: Eventually consistent execution of $T_2$.

Versioning algorithms are distributed and are capable of delivering strong safety and consistency guarantees.

## 3. Eventual Consistency Extension

We propose to extend the versioning algorithms with a mechanism that allows certain *eventually consistent* transactions to execute quickly, without waiting for currently running transactions. When they commence, such transactions grab the most recent consistent snapshot of all the variables they need of those snapshots that can be obtained without waiting. Once the snapshot is buffered, these transactions operate only on the buffers, to avoid waiting during reads and invalidating the global state on writes. Thus, this mode relaxes safety—the client may initially see an inconsistent view (although one generated using read-consistent data) and, since his updates are not propagated, has a different impression of the global state. Thus, the state must eventually be converged, and so, the transaction is concurrently re-executed in consistent mode to fix the client's view and apply modifications. Note that other clients only see the execution of the consistent transaction.

Eventually consistent transactions are meant to operate on a consistent snapshot and are specified as follows. Let $\overset{i}{x}$ denote shared variable $x$ in version $i$ and let $V_k$ be an access-set of transaction $T_k$ containing a set of variables in versions accessed by $T_k$. Then, let us distinguish two modes of executing $T_k$: an eventually consistent mode (as $T_k^{ec}$) and the regular mode (as $T_k^c$). Let $T_k^{ec}$ be any eventually consistent transaction from the set of all such transactions $\mathbb{T}^{ec}$ and let $T_k^c$ be any consistent transaction from $\mathbb{T}^c$. Let $\mathbb{S} \xrightarrow{T_k} \mathbb{S}'$ be a transition from system state $\mathbb{S}$ to system state $\mathbb{S}'$ caused by an atomic execution of $T_k$ where a state is a set of all shared variables in current versions $\{\overset{i}{x}, \overset{j}{y}, ...\}$. Finally, let $\mathbb{S}_0$ be the initial state. Then, given any $T_k^{ec}$, these conditions must be met:

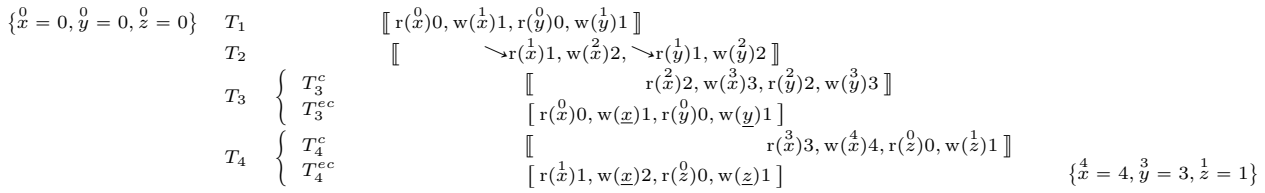



Figure 3: Eventually consistent execution of $T_3$ and $T_4$.

1. For any $\overset{i}{x} \in V_k$, $i$ is a version of $x$ such that, if for some (non–eventually-consistent) $T_q$ it is true that $\overset{i}{x} \in V_q$, then $T_q$ released $\overset{i}{x}$ or $T_q$ is committed.
2. Given $V_k$, there exists state $\mathbb{S}$ (either $\mathbb{S}_0$ or one resulting from a transition $\mathbb{S}' \xrightarrow{T_q} \mathbb{S}$) such that $V_k \subseteq \mathbb{S}$.
3. There is no $\overset{i}{x} \in V_k$ such that $T_k^{ec}$ writes to $\overset{i}{x}$ (instead $T_k^{ec}$ writes to a non-shared buffer $\underline{x}$).

In addition, any $\overset{i}{x} \in V_k$ is the most recent (i.e., the largest) version of $x$ that meets conditions 1–3 when $T_k^{ec}$ starts and $T_k^{ec}$ does not increment the version of $x$ or its local counter.

The gist of this idea is shown in Fig. 2, where transaction $T_1$ executes normally and $T_2$ is eventually consistent. Thus $T_2$ is executed concurrently in two modes as $T_2^{ec}$ and $T_2^c$. $T_2^{ec}$ is the relaxed version of the transaction, which takes a consistent snapshot of its access set and begins without waiting for $T_1$ to finish working on $x$. Therefore, instead of reading from version $\overset{1}{x}$, $T_2$ reads from the earlier $\overset{0}{x}$. In effect, the client first sees a read-consistent result of execution of $T_2$, but one that may be outdated in reference to the global state. Since any writes on the basis of this data likely would be globally inconsistent, the write to $x$ in $T_2^{ec}$ is done only to a local buffer ($\underline{x}$). Thus, it is not visible to other transactions outside of $T_2$, so the inconsistent state does not propagate or affect other transactions. Simultaneously, $T_2^c$ executes in accordance to some original versioning algorithm and produces the final result after a delay and performs all the write operations required by $T_2$. In effect, the client's final view becomes consistent.

Fig. 3 contains a more involved example of early-release–capable transactions operating on multiple variables. Transactions $T_1$ and $T_2$ increment $x$ and $y$ and both release $x$ before committing. Transaction $T_3$ increments $x$ and $y$. It is eventually consistent, so it is split into $T_3^{ec}$ and $T_3^c$. $T_3^{ec}$ executes instantly and uses versions $\overset{0}{x}$ and $\overset{0}{y}$. $T_3^{ec}$ does not use $\overset{1}{x}$ even though it is available to $T_3$, because the snapshot of $\{\overset{1}{x}, \overset{0}{y}\}$ would not be internally consistent (see Condition 2). That is, given a transition $\mathbb{S}_0 \xrightarrow{T_1} \mathbb{S}_1$, $\overset{1}{x}$ belongs in $\mathbb{S}_1$ and $\overset{0}{y}$ and $\overset{0}{x}$ are both in $\mathbb{S}_0$. Transaction $T_4^{ec}$, on the other hand, accesses $x$ and $z$, so it is capable of using $\overset{1}{x}$, because $\overset{1}{x}$ and $\overset{0}{z}$ do create a consistent snapshot. However, both $T_3^{ec}$ and $T_4^{ec}$ work on stale data, so their consistent counterparts $T_3^c$ and $T_4^c$ are run so that clients' views eventually converge on a consistent state.

Since transactions in versioning algorithms know their write sets in advance, the mechanism can be introduced to the original versioning algorithms by adding three additional structures for any variable $x$: a buffer for storing a copy of the most recent version of a committed variable $B^c(x)$, a similar buffer for a variable that was released early $B^r(x)$, and a register $F(x)$ for keeping track of the access set of the transaction that released $x$ early minus any other variables that this transaction also released early. Then, any $T_q$ needs to copy the latest version of variables in $V_q$ to $B^c$ during commit. When releasing $x$ early, $T_q$ needs to copy the latest version of $x$ to $B^r$ and also to register any variables to $F(x)$ that are in $V_q$ but were not yet released. Then, any eventually consistent transaction $T_k^{ec}$ must read $x$ from $B^r$ if $V_k \cap F(x) = \varnothing$. Otherwise $T_k^{ec}$ must read $x$ from $B^c$. In effect $T_k^{ec}$ satisfies our specification with respect to its access set. $T_q$ can clear $F(x)$ on commit.

The extension allows versioning algorithms to improve client response time where the consistency of the client view can be estimated. The cost incurred by the modification of the algorithm is minimal, since in terms of network communication, only a transmission of the access set is required, and in terms of space, only three additional structures are needed (as opposed to copies per transaction). It is also important to note that, excluding the transactions executing in inconsistent mode, the system preserves its original properties. On the other hand, side effects of eventually consistent transactions must be minded by the programmer. Since an eventually-consistent transaction is re-executed in order to converge, any non-transactional operations within will be re-executed. If these are irrevocable operations, re-execution may be dangerous: e.g., a non-reentrant lock may be re-acquired and cause a deadlock.

Our future research includes a number of extensions of the idea presented here. A straightforward one is opportunistically to re-use in consistent mode the effects of the transaction's execution in relaxed mode. Given a transaction $T_k$ executed as $T_k^{ec}$ and $T_k^c$, if no variable from $T_k$'s readset is modified between $T_k^{ec}$ commits and $T_k^c$ begins, the latter could benefit by updating the variables from $T_k$'s writeset with the values buffered by $T_k^{ec}$ instead of executing all the reads and local code.

## 4. Related Work

Below we compare our approach with related work and example systems. The readers should bear in mind, however, that although we managed to eliminate some restrictions present in these systems, our model does require some limited synchronization, that others may not require. Our contribution is to show that we can significantly relax synchronization (but not eliminate it completely) and still be able to achieve useful properties.

Weaker consistency models are widely used in practice in replicated systems. For instance, in Bayou [13], a weakly connected replicated storage system, conflict resolution is done automatically using user-defined application-specific merge operations. Amazon's Dynamo [6] provides eventual consistency, where updates are propagated to replicas asynchronously, so reads and writes may operate on stale versions and be reconciled later. It requires the client to resolve version (or consistency) conflicts whereas in our system the client can either accept inconsistent data quickly (as returned by ec-transactions) or may choose to wait and obtain the globally consistent data returned by the c-transactions that always give the most accurate version of data. Thus, we simplify the choice. Pileus [12] provides a transactional geo-replicated key-value store where consistent primary replicas propagate changes to eventually consistent secondary replicas. Writes are limited to primary replicas and reads can be done from any replica. Then, much like in our work the client can select whether consistent or eventually consistent state is accessed. However, this is done by selecting replicas rather than just switching modes, which means complicating network communication (i.e., if many clients want to read consistent data a bottleneck is likely). Most importantly, however, the replicated model used in all three examples above is different than the one we present here, more akin to a service-oriented environment or a distributed key-value store. Here, the purpose of consistency is to preserve the correctness of client views rather than uniformity of remote resources.

There were also attempts at relaxing the consistency of transactions and TM. E.g., view transactions [1] operate on a consistent snapshot but may commit in a different snapshot, if its user-specified subset is valid, i.e., such that had the transaction operated on the commit-time snapshot, the visible effect would be the same. Elastic transactions [7] are each composed of smaller transactions that work on consistent states, but which may be mutually inconsistent. However, in such systems inconsistent views are not later reconciled, so consistency is relaxed permanently, not temporarily.

Graph revisions [3] allow to obtain eventual consistency at the central server through loosely synchronized interaction of all distributed clients with the server. Our model is fully distributed (no central server) with data and clients located on various nodes and synchronized through versions. More importantly, regular c-transactions maintain globally consistent view at all times, regardless of interaction between clients and servers that hold data. In [2] the researchers propose a transactional interface to abstract query-update data stores with eventual consistency using the revision diagram model. Unlike the work shown here however, these transactions must always commit, which limits their expressiveness and fault-tolerance capability. Plus, our work aims to have eventually consistent transactions alongside consistent ones.

Conflict-free Replicated Data Types (CRDTs) [9] allow to converge replicated data to be a globally consistent state without update synchronization, if only data satisfy the monotonic semi-lattice property (roughly: the order of updates on different replicas can differ). Our model does not impose such restrictions on shared data, albeit it does require some synchronization. Moreover, we target general distributed transactions, not only replication.

## Acknowledgments

The project was funded from National Science Centre funds granted by decision No. DEC-2012/07/B/ST6/01230.

## References

[1] Y. Afek, A. Morrison, and M. Tzafrir. Brief announcement: View Transactions: Transactional Model with Relaxed Consistency Checks. In *Proc. PODC'10*, July 2010.

[2] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *Proc. ESOP'12*, Apr. 2012.

[3] S. Burckhardt, M. Manuel Fähndrich, D. Leijen, and B. Wood. Cloud types for eventual consistency. In *Proc. ECOOP'12*, June 2012.

[4] J. C. Corbett and et al. Spanner: Google's Globally-Distributed Database. In *Proc. OSDI'12*, Oct. 2012.

[5] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proc. PRDC'09*, Nov. 2009.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP'07*, Oct. 2007.

[7] P. Felber, V. Gramoli, and R. Guerraoui. Elastic Transactions. In *Proc. DISC'09*, Sept. 2009.

[8] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA'93*, May 1993.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proc. SSS'11*, Oct. 2011.

[10] K. Siek and P. T. Wojciechowski. A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java. In *Proc. FMICS'12*, Aug. 2012.

[11] K. Siek and P. T. Wojciechowski. Brief announcement: Towards a Fully-Articulated Pessimistic Distributed Transactional Memory. In *In Proc. SPAA'13*, July 2013.

[12] Y. Sovran, R. Power, M. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. SOSP'11*, Oct. 2011.

[13] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP'95*, Dec. 1995.

[14] A. Turcu, B. Ravindran, and R. Palmieri. HyFlow2: A High Performance Distributed Transactional Memory Framework in Scala. In *Proc. PPPJ'13*, Sept. 2013.

[15] P. T. Wojciechowski. Isolation-only Transactions by Typing and Versioning. In *Proc. PPDP '05*, July 2005.

[16] P. T. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A Framework for a Synchronisation–Augmented Microprotocol Approach. In *Proc. IPDPS '04*, Apr. 2004.