Highly-Available and Consistent Group Collaboration at the Edge with Colony

Ilyas Toumlilt, Pierre Sutra, Marc Shapiro

Highly-Available and Consistent Group Collaboration at the Edge with Colony

Ilyas Toumlilt
Sorbonne Université–LIP6
Inria
Paris, France
ilyas.toumlilt@lip6.fr

Pierre Sutra
Télécom SudParis
Paris, France
pierre.sutra@telecom-sudparis.eu

Marc Shapiro
Sorbonne Université–LIP6
Inria
Paris, France
marc.shapiro@acm.org

Abstract

Edge applications, such as gaming, cooperative engineering, or in-the-field information sharing, enjoy immediate response, autonomy and availability by distributing and replicating data at the edge. However, application developers and users demand the highest possible consistency guarantees, and specific support for group collaboration. To address this challenge, Colony guarantees Transactional Causal Plus Consistency (TCC+) globally, strengthened to Snapshot Isolation within edge groups. To help with scalability, fault tolerance and security, its logical communication topology is forest-like, with replicated roots in the core cloud, but with the flexibility to migrate a node or a group. Despite this hybrid approach, applications enjoy the same semantics everywhere in the topology. Our experiments show that local caching and peer groups improve throughput and response time significantly, performance is not affected in offline mode, and that migration is seamless.

CCS Concepts: • **Computer systems organization** → **Peer-to-peer architectures**; • **Networks** → **Cloud computing**; • **Information systems** → *Key-value stores*; *Distributed storage*; • **Computing methodologies** → *Distributed algorithms.*

Keywords: Edge Computing, Collaborative Computing, Causal Consistency, CRDTs, Peer-to-Peer Systems

ACM Reference Format:
Ilyas Toumlilt, Pierre Sutra, and Marc Shapiro. 2021. Highly-Available and Consistent Group Collaboration at the Edge with Colony. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Virtual Event, Canada.* ACM, Québec (online), Canada, 16 pages. https://doi.org/10.1145/3464298.3493405

1 Introduction

Internet-scale collaboration is a growing application area, as evidenced by games such as Overwatch or Ingress, shared editors such as Google Docs, Microsoft Office 365 or Apple iWork, or file-sharing systems such as Dropbox or Nextcloud. Mobile devices with Augmented Reality capabilities support location-aware games such as Pokémon Go and Harry Potter Unite, or collaborative 3D modelling and manufacturing applications [43, 49, 59].

Existing systems are cloud-based, sometimes providing ad-hoc application-level caching. Consistency violations are common, baffling users and vexing application developers [10, 18, 19, 50, 63]. Support for offline operation is spotty. Users interact through the cloud only, even when direct communication would be possible. These systems lack collaboration features such as group management or versioning.

This paper presents the Colony database and middleware designed to address these issues. A top requirement is an *edge-first* approach [24] that locates data on device, to provide availability, fast and seamless response independently of network connectivity and of location, and so that users have ownership of their data. However, this makes it challenging to satisfy consistency and freshness expectations. To answer this challenge, Colony takes a hybrid approach, and provides the highest consistency guarantees compatible with availability (TCC+), strengthened further (to SI) in well-connected zones. CRDT data types ensure convergence without rollbacks. A related challenge is the overhead of concurrency metadata (fat vector clocks), which we limit thanks to a flexible forest topology and to SI zones.

To address group collaboration requirements, Colony versions data, enables an edge group to share without relying on the cloud, and supports collaborative security features. Our design provides uniform access to data across a spectrum spanning core cloud to the far edge.

Finally, we address a number of design and implementation challenges, including disconnected operation, consistency under migration, total-order consensus at the edge, and avoiding single points of failure despite the forest topology.

We claim the following contributions:

- A decentralised database architecture, designed for collaborative applications, that provides a continuum spanning from the core cloud to the far edge.
- A hybrid consistency model, based on causal and transactional guarantees globally, strengthened to total-order consistency in edge collaboration groups and in geographical proximity.
- A scalable metadata and topology design that bounds the overhead of the required consistency metadata, and that

supports seamless disconnection or migration of a node or of a whole group.
- A novel approach to access control that leverages the consistency model.
- Efficient design and implementation of Colony and an experimental evaluation showing our approach benefits.

Our experimental evaluation demonstrates that: local and group caching improve throughput by 1.4× and 1.6× respectively, and response time by 8× and 20×, compared to a classical cloud configuration; performance in offline mode remains the same as online; both the offline/online transition and migration are seamless.

2 Support for cooperation at the edge

Edge computing enables a new class of distributed cooperative applications.[1] Consider for instance a distributed multiplayer game; the state of the game universe is shared, persistent, and mutable, typically stored in a database. Instead of a centralised cloud, state should be decentralised, distributed and replicated across edge devices, to enable quick application response (no waiting for a network round-trip) and availability (the application can read and write data, even when disconnected from any remote server). The local-first approach provides users with a sense of ownership and helps with confidentiality [24]. Nonetheless, we leverage the strengths of the cloud infrastructure, i.e., stronger consistency and well-managed, abundant resources.

Edge applications must remain available despite unpredictable network connectivity, disconnection, and mobility. In this context, strong consistency is not possible globally [16].

Consistency anomalies frustrates users and complicate correct application coding. We take a hybrid approach: globally, provide the strongest model compatible with availability; and locally, strengthen the guarantees where possible.

2.1 Global consistency guarantee: TCC+

In the edge context, two different consistency models have been explored. Although they are incomparable, both have been proved to be a strongest possible model compatible with availability under partition. One is Monotonic Prefix Consistency (MPC), which combines the per-process orders into a global total order; however a process is exposed to arbitrary rollbacks [17]. We argue that a client losing an unpredictable amount of work is an unacceptable user experience.

Our preferred alternative is Causal Consistency (CC) [4]. Intuitively, if a client observes some update, it also observes all preceding updates. Only concurrent updates may be observed in different orders.

CC can be enforced locally and does not require consensus; the drawback is that tracking the partial order in CC can have a heavy metadata cost, as we discuss later. On top of CC, atomic transactions and convergence guarantees can be supported without impacting availability [1, 64], a model we call *Transactional Causal Plus Consistency (TCC+)*. Section 3.1 formalises the TCC+ guarantees.

2.2 Local strengthening: data centre

Nodes that are strongly connected to each other can provide even stronger guarantees, totally ordering updates upfront, for instance Snapshot Isolation (SI). SI is stronger than MPC, as it does not suffer rollbacks, and than CC, as it totally orders updates; its metadata cost is low. We call a set of nodes that enjoy SI among themselves an *SI zone*.

One kind of SI zone is a data centre (DC). A DC has a large number of parallel servers, connected through a high-quality network. Colony executes transactions across multiple servers in the same DC under SI [1, 12].

From the perspective of global TCC+, a DC behaves like a single sequential process, thus limiting metadata size.

2.3 Local strengthening: Peer groups

Another kind of SI zone is the peer group. Inconsistency is especially problematic to users who communicate directly, outside of the database. For instance, in the enhanced-reality game Pokémon Go, there is an anomaly where two users in close proximity can both become the owner of the same game character, confusing them [28]. This and similar examples argue for groups with stronger consistency.

In Colony, edge nodes that are close to each other can form an SI zone, called *peer group*. Their mutual strong consistency improves user experience and metadata management. To maintain availability, the system should support disconnecting and/or migrating the group, without losing the consistency guarantees.

2.4 Security requirements

To support collaboration, Colony supports versioning and trust management.

As the edge device executes and merges updates, data can remain encrypted end-to-end; the untrusted cloud serves merely for transport and persistence [26].[2]

However, the edge use case poses new security challenges. Information is exposed on compromised edge nodes [57]; concurrent changes to the security policy changes and to data weaken security [60, 62]; and decentralised key management is problematic [26]. We alleviate these difficulties

[1] Loosely, we distinguish three areas in the network. The *core* cloud is made of a few tens of resource-rich, well-managed data centres (DCs). At the opposite end of the spectrum, millions of resource-limited and mismanaged *far-edge* devices, such as mobile phones or IoT devices. In between, thousands of *border* nodes, such as CDN Points-of-Presence (PoPs), metropolitan data servers, or 5G MEC servers. We refer as "edge" to the union of the border and far edge, and as "infrastructure" to the union of the core and border.

[2] End-to-end encryption is not yet implemented in the current prototype.

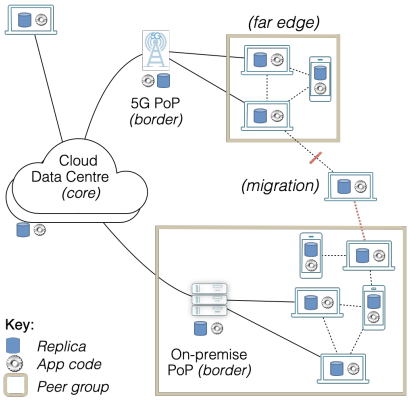


Figure 1. Example Colony topology. A small number of DCs forms the core. A far edge device connects either directly to a DC, or via a point-of-presence (PoP) server at the border. A peer group contains devices in geographical proximity. Note the device migrating between subtrees.

by leveraging the cloud, e.g., for authentication and key management.

Our focus in the security area is support for group collaboration. Every data object comes with an Access Control List (ACL) that describes what updates users are allowed to perform. The system preventatively enforces ACL in edge devices. Because an edge device may be compromised, every node double-checks the updates it receives, and masks an update that is not allowed by the corresponding ACL, and transitively any update that depends on it. Thus a correct node never depends upon a state that violates the security policy.

3 Protocol design

We turn now to a system design for satisfying the above requirements efficiently. Our design is an extension of the SwiftCloud approach [64]. Colony uses caching and replication to ensure that a client can execute locally. The system must remain safe at all times; specifically, the data observed by a client always satisfies the TCC+ and security invariants defined below. It should also remain available.

The trade-off is that, during some failures, liveness cannot be ensured. A client cannot make progress in two cases: if it requires data that cannot be retrieved; or if it runs out of storage. Furthermore, there are corner cases (described later) where a client commits updates, but they cannot become visible. The above situations are temporary, and last only until the problem is repaired.

Our system ensures convergence by using operation-based CRDTs, which merge concurrent conflicting operations deterministically [44]. As underlined in the Introduction, supporting causal consistency (CC) can have high metadata overhead; our design bounds metadata to a small size. Similarly to recent CC designs [1, 8], Colony separates (internal) state management from (external) visibility: the backend layer transmits and stores states efficiently, without regard for correctness, whereas the visibility layer manages metadata and ensures that an application observes only those states that satisfy the TCC+ guarantees.

3.1 The TCC+ guarantees

We now tersely specify the TCC+ guarantees. We use the following notations and definitions. Nodes (at any level of the topology) are noted p, p'. A node behaves sequentially, executing one transaction at at time. A node might fail, in which case it ceases executing (fail-stop); a node that does not fail is said correct. x, y designate data objects. Transactions are noted T, T'. A transaction consists of a sequence of reads and updates. A transaction is interactive, i.e., the objects it accesses are not known in advance. A read has no side effect; an update does not return a response value. We write $a \in T$ when operation a (a read or an update) belongs to transaction T. A transaction executes at a single replica; if it commits, its updates are broadcasted to be replayed by the other replicas. An operation is noted $a, b, \ldots$; in more detail, updating x is written $u(x)$, and a read $r(x)$. The response value of $r(x)$ is $\text{res}(r(x))$.

Following Viotti and Vukolić [58], an abstract execution $A = (H, \overset{vis}{\rightarrow}, \overset{ar}{\rightarrow})$ consists of an interleaving of operations executed by the nodes, or history (H), a visibility relation ($\overset{vis}{\rightarrow}$), a partial order that accounts for the propagation of updates in the system, and the arbitration relation ($\overset{ar}{\rightarrow}$), a total order over H that helps to resolve concurrency conflicts. The order in which nodes execute operations is called the program order. The happened-before relation ($\prec$) is the transitive closure of the union of visibility and program order [58].

Hereafter, we consider only transactions that commit; we can safely ignore the operations of a transaction that aborts, since it has no effect.

The phrase "visible in node p" refers to an operation that is visible by some operation executed at node p.

Each object starts in some known initial state. The return value of a read is computed according to the semantics of prior updates to the object (including updates in the same transaction). That is, for each read operation $r(x)$, $\text{res}(r(x))$ results from some linearization $l_{r(x)}$ of the updates visible to $r(x)$ consistent with $\prec$.

TCC+ is defined by the following invariants.

Causal Consistency (CC). Causal consistency requires that every update that happened-before an operation is visible to that operation, and that arbitration is consistent with happened-before. Formally, $(\prec \subseteq \overset{vis}{\rightarrow}) \wedge (\prec \subseteq \overset{ar}{\rightarrow})$.

Rollback-freedom. Once a node has read a value, it does not roll it back: If $r(x) \prec_p r'(x)$ then $l_{r(x)}$ prefixes $l_{r'(x)}$.

Strong Convergence. Any two nodes that observe the same set of updates read the same value. Formally, $\forall r(x), r'(x) : (\forall u(x); u(x) \xrightarrow{vis} r(x) \Leftrightarrow u(x) \xrightarrow{vis} r'(x)) \implies \text{res}(r(x)) = \text{res}(r(x))$.

The above invariants constrain the behaviour of individual operations. Below, we formalise the fact that a transaction is atomic (i.e., all-or-nothing). We define the following equivalence relation, written ≡: if operations a and b are in the same transaction T, then $a \equiv b$. For some relation R over the set of operations, we say that R is left-compatible with ≡ when for any three operations a, b and c, if $a \equiv b$ and $(a, c) \in R$ then $(b, c) \in R$. Right-compatibility is defined symmetrically, that is $b \equiv c \wedge (a, b) \in R \implies (a, c) \in R$. Relation R is compatible with ≡ when it is both left- and right-compatible with it.

Atomicity. If two updates occur in the same transaction, then they are visible atomically, and arbitrated in the same way. Formally, visibility and arbitration are compatible with transactional ≡.

Snapshot. A transaction takes all its reads (independently of their order) from a same snapshot, which is sound both causally and for the atomicity relation.

Additionally, the following liveness property should hold:

Eventual Visibility. If two correct nodes p and p' are not permanently disconnected from one another, and $u(x)$ is visible in p, then eventually $u(x)$ is visible in p'.

TCC+ extends Transactional Causal Consistency, as defined by Zawirski et al. [64], with the Strong Convergence and Rollback-Freedom properties. This ensures that progress is monotonic at each node.

To illustrate the concepts in this section, consider the history in Figure 2, which depicts the evolution of a CRDT counter (x) when nodes execute increment operations (inc), and propagate such updates (depicted by arrows).[3]

The history in the figure is causally consistent. Indeed, every new increment updates the counter to a state also containing the preceding operations (e.g., after event ⑥, the counter value is 2). Similarly, there is no roll-back at any node. Nodes that received the same increments (e.g., events ⑦ and ⑧) are in the same state; therefore this history satisfies strong convergence. Moreover, since every transaction contains a single operation, the history trivially ensures the atomicity and snapshot requirements.

[3]For now, ignore the version, commit and snapshot information, which will be detailed later.

3.2 Strengthening to SI

Colony strengthens the above TCC+ guarantees to strong consistency in an SI zone.

In a SI zone, Colony ensures Snapshot Isolation (SI). This means that $\xrightarrow{ar}$ is gapless [45], i.e., for any operation b visible to a, every operation $c \xrightarrow{ar} b$ is also visible to a.

3.3 Bounding metadata

This and the following sections detail the logic to achieve the above consistency guarantees.

Supporting CC requires metadata, which can represent a substantial overhead; this section explains how Colony bounds metadata to a small size.

The CC invariant dictates that an update may become visible only if its *dependencies* (i.e., the updates that happened-before it) are themselves visible. To check this, when transmitting an update, Colony piggy-backs some associated *visibility metadata*, a vector timestamp (or version vector) that summarises its dependencies [14, 34]. Vector timestamps support efficiently computing the set of missing dependencies [38].

A precise representation of the happened-before order among N concurrent writers requires a vector of size $\geq N$ [11]. As N grows, the overhead on every message quickly becomes unacceptable.[4] The following sections describe some techniques that we use to keep the size small, at the cost of spuriously ordering some concurrent events.

3.4 Topology and metadata design

We first turn to the topology design (illustrated in Figure 1) and the metadata design.

Each DC forms an SI zone; therefore, the updates of a given DC are totally ordered; externally, it behaves as a single sequential node. On the other hand, DCs are connected in a full peer-to-peer mesh; their updates are partially ordered, which requires a vector. Since each DC appears sequential, a timestamp vector V of size N suffices to a point in the CC partial order between DCs. Component $V[i]$ numbers the (sequentially ordered) transactions committed at DC i.

The *least upper bound* (LUB) of two vectors is defined as their component-wise maximum. Each node maintains its *state vector*, which is the LUB of the commit timestamps (defined next) that it has observed.

A transaction has a unique identifier called its *dot* [2].

Communication between DCs is a full mesh. Edge nodes (border or far-edge) are partitioned into distinct trees, forming a forest, as illustrated in Figure 1. Each tree is rooted at a specific DC, which we call its *connected DC*.[5] A subtree may detach itself from its parent and *migrate* to a different tree, e.g., to accommodate mobility or a failure.

[4]In Colony each component of the vector is 8 bytes, in order to store a monotonic clock that does not wrap around.

[5]A peer group counts for a single node in the tree.

3.5 Transaction metadata

We now describe the metadata associated with a transaction T: its snapshot and commit timestamp vectors, and its dot.

Transaction T's *snapshot vector* $T.S$ describes the (previous) transactions it depends upon. $T.S$ forms a snapshot closed under CC and atomicity. The meaning of $T.S[j] = n$ is the following: T reads from all the transactions T' committed at DCj up to time n, and no later, i.e., such that $T'.C[j] \leq n$.

A read-only or aborted transaction terminates without side effects. The *commit vector* $T.C$ of an update transaction represents the point where it commits.[6] It is greater than its snapshot vector; if the transaction commits at DCi, they differ only at index i, i.e., $T.S[i] < T.C[i] \wedge \forall j \neq i : T.S[j] = T.C[j]$, where $T.C[i]$ is a timestamp assigned by DCi.

Transaction T is before T' if $T.C \leq T'.S$. If neither of T or T' is before the other, they are said concurrent.

Finally, a transaction has a unique timestamp called a *dot* $T.D$, which both serve as a unique identifier and provides the (total) arbitration order between concurrent transactions (as defined in Section 3.1).

3.6 In-DC transaction protocol

Let us describe how the system computes metadata in the simple case of a transaction that executes within some DCi. By default, $T.S$ is assigned the current state vector of DCi. The system checks that $T.S$ represents a consistent cut [1, 41] such that $T.S[i] \leq current\_time$. Its unique dot is $T.D := (current\_time, i)$. The commit protocol is a standard two-phase commit among the servers of DCi (we use ClockSI [12]). The commit vector is equal to the snapshot vector, except that $T.C[i] := current\_time$. Object versions created by the transaction are marked with version timestamp $T.C$.

As Colony objects are operation-based CRDTs, materialising a version may require to apply multiple updates [9, 44]. Conversely, concurrent transactions that update the same CRDT object can be merged and by default do not abort, although it can abort for semantic reasons, e.g., if it would violate some invariant; we assume a higher level of concurrency control to detect such violations [3, 20, 21], which is out of the scope of this paper.

We illustrate the in-DC transaction lifecycle in Figure 2, events ⓪ through ④. Focus on the three DCs, numbered 0, 1 and 2, and on the CRDT counter x.[7] ⓪ All DCs have a copy of x with value 0 and version timestamp $[0, 0, 0]$. The three DCs are in state $[0, 0, 0]$. ① DC0 executes transaction T0. Its snapshot vector is set from its current state, at $[0, 0, 0]$. T0 increments x. Its commit vector is $[1, 0, 0]$, i.e., its snapshot vector incremented by 1 at the component for DC0. It commits version $[1, 0, 0]$ of x, with value 1. ② Concurrently, DC1 executes T1, with the same snapshot. T1 also increments x. As x is a CRDT, T1 can also commit; its commit vector is $[0, 1, 0]$ and it updates x to version $[0, 1, 0]$ with value 1. At this point, T0 is visible only to DC0, and T1 only to DC1. ③ DC0 replicates T0 to DC2, where x has version $[1, 0, 0]$. ④ DC1 replicates T1 to DC2. All three versions of x are visible at DC2, as well as a merged version with least-upper-bound timestamp $[1, 1, 0]$. As this version includes the increments from both T0 and T1, its value is 2.

3.7 Basic edge transaction protocol

A transaction may execute and commit in an edge node. In this case, commit is asynchronous, i.e., for availability, the edge node continues to execute further transactions without waiting for the DC to assign its commit vector.

Starting a transaction T is similar to the in-DC case: the edge node assigns its snapshot, and a dot using the edge node's unique identifier. The transaction commits locally at the edge node, which can immediately start another dependent transaction. Until it receives the DC's acknowledgement, the commit timestamp remains *symbolic*, i.e., indeterminate, subject only to the invariant $T.S < T.C$.

Eventually, the edge node sends the transaction to its connected DCi, which acknowledges with a concrete commit vector. Similarly to the in-DC case, the commit vector differs from the snapshot only in the component corresponding to the connected DC: $T.C[i] := current\_time$. However, because of migration, index i cannot be predicted, as described in the next section.

Let us return to Figure 2 to illustrate the lifecycle of edge transactions. ⑤ Edge node A pulls x into its interest set, then starts transaction TA1 with snapshot vector $TA1.S = [0, 0, 0]$. TA1 reads version $[0, 0, 0]$ of x. TA1 increments x. TA1 commits; its commit vector is still uncertain, noted with the symbolic $TA1.C = [\alpha, \beta, \gamma] > [0, 0, 0]$. ⑥ EdgeA starts a second transaction TA2. To be able to read the writes of TA1 from the local cache, it assigns snapshot vector $TA2.S = [\alpha, \beta, \gamma]$. TA2 increments x and commits with symbolic $TA2.C = [\alpha', \beta', \gamma'] > TA2.S = [\alpha, \beta, \gamma]$. ⑦ Concurrently, EdgeA sends TA1 to DC0. Similarly to an in-DC transaction, DC1 assigns the commit vector $TA1.C = [\alpha, \beta, \gamma] := [1, 0, 0]$. ⑧ EdgeA receives the updated descriptor for TA1 and fills in the concrete values $TA1.C = TA2.S = [1, 0, 0]$.

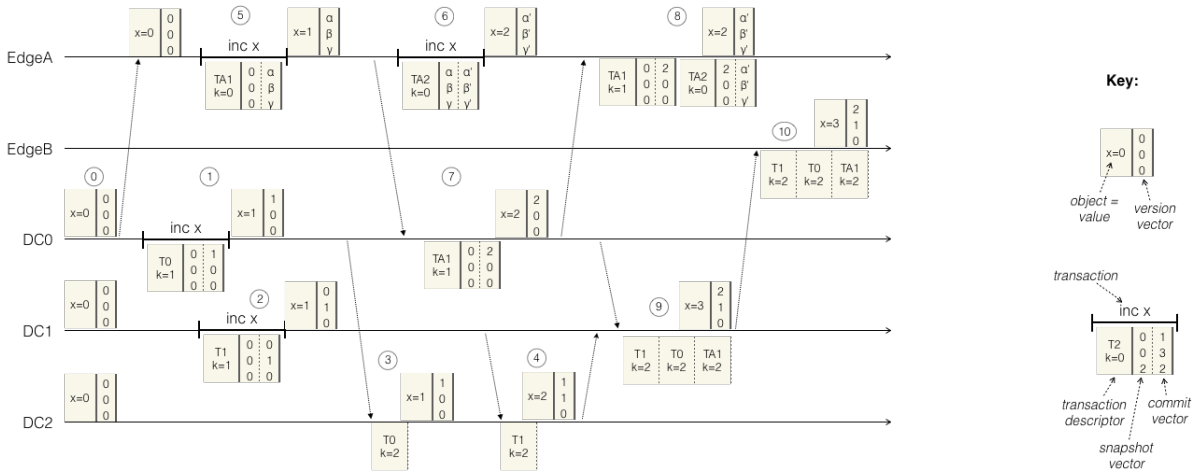
3.8 Node migration and K-stability

A fixed forest is inflexible, and a single fault may have a disproportionate impact. Therefore, Colony supports migrating a node and the subtree attached to it. Ideally, node migration should be seamless and transparent to applications, but unfortunately this is not completely possible.

Migration creates some extra complications to the edge transaction protocol, which we consider next. For simplicity, we consider a single edge node case, focus on the migration

[6] In fact, to tolerate faults, it may have multiple equivalent commit vectors, as described later, in Section 3.8.

[7] Ignore for now the two edge nodes, and references to k and to stability.



This system has three data centres DC0, 1, and 2, and edge nodes A and B. Vector components refer to DC0, 1 and 2 respectively. Dots are omitted from the figure. The k-stability objective is 2.

⓪ Initially, the DCs observe $x = 0$ with version vector $[0, 0, 0]$.

①, ② Transaction T0 (resp. T1) increments x at DC0 (resp. DC1), committing $x = 1$ with version $[1, 0, 0]$ (resp. $[0, 1, 0]$). Both have $k = 1$.

③ DC0 transmits T0 to DC2. Being replicated at two DCs, it is 2-stable ($k = 2$).

④ DC1 transmits T1 to DC2. T1 is now 2-stable. DC2 observes two increments, T0 and T1: now $x = 2$ with version $[1, 1, 0]$, the least-upper-bound of the commit vectors of T0 and T1.

⑤ EdgeA caches x. Transaction TA1 increments x and commits locally. Its commit vector remains the symbolic $[\alpha, \beta, \gamma]$. As TA1 has not been transmitted to a DC, it has $k = 0$.

⑥ Transaction TA2 at EdgeA again increments x. Its snapshot vector is symbolic $[\alpha, \beta, \gamma]$; its commit vector is symbolic $[\alpha', \beta', \gamma']$. The value of x at EdgeA is now 2.

⑦ Concurrently, EdgeA transmits TA1 to DC0. DC0 assigns commit vector $[\alpha, \beta, \gamma] := [2, 0, 0]$. TA1 being known in one DC, it has $k = 1$. DC0 observes two increments, T0 and TA1: now $x = 2$, with version $[2, 0, 0]$.

⑧ DC0 transmits back the concrete descriptor of TA1, informing EdgeA that $[\alpha, \beta, \gamma] = [2, 0, 0]$. EdgeA observes local increments TA1 and TA2, hence $x = 2$. T0 is not visible to EdgeA because $T0.k = 1$.

⑨ DC0 transmits T0 and TA1 to DC1, making them both 2-stable.

⑩ T0, T1 and TA1 being 2-stable at DC1, are made visible to EdgeB, where $x = 3$.

Eventually (not depicted): TA2 is delivered to a DC, filling in the values for $[\alpha', \beta', \gamma']$; TA2 becomes 2-stable; all four transactions reach all replicas; all replicas observe $x = 4$.

Figure 2. DC and edge transaction protocols

mechanism, and ignore the policy decision of why or when to migrate, e.g., in response to a network failure.

Avoiding Duplicates. Migration can change the connected DC of the node. Consider the edge transaction protocol described above. Suppose that some edge node sends its transaction T to its connected DCi, loses the connection to DCi, then migrates to DC$j \neq i$. As the edge node does not know whether DCi received T, it sends T again to DCj. Although T might now be received twice, via both DCs, a replica should replay it only once; the transaction's dot $T.D$ serves to filter out such duplicates. To this effect, every node keeps track of the highest dot assigned by another node, and ignores a transaction whose dot is less or equal this value.

K-stability to avoid causal incompatibility. Consider an edge node that migrates from DCi to a new connected DCj. If the state of DCj includes that of DCi, the edge node's dependencies remain satisfied, and migration is seamless. We say that the states are causally compatible. However, it might happen (for instance, because of a communication failure) that an edge transaction T' depends on a transaction T that was visible at DCi but not yet at DCj. The snapshot of T' does not satisfy the CC invariant at DCj, which cannot apply it and cannot assign its commit vector. The edge node remains effectively disconnected, and its transactions are non-visible to the rest of the system. We say the edge node state is incompatible with DCj.

If T was not visible to T', the above dependency could not exist, and the nodes would remain compatible. Thus, one possible approach would be to let transaction T become visible at the edge only once it is known at all DCs. However, a single slow DC would delay edge visibility of all transactions.

Our solution, taken from SwiftCloud [64], is twofold. First, to ensure the Read-My-Writes session guarantee [52], an edge node's transactions are always visible to itself. Second, to decrease the incompatibility probability, transaction T

becomes visible to edge nodes only after it is visible by $\geq K$ DCs, where $1 \leq K \leq N$ [64]. The higher K, the higher the probability that the new DC is compatible with the old one, i.e., that its state includes the dependencies of T'. The value of K is a trade-off between two extremes. If $K = 1$, the probability of incompatibility is high. If $K = N$, one slow DC could prevent all edge transactions from becoming visible.

To illustrate K-stability, refer again to Figure 2. $T.k$ counts the number of DCs where T is stable. The visibility limit is set to $k \geq K = 2$. At ⑧, transaction T0 is not visible to EdgeA, because $TO.k = 1$. At ④, DC1 sends T1 to DC2; now $T1.k = 2$. At ⑨, DC0 transmits T0 and TA1 to DC1; now all three transactions T0, T1 and TA1 have $k = 2$. As DC1 has observed three increments, $x = 3$ with version $[2, 1, 0]$, the least-upper-bound $TC0.C, TC1.C$ and $TA1.C$. Therefore, in ⑩, DC1 can make them visible to EdgeB, where later transactions may depend on $x = 3$ with version $[2, 1, 0]$.

Transaction ordering. Finally, both DCi and DCj may assign different commit vectors, $T.C_i \neq T.C_j$. This could cause an ordering anomaly: if some transaction T' could depend on T with $T.C_i \leq T'.S$, where $T.C_j \not\leq T'.S$; a node that knows only of $T.C_j$ is not aware that T happens before T'. Observe, however, that $T.C_i$ and $T.C_j$ conceptually denote the same point in the TCC+ partial order. To ensure this concretely, Colony considers the two timestamps as equivalent; they already have the same causal past, and the equivalence ensures they have the same causal descendance.

Thus, a same transaction may carry up to N equivalent commit timestamps. We optimise their memory size as follows. Recall that a commit vector differs from the snapshot vector in a single component, that of the DC that accepted it; the others are not significant. Therefore, Colony stores multiple commit vectors into a single vector of size N, containing a significant value only for a DC that accepted the transaction. For simplicity, Figure 2 does not depict this optimisation.

3.9 Transaction migration

Resource-hungry transactions should run in the core cloud rather than the edge. Examples include analytics, or large queries. Colony supports migrating them to a trusted node in the core cloud for execution.

The migrated transaction must have the same effect as if it ran on the edge node; only performance should differ. Thanks to TCC+, it suffices to assign the same snapshot vector.

Thus, the client primes the snapshot with its own state vector and sends the transaction code. Before the transaction starts, the DC must have received the client's local transactions, which that the new one depends upon (Section 5.1.3 explains how we accelerate this). This ensures that every read can be satisfied. The migrated transaction executes in the DC just like a standard local client, and its results are sent back to the requesting edge node.

4 Data management

Colony ensures convergence by using operation-based CRDTs, which merge concurrent conflicting operations deterministically [44]. Similarly to recent CC designs [1, 8], Colony separates (internal) state management from (external) visibility: the backend layer transmits and stores the state efficiently, without regard for correctness, whereas the visibility layer manages metadata and ensures that an application observes only those states that satisfy the TCC+ guarantees.

4.1 Versioning

Colony stores an object persistently as a base version and a journal of updates since the base version. To materialise an arbitrary object version, the cache first reads the base version from the store, and applies the missing updates from the journal. Occasionally, the system advances the base version.

A transaction reads from its snapshot, logs its updates to the journal, and materialises new versions in a private buffer. When the transaction commits, it updates the cache from the buffer. Both the updates recorded in the journal, and object versions that result from committed transaction T, are labelled with vector $T.C$ and dot $T.D$.

4.2 Edge caching

An edge node cannot replicate the whole database, but can only cache some small fraction of it. An edge client may declare *interest* in some object to add it to its node's cache. The connected DC regularly informs the client of updates to its interest set.

At any point in time, the state vector of an edge node is the LUB of the state received from its connected DC (itself $\leq$ the DC's current state vector) and the commit vectors of local transactions. Choosing a snapshot vector $\leq$ the node's state vector ensures that every read could be satisfied either from the local cache or from the connected DC. It may happen that the client requires an object version that cannot be retrieved (in the cache, from the DC or from another node), in which case the transaction cannot proceed. This limitation of availability is inherent to the edge environment.

5 Groups

Colony supports two distinct group mechanisms: the peer group, an SI zone at the edge, and the collaboration group, nodes that update the same data. Peer groups are disjoint, whereas collaboration groups may overlap. All nodes in a peer group are in the same collaboration group.

5.1 Peer groups

A peer group is a set of nodes with high-availability, low-latency connection to one another. It makes sense to provide SI within the group. This enhances the user experience, and simplifies metadata management. A peer group creates opportunities to improve performance, by pooling resources

into a collaborative cache, and to decrease network load to the cloud by collecting the updates from many clients. Conceptually, a peer group consists of four related components, with distinct roles: managing group membership, sharing content within the group, communicating with the outside, and enforcing the SI order. Described below.

5.1.1 Membership. Membership of a peer group is seeded and managed by a single node, called the group's *parent*. The parent maintains a connection to each of the group members, stores their list, and informs them of any membership change. The parent is fixed but arbitrary, possibly located in the DC or on a point-of-presence (PoP) server. A node may serve as a member and a parent at the same time. To join or leave a group, a node contacts the group's parent. The parent responds with the membership list, as well as the session security key (described shortly). When a node migrates between groups, it uses the migration protocol previously described (Section 3.8); the new group must be causally compatible with the node's state.

5.1.2 Content sharing. Using the membership list, the group members and their parent maintain point-to-point connections. Above these connections, they construct a collaborative cache using a simple peer-to-peer protocol. Each member publishes its current interest set to all its neighbours (other members and parent). This subscribes the member to receive all updates to its interest set. When a member updates an object in a neighbour's interest set, it pushes that update in a best-effort manner. Conversely, if a member observes that it is missing an update to its interest set (by examining the visibility log described below), it pulls the transaction from some neighbour. Objects evicted from a cache are unsubscribed to save resources. The parent maintains an interest set that is the union of those of the group members. It subscribes for updates outside the peer group on behalf of its members, as detailed next.

5.1.3 Communicating outside the group. As illustrated in Figure 1, a subtree communicates with another one via some common ancestor. For simplicity, the following description assumes this ancestor is its connected DC.Let us call *synchronisation point* (sync point) a node within a group that communicates with the DC. In the common case, this is the parent, but any member may also unilaterally become a sync point (for instance before migrating a transaction to the DC, Section 3.9), thus avoiding any single point of failure. A sync point sends all missing updates to the DC, and symmetrically subscribes to updates to its interest set. Importantly, the sync point makes updates visible to the DC in the visibility order described in the next section. This ensures that different sync points send identical information.

5.1.4 Transaction protocol for peer groups. A peer group as a whole should behave like a single, sequential edge node, from the perspective of the rest of the system. To ensure sequential ordering, causality and progress within the group, Colony relies on EPaxos [35] within the peer group. Compared to other consensus protocols, EPaxos improves availability and performance, by allowing any group member to become the leader for any transaction, and by minimising synchronisation between non-conflicting transactions.

In addition to improving the user experience, consensus is essential to correct metadata management.

Recall from Section 5.1.3 that possibly multiple sync points send transactions to the DC. Without consensus, conflicting transactions would be sent in different orders, breaking causality and causing unsafe commit vectors.

When a peer node commits a transaction, it submits it to EPaxos. EPaxos ensures consensus on the order in which versions become visible sequentially according to SI, which we call *visibility order*. Every peer maintains the list of visible transactions in a *visibility log*.

The transaction then executes in isolation against the local cache. Its dependencies are the union of the state vector, the node's previous transactions, and the transactions in the node's visibility log.

Within a peer group, two different variants of commit exist. In the first, the node submits the transaction to EPaxos in the critical path of commit. This has the effect of ordering the commitment of conflicting transactions within the peer group, possibly leading to aborts; non-conflicting transactions commit in parallel. This variant maintains Parallel Snapshot Isolation (PSI) within a group [47], ensuring that the group behaves as an SI zone.

The second variant follows a similar approach to Section 3.7. It assumes that transactions never conflict. The transaction commits locally as soon as it reaches the commit statement, and a new transaction can follow immediately. The transaction is then submitted to EPaxos in the background.

Committed transactions become visible in the order assigned by EPaxos. A sync point sends visible transactions to the connected DC according to the visibility order, where they get assigned a commit timestamp.

5.2 Migration between peer groups

Just as a node can migrate between DCs (Section 3.8), it may migrate between peer groups. Similar consistency issues occur here. In this case, the base version of cached objects on the migrating node must be compatible with that of the new group. If the client is not missing any dependencies, or can retrieve them, then migration is seamless.

However, if the client is missing dependencies and the new peer group is offline, migration cannot succeed. If the client waits, its pending commits remain logged until the communication problem is fixed and they can be merged into the DC. In the meantime, the client might start a session with the new group, but its pending updates in the old session

become invisible. Alternatively, the client might attempt to migrate again.

5.3 Collaboration groups

The mechanisms related to collaboration groups are trust management and versioning.

Messages are protected using symmetric cryptography. The authentication service provides a client with a session key per shared object, to decrypt data and sign updates. This ensures that only legitimate clients can read an object. The key remains valid through disconnection and reconnection.

To keep out untrusted or unwanted updates, we leverage the separation between state and visibility, previously discussed in Section 3. Recall that an update is visible only if it satisfies the TCC+ invariants. In addition, it is visible only if it satisfies collaboration constraints.

To manage trust, the security administrator sets ACL. Furthermore, a collaboration group can, for instance, restrict visibility to include only versions produced within the group. An update that does not satisfy the corresponding ACL or group constraints remains invisible, and transitively the updates that depend upon it. Thus, security policies and groups can evolve dynamically. Technically, this violates the monotonicity invariant, but in a very restricted manner. The store remains TCC+, but security and group constraints expose only a variable-size window thereof.

6 System API and implementation

The Colony middleware is designed to provide a simple API for developing and deploying collaborative applications. This section presents its implementation and programming interface. The code is open-source and available on Gitlab [55].

6.1 API and programming model

An application node connects to a session manager (currently implemented in the core cloud), which authenticates the node. With the session opened, the node may join a collaboration or peer group, and run transactions accessing database objects. The node is notified of group change events (e.g., a new peer joins).

The database stores CRDT objects, such as counters, registers, sets, maps, or sequence datatypes. An object is stored in a namespace called a bucket. Opening a bucket caches it in the node; optional parameters can specify cache policies (e.g., LRU, writeback, etc.). The application can subscribe to an object's update events, in order to implement reactive programming patterns. A transaction is atomic (all-or-nothing) against multiple updates, and reads a TCC+-consistent snapshot of its opened buckets. Colony supports both interactive and batch transactions.

The example in Figure 3 illustrates the API. This application opens a session (Line 1). Then, it creates and increments a CRDT counter object (Lines 2–3). Then it connects to a peer group (line 4), and updates the grow-only map (gmap) "myMap" in a transaction (lines 5–8). This map contains references to a register object (key "a") and a set object (key "e"). The counter update and the commit are both asynchronous (Lines 7 and 9), returning a promise. At line 11, the client waits for the promise, and displays the content of the set.

```
1  let dc_con = dc.connect(CONF.dbURI, CONF.credentials);
2  let cnt = dc_con.counter("myCounter");
3  dc_con.update(cnt.increment(3))
4  let peer = pop.connect(CONF.popServers, CONF.credentials)
5  let tx = await peer.startTransaction()
6  let map = tx.gmap("myMap");
7  tx.update([ map.register("a").assign(42),
8              map.set("e").addAll([1, 2, 3, 4]) ])
9  tx.commit().then(
10   console.log(
11     await peer.gmap("myMap").set("e").read() ) )
```

Figure 3. Example Colony program.

6.2 Communication protocol

Edge nodes communicate over WebRTC. Opening a client session occurs in the signalling phase of WebRTC and currently relies on a server in the core cloud, to simplify authentication and trust management. The session provides the networking information required to communicate with the system, i.e., the IP addresses and ports of nearby peers, and the keys required to establish secure point-to-point connections with them. [8] To migrate to a different peer group, the node relies again on the session server.

6.3 Storage

Cloud nodes (DCs and PoPs) have secondary storage and persist their data to it. They also cache data in memory for performance. Data in a DC is sharded by consistent hashing across multiple server machines, leveraging `riak_core` [25].

We do not assume that a far-edge node has disk, and store data in browser memory. When a disconnected client reconnects again, it repopulates its cache, either from its peer group's content sharing network, or from its connected DC.

6.4 Security

The authentication keys received from the session server serve to encrypt communication between nodes, using symmetric encryption. Thus, only authenticated clients are able to observe and update objects. End-to-end encryption and decentralised authentification [26] is left for future work.

A system administrator can set a security policy with the help of access-control lists (ACLs). An ACL is a tuple from the set objects × users × permissions. It defines that a given user

[8] The first connection is established via STUN. If this fails (due to a firewall or NAT), Colony falls back to using TURN [61].

is granted access to some object and the operation she is allowed to execute on that object. Right inheritance (RI) is modelled using two forests, atop objects and users. If user u inherits from user v, then u holds the same ACL as v. Similarly, if an object x inherits from some object y, then any ACL granted on y also holds for x. Checking an ACL evaluates a first-order logic predicate over the RI and ACL relations following the above logic. For instance, (C1) $(\text{book}, \text{Alice}, \text{own}) \in ACL$, or the more complex
(C2) $(\text{book}, \text{shelf}) \in RI \wedge (\text{shelf}, \text{Bob}, \text{read}) \in ACL$ specify respectively that Alice owns a book and that this book is on a shelf readable by Bob.

ACL check must respect the order in which clients modify both data and the security policy, to avoid unexpected behaviour. More precisely, the system must ensure [36] that: *(i)* ACLs are applied in the order they were issued, and *(ii)* ACL checks are evaluated on a fresh copy of data and metadata. If data and security metadata are mutually consistent according to TCC+, the first constraint is trivially satisfied.

Let us use an example to illustrate the problem with the second constraint. Consider predicate C2, and assume that Alice, Bob and Carl share the bookshelf. Suppose Alice removes a book from the shelf on her node, while Bob makes the shelf readable by everyone. The two are concurrent from the causality perspective, and thus Carl may observe them in any order. However, by the second constraint, if Bob's update occurs later in real time, then Carl must never see Alice's book on the shelf. If Bob's node is disconnected or slow to transmit, this requirement is violated.

Colony alleviates this problem as follows. First, object versions are visible according to the local copy of the *ACL* and *RI* relations. Second, it defers ACL checks to after commit. A committed transaction that fails an ACL check is not visible. In the above example, Alice's book may appear briefly on Carl's node; but as soon as Bob's update is delivered, it will disappear.

7 Experimental evaluation

This section presents an empirical evaluation of Colony. We first demonstrate the implementation of a realistic collaborative application atop the middleware. With this as our main benchmark, we then evaluate the platform experimentally, comparing it to a classical client-server approach in the cloud, and to a simple caching approach. We consider both the online and the offline case. In the former, we evaluate transaction throughput vs. response time, and behaviour under load. In the offline case, we measure reconnection time, i.e., the time it takes for disconnected clients to be synchronised again. We also evaluate the performance benefit of peer groups. Finally, we study migration in mobile setups, measuring the time to return to normal performance.

7.1 ColonyChat benchmark application

Overview. ColonyChat emulates a team collaboration application modelled after the Slack and Mattermost communication platforms [33, 46]. It consists of ∼1500 lines of Typescript. ColonyChat represents its three main entities, users, workspaces and bots with the help of CRDT objects. In detail, a *user* has a profile, a list of events, a set of friends, and a set of workspaces she is a member of. A *workspace* contains the users that collaborate through the application and a set of channels. It also maintains the status of the users within the workspace (e.g., owner, ordinary, invited, or deleted). A *channel* holds a description, and the list of messages posted to it. A *bot* is a special kind of user. It automatically triggers an action when it observes some event, or a specific message on a channel. For instance, a bot might monitor activities within a file-system tree, or display weather information. Bots play an important role in the benchmark, as they generate a large number of update transactions.

The TCC+ guarantees of Colony ensure that there are no ordering anomalies in the application. For instance, an answer is guaranteed to be visible in a chat after the corresponding question. Moreover, transactions are atomic, allowing maintaining invariants such as "a user is in a workspace if and only if the workspace is in the user's profile." Furthermore, within an SI zone such as a peer group, users observe updates in the same order, greatly simplifying collaboration.

Workload. The workload consists of a modified trace from a popular Mattermost server. The trace contains the actions of around 2,000 users spreads over 3 workspaces; each workspace holds 20 channels on average. A user can be in more than one workspace, and one of the workspaces contains 1,000 users. Around 10% of the users are bots that act randomly upon receiving a message on the channel, they have subscribed to. An action of a user follows a 90/10 read-/write ratio. A user refreshes its local copy of a channel every 5 transactions. The trace follows a Pareto distribution for the action, where 20% of the users execute 80% of operations. It contains 40 days of activity in total on the Mattermost server and exhibits a diurnal cycle. In the experiments, the trace is accelerated to execute in a few minutes only.

For each experiment, we indicate when users are scattered in peer groups, or directly connected to a remote DC. The experiments use the second variant of the peer group commit protocol, i.e., EPaxos is off the critical path of commitment (Section 5.1.4). The current version of our benchmark does not exercise transaction migration (Section 3.9); this will be added in future work. Each experiment is executed 10 times, and we report the average.

7.2 Experimental setup

We deploy each Colony component (edge client, cloud server, peer group, etc.) as a Docker container, on a set of dedicated servers in a cluster. Each server has two Intel Xeon Gold

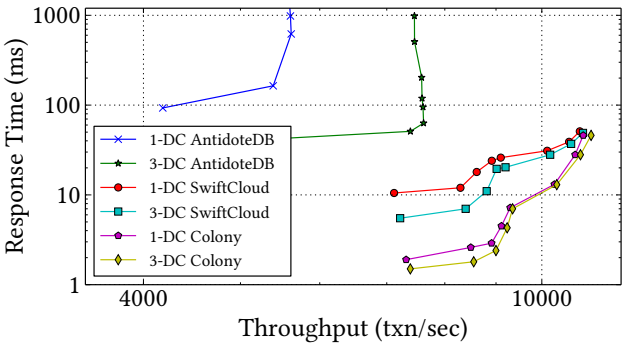


Figure 4. Performance of Colony.

CPUs, each with 16 cores per CPU, 128 GB of memory, and 2 TB of (spinning) hard disk. Nodes are all connected through 10 Gb/s network switches. A monitoring server, deployed on a separate container, captures the performance metrics.

We measure an average 0.15 ms network response time within the cluster. We use the Linux traffic shapping tool (`tc`) to simulate larger network response time, with a mean of 50 ms for mobile cellular data and 10 ms for carrier Ethernet. DCs are connected in a mesh using RabbitMQ sockets above TCP; peer groups are connected using WebRTC.

7.3 Response time and throughput

In this first experiment, we evaluate system performance when scaling up, increasing the number of clients until performance saturates. We compare three approaches. One emulates AntidoteDB [54], a classical geo-replicated approach, where a client does not have a local cache, and must contact the DC for each operation. Another emulates SwiftCloud [64], where clients have a local caches but do not form peer groups. Finally, the Colony label indicates a system with peer groups enabled. In each case, we evaluate a deployment with a single DC and one with three DCs.

Figure 4 reports throughput vs. response time. It uses a log-log scale; down and to the right is better. Load doubles from one point to the next, from 4 to 1024 clients. As expected, at the beginning of the curve, throughput improves and response time remains stable. At some point, throughput levels out and response time degrades, indicating saturation.

Observe that the Colony's response time is approximately 5 times better than Swiftcloud's, which itself performs one order of magnitude better than AntidoteDB (both for throughput and response time). This difference is explained by the caching policy. AntidoteDB does not have a client-side cache. In the SwiftCloud configuration, 90% of transactions hit the local cache. The hit rate reaches 95% in the shared cache of the Colony peer group configuration.

Adding more DCs spreads the load in the AntidoteDB configuration, improving the maximum throughput of the system, by 40% from a single to three DCs. However, adding more DCs does not improve response time, since clients need still to contact them for each operation. This is 8x slower than the SwiftCloud configuration. In contrast, the number of DCs has a minor impact in the SwiftCloud and Colony configurations.

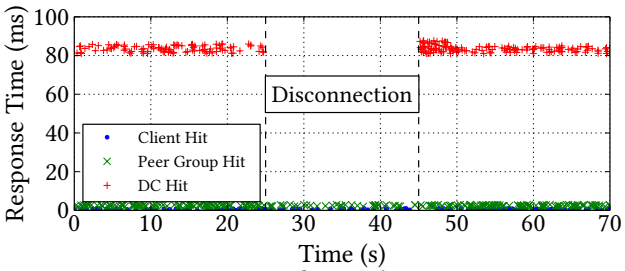


Figure 5. Impact of a DC disconnection

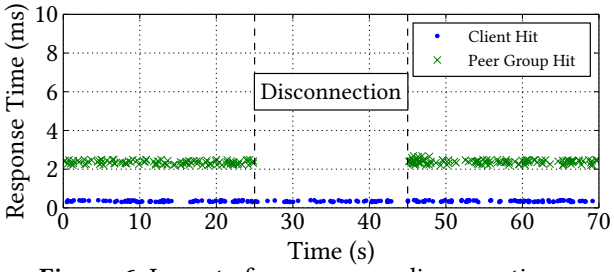


Figure 6. Impact of a peer group disconnection

7.3.1 Response time of offline collaboration.

We now evaluate how the response time varies under offline collaboration, or when the sync point of a group fails to connect to a DC. To this end, we use a single ColonyChat workspace that contains 36 users. We pack 12 of these users in a peer group, whereas the others remain independent. All users start with a warmed-up cache. Figure 5 shows the response time perceived at each user during the experiment. Each dot in this figure corresponds to a transaction.

In Figure 5, we observe that the response time for local cache hits is near zero (in blue). Users that belong to the same peer group benefit from an average 2.3 ms response time when data is fetched from the collaborative cache (in green). This raises to around 82 ms when the user needs to perform a remote read from the DC (in red).

Approximately 25 s after the start of the experiment, the peer group goes offline, and only collaborates on its shared interest set of objects. After this event, we observe that both the local and peer response time is unchanged: users in the group will not observe remote transactions due to the disconnection, but they continue their collaboration seamlessly. Around 45 s after the beginning of the experiment, the group is then reconnected to the DC. In Figure 5, we can observe a slight increase of the response time at the reconnection, yet it has minimal impact on performance.

In Figure 6, we consider the same workload but this time disconnect a user from its peer group. The disconnection

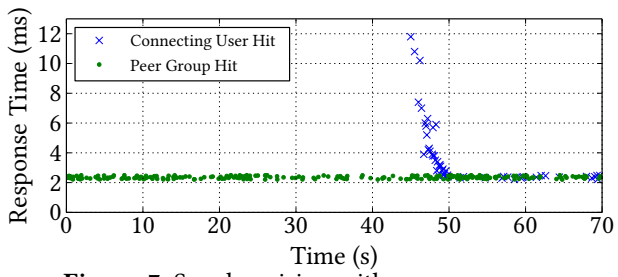


Figure 7. Synchronising with a peer group.

occurs after 25 s and the user reconnects 20 s later. In this figure, we may observe that the response time of the ColonyChat application is slightly impacted by the reconnection to the peer group. Upon reconnecting to the peer group, the user notices a slight increase (below the millisecond) in its transactions. This variation comes from the fact that the channels were updated with the new content published by the users in the peer group.

7.4 Migration effect on response time

Mobile clients, especially in location-based collaborative applications, like games, frequently switch from one peer group to another. Our last experiment studies the synchronisation time for a client to connect to a group when its cache is invalid. This experiment exercises both the cache refreshing mechanism of Colony and the collaborative cache in a peer group. The results are presented in Figure 7.

In this figure, 45s after the start of the experiment, a mobile client migrates and joins the peer group. The client has a completely invalid chat history. She thus needs to synchronise her cache before interacting with the peer group. Figure 7 plots the (average) response time observed by the connecting user (in blue), and the rest of the group (in green). As previously, each dot in the plot represents the response time of a transaction. In this figure, we can observe that the first transactions of the connecting user have a higher response time (below 12 ms). This performance degradation is way lower than the cost of reconnecting to a DC, and fetching data from it (as in Figure 5). Moreover, after only a few seconds, it returns to the normal and matches the perceived response time of the group users (in green).

8 Related work

Previous work on data in edge computing [22, 40, 53] focuses on streaming and content delivery. Sharing mutable state raises extra challenges, which are the focus of this paper.

To deliver fast response and offline support, applications may cache data at the client side, e.g., in the browser, as in News Feed [32], or Google Docs [48]. Mobile operation requires on-device replicas of data under weak consistency [42, 51], as in Bayou [53], Rover [22] or Coda [23]. Similarly, Cimbiosys supports decentralised Internet Services [40]. The COPS system introduces Causal Plus Consistency (CC+), strengthening causal consistency with strong convergence [30]. ChainReaction augments CC+ with transactional reads, thanks to a sequencer per DC, and executes a write only after the versions read by the client are stable in the DC. TCC+ [1] extends the guarantees of CC+ to transactions. This model is closely related to Parallel Snapshot Isolation (PSI) [47]. Whereas TCC+ supports concurrent updates and arbitrary CRDT types, PSI restricts concurrency to a single data type (the cset). Its SI zones, the DCs, are fixed. PSI supports global transactions that are strongly consistent, but this impacts availability and performance.

Hybrid consistency models combine different consistency guarantees [5, 13, 29]. In Lazy Replication, operations are CC by default, and optionally linearisable [27]. Unistore [7] supports causal and linearisable transactions over a geo-replicated store; for fault tolerance, the causal dependencies of a linearisable transaction must be stable before it commits. Fisheye Consistency [15] is a proximity-based hybrid model, such that close-by nodes are mutually strongly consistent, and consistency is weak between far-away nodes. Depot [31] and PRACTI [6] pioneer highly available caching at the edge, under CC with one vector clock entry per replica; this limits scalability. Depot targets Byzantine fault tolerance, but not transactions. Simba [37] enables the edge application to select among eventual, causal or serialisable consistency. PouchDB [39] is a client-side cache replica for a CouchDB server; it supports offline operation and detects conflicts, but does not merge them. SwiftCloud [64] introduces bounded-size vectors and migration. Legion [56] extends web applications with P2P interaction using CRDTs under CC. Colony extends the above designs with collaboration and peer groups, and seamless migration.

9 Conclusion

We presented the design, implementation and performance of Colony, a system that brings the strongest consistency guarantees (while bounding the cost of causality metadata) to applications at the edge. According to an edge-first design, edge applications enjoy data locality, fast response, and disconnected operation. Colony supports seamless migration of a device or a whole peer group. Furthermore, Colony supports collaboration, ensuring total-order consistency within an edge group, and relevant security guarantees.

Several aspects remain open for improvement. As an edge device has limited resources, applications with a large footprint would benefit from better caching heuristics and automatic transaction migration. Placing clients at different levels of the hierarchy, in particular in Content Delivery Network points of presence, might improve perceived response time even more. Extending peer-to-peer communication beyond edge groups would make the system less dependent on the cloud.

References

[1] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong semantics meets high availability and low latency. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*. Nara, Japan, 405–414. https://doi.org/10.1109/ICDCS.2016.98

[2] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça, and Victor Fonte. 2014. Scalable and Accurate Causality Tracking for Eventually Consistent Stores. In *Int. Conf. on Distr. Apps. and Interop. Sys. (DAIS) (lncs, Vol. 8460)*, Kostas Magoutis and Peter Pietzuch (Eds.). Int. Fed. for Info. Processing (IFIP), Springer-Verlag, Berlin, Germany, 67–81. https://doi.org/10.1007/978-3-662-43352-2_6

[3] Peter Alvaro, Neil Conway, Joe Hellerstein, and William Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Biennial Conf. on Innovative DataSystems Research (CIDR)*. Asilomar, CA, USA. http://www.cidrdb.org/cidr2011/

[4] Hagit Attiya, Faith Ellen, and Adam Morrison. 2017. Limitations of Highly-Available Eventually-Consistent Data Stores. *IEEE Trans. on Parallel and Dist. Sys. (TPDS)* 28, 1 (Jan. 2017), 141–155. https://doi.org/10.1109/TPDS.2016.2556669

[5] Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency back into Eventual Consistency. In *Euro. Conf. on Comp. Sys. (EuroSys)*. Bordeaux, France, 6:1–6:16. https://doi.org/10.1145/2741948.2741972 Indigo.

[6] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. 2006. PRACTI Replication. In *Networked Sys. Design and Implem. (NSDI)*. Usenix, Usenix, San Jose, CA, USA, 59–72. https://www.usenix.org/legacy/event/nsdi06/tech/belaramani.html

[7] Manuel Bravo, Alexey Gotsman, Borja de Régil, and Hengfeng Wei. 2021. UniStore: A fault-tolerant marriage of causal and strong consistency. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 923–937. https://www.usenix.org/conference/atc21/presentation/bravo

[8] Manuel Bravo, Luìs Rodrigues, and Peter Van Roy. 2017. Saturn: A Distributed Metadata Service for Causal Consistency. In *Euro. Conf. on Comp. Sys. (EuroSys)*. Assoc. for Computing Machinery, Assoc. for Computing Machinery, Belgrade, Serbia, 111–126. https://doi.org/10.1145/3064176.3064210

[9] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Symp. on Principles of Prog. Lang. (POPL)*. San Diego, CA, USA, 271–284. https://doi.org/10.1145/2535838.2535848

[10] W. Cai, A. Ng, and C. Sun. 2019. Design and Implementation of a Concurrency Benchmark Tool for Cloud Storage Systems. In *Int. Conf. on Computer-Supported Coop. Work (CSCW)*. IEEE Comp. Society, IEEE Comp. Society, Porto, Portugal, 386–391. https://doi.org/10.1109/CSCWD.2019.8791849

[11] Bernadette Charron-Bost. 1991. Concerning the size of logical clocks in distributed systems. *Inform. Process. Lett.* 39, 1 (July 1991), 11–16. https://doi.org/10.1016/0020-0190(91)90055-M

[12] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *Symp. on Reliable Dist. Sys. (SRDS)*. IEEE Comp. Society, Braga, Portugal, 173–184. https://doi.org/10.1109/SRDS.2013.26

[13] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. 1999. Eventually-Serializable Data Services. *Theoretical Computer Science* 220 (1999), 113–156. https://doi.org/10.1016/S0304-3975(98)00239-4 Special issue on Distributed Algorithms.

[14] C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*. University of Queensland, Australia, 55–66.

[15] R Friedman, M Raynal, and François Taïani. 2015. Fisheye Consistency: Keeping Data in Synch in a Georeplicated World. In *International Conference on NETworked sYStems (NETYS'2015) (Networked Systems : Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers, 9466)*. Springer International Publishing, Agadir, Morocco. https://doi.org/10.1007/978-3-319-26850-7_17

[16] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. https://doi.org/10.1145/564585.564601

[17] Alain Girault, Gregor Gössler, Rachid Guerraoui, Jad Hamza, and Dragos-Adrian Seredinschi. 2018. Monotonic Prefix Consistency in Distributed Systems. In *Formal Techniques for Distributed Objects, Components, and Systems (FORTE) (Lecture Notes in Comp. Sc., Vol. 10854)*, C. Baier and Caires L. (Eds.). Springer-Verlag. https://doi.org/10.1007/978-3-319-92612-4_3

[18] Google. [n.d.]. Fix common issues in Google Drive. https://support.google.com/drive/answer/2456903. Accessed: 2021-05-05.

[19] Google Drive Doc Editors Help, Community. [n.d.]. Lost doc going from offline to online. I looked in trash, recents, any ideas? https://support.google.com/docs/thread/10026411/lost-doc-going-from-offline-to-online-i-looked-in-trash-recents-any-ideas?hl=en. Accessed: 2021-05-05.

[20] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In *Symp. on Principles of Prog. Lang. (POPL)*. Assoc. for Computing Machinery, St. Petersburg, FL, USA, 371–384. https://doi.org/10.1145/2837614.2837625

[21] Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. In *Symp. on Principles of Prog. Lang. (POPL) (Proc. ACM Program. Lang., Vol. 3)*. Assoc. for Computing Machinery, Cascais, Portugal, 74:1–74:32. https://doi.org/10.1145/3290387

[22] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. 1995. Rover: A Toolkit for Mobile Information Access. *Operating Systems Review* 29, 5 (Dec. 1995), 156–171. https://doi.org/10.1145/224057.224069

[23] James J. Kistler and M. Satyanarayanan. 1992. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)* 10, 5 (Feb. 1992), 3–25. http://www.acm.org/pubs/contents/journals/tocs/1992-10

[24] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in spite of the Cloud. In *Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!* Assoc. for Computing Machinery Special Interest Group on Pg. Lang. (SIGPLAN), Assoc. for Computing Machinery, Athens, Greece, 154–178. https://doi.org/10.1145/3359591.3359737

[25] Rusty Klophaus. 2010. Riak Core: Building Distributed Applications without Shared State. In *Commercial Users of Functional Programming (CUFP)* (Baltimore, Maryland) *(ICFP'10)*. Assoc. for Computing Machinery, Baltimore, Maryland, USA, Article 14, 1 pages. https://doi.org/10.1145/1900160.1900176

[26] Stephan A. Kollmann, Martin Kleppmann, and Alastair R. Beresford. 2019. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. In *Privacy Enhancing Technologies (PoPETS)*, Vol. 2019. Stockholm, Sweden. https://martin.kleppmann.com/2019/07/16/snapdoc-authenticated-snapshots.html

[27] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing High Availability Using Lazy Replication. *Trans. on Computer Systems* 10, 4 (Nov. 1992), 360–391. http://dx.doi.org/10.1145/138873.138877

[28] João Leitão. 2019. Pokémon Go in-the-field anomaly. (Sept. 2019). Private communication.

[29] 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 *Symp. on Op. Sys. Design and Implementation (OSDI)*. Hollywood, CA, USA, 265–278. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li

[30] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*. Assoc. for Computing Machinery, Cascais, Portugal, 401–416. https://doi.org/10.1145/2043556.2043593

[31] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. 2011. Depot: Cloud Storage with Minimal Trust. *Trans. on Computer Systems* 29, 4 (Dec. 2011), 12:1–12:38. https://doi.org/10.1145/2063509.2063512

[32] Chris Marra and Alex Sourov. 2015. Continuing to build News Feed for all types of connections. https://engineering.fb.com/2015/12/09/android/continuing-to-build-news-feed-for-all-types-of-connections/.

[33] Mattermost Inc. 2015. Mattermost. https://mattermost.com.

[34] Friedmann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Int. W. on Parallel and Distributed Algorithms*. Elsevier Science Publishers B.V. (North-Holland), 215–226.

[35] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Symp. on Op. Sys. Principles (SOSP) (SOSP '13)*. Assoc. for Computing Machinery, Farminton, PA, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[36] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christopher D. Richards, and Mengzhi Wang. 2019. Zanzibar: Google's Consistent, Global Authorization System. In *Usenix Annual Tech. Conf.* Usenix, Renton, WA, USA. https://www.usenix.org/conference/atc19/presentation/pang

[37] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V. Madhyastha, and Cristian Ungureanu. 2015. Simba: Tunable End-to-End Data Consistency for Mobile Apps. In *Euro. Conf. on*

Comp. Sys. (EuroSys). Assoc. for Computing Machinery, Bordeaux, France, 7:1–7:16. https://doi.org/10.1145/2741948.2741974

[38] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. 1997. Flexible Update Propagation for Weakly Consistent Replication. In *Symp. on Op. Sys. Principles (SOSP)*. ACM SIGOPS, Saint Malo, 288–301. http://doi.acm.org/10.1145/268998.266711

[39] PouchDB. [n.d.]. PouchDB website. https://pouchdb.com/.

[40] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. 2009. Cimbiosys: A platform for content-based partial replication. In *Networked Sys. Design and Implem. (NSDI)*. Usenix, Usenix, Boston, MA, USA, 261–276. https://www.usenix.org/legacy/event/nsdi09/tech/full_papers/ramasubramanian/ramasubramanian.pdf

[41] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Symp. on Reliable Dist. Sys. (SRDS)*. IEEE Comp. Society, Braga, Portugal, 163–172. https://doi.org/10.1109/SRDS.2013.25

[42] Yasushi Saito and Marc Shapiro. 2005. Optimistic Replication. *Comput. Surveys* 37, 1 (March 2005), 42–81. https://doi.org/10.1145/1057977.1057980

[43] Alexander Schäfer, Gerd Reis, and Didier Stricker. 2021. A Survey on Synchronous Augmented, Virtual and Mixed Reality Remote Collaboration Systems. *arXiv preprint arXiv:2102.05998* (2021).

[44] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS) (Lecture Notes in Comp. Sc., Vol. 6976)*, Xavier Défago, Franck Petit, and V. Villain (Eds.). Springer-Verlag, Grenoble, France, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[45] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. 2016. Consistency in 3D. In *Int. Conf. on Concurrency Theory (CONCUR) (Leibniz Int. Proc. in Informatics (LIPICS), Vol. 59)*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, Québec, Québec, Canada, 3:1–3:14. https://doi.org/10.4230/LIPIcs.CONCUR.2016.3

[46] Slack Technologies. 2013. Slack. https://slack.com.

[47] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*. Assoc. for Computing Machinery, Cascais, Portugal, 385–400. https://doi.org/10.1145/2043556.2043592

[48] support.google.com. [n.d.]. Work on Google Docs, Sheets, & Slides offline. https://support.google.com/docs/answer/6388102.

[49] Dean Takahashi. 2018. Onshape lets engineers collaborate on 3D designs with Magic Leap's AR glasses. https://venturebeat.com/2018/10/13/onshape-lets-engineers-collaborate-on-3d-designs-with-magic-leaps-ar-glasses/.

[50] Vinh Tao, Marc Shapiro, and Vianney Rancurel. 2015. Merging Semantics for Conflict Updates in Geo-Distributed File Systems. In *ACM Int. Systems and Storage Conf. (Systor)*. Assoc. for Computing Machinery, Haifa, Israel, 10.1–10.12. https://doi.org/10.1145/2757667.2757683

[51] Doug Terry. 2013. Replicated Data Consistency Explained Through Baseball. *Commun. ACM* 56, 12 (Dec. 2013), 82–89. https://doi.org/10.1145/2500500

[52] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*. Austin, Texas, USA, 140–149.

[53] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Symp. on Op. Sys. Principles (SOSP)*. ACM SIGOPS, ACM Press, Copper Mountain, CO, USA, 172–182. http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf

[54] The SyncFree Consortium. [n.d.]. AntidoteDB: A planet scale, highly available, transactional database. Website http://antidoteDB.eu/.

[55] Ilyas Toumlilt. [n.d.]. Colony source code. https://gitlab.lip6.fr/itoumlilt/colony.

[56] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *Int. Conf. on World Wide Web (WWW) (WWW '17)*. International World Wide Web Conferences Steering Committee, Perth, Australia, 283–292. https://doi.org/10.1145/3038912.3052673

[57] Albert van der Linde, João Leitão, and Nuno Preguiça. 2020. Practical Client-side Replication: Weak Consistency Semantics for Insecure Settings. *Proc. VLDB Endow.* 13, 11 (July 2020), 2590–2605. https://doi.org/10.14778/3407790.3407847

[58] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *Comput. Surveys* 49, 1 (jul 2016), 19:1–19:34. https://doi.org/10.1145/2926965

[59] Peng Wang, Shusheng Zhang, Mark Billinghurst, Xiaoliang Bai, Weiping He, Shuxia Wang, Mengmeng Sun, and Xu Zhang. 2019. A comprehensive survey of AR/MR-based co-design in manufacturing. *Engineering with Computers* (2019), 1–24.

[60] Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2016. Access Control for Weakly Consistent Replicated Information Systems. In *Int. W. on Security and Trust Management (Lecture Notes in Comp. Sc., Vol. 9871)*, Gilles Barthe, Evangelos P. Markatos, and Pierangela Samarati (Eds.). Springer-Verlag, Heraklion, Crete, Greece, 82–97. https://doi.org/10.1007/978-3-319-46598-2_6 ACGreGate.

[61] Dan Wing, Philip Matthews, Rohan Mahy, and Jonathan Rosenberg. 2008. Session traversal utilities for NAT (STUN). *RFC5389, October* (2008).

[62] Ted Wobber, Thomas L. Rodeheffer, and Douglas B. Terry. 2010. Policy-based access control for weakly consistent replication. In *Euro. Conf. on Comp. Sys. (EuroSys)*, Christine Morin and Gilles Muller (Eds.). Assoc. for Computing Machinery, Paris, France, 293–306. https://doi.org/10.1145/1755913.1755943

[63] www.reddit.com/r/pokemongo. [n.d.]. Incubator duplication glitch?? https://www.reddit.com/r/pokemongo/comments/lxzsf1/incubator_duplication_glitch/. Accessed: 2021-05-05.

[64] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. In *Int. Conf. on Middleware (MIDDLEWARE)*. ACM/IFIP/Usenix, Vancouver, BC, Canada, 75–87. https://doi.org/10.1145/2814576.2814733