On Mixing Eventual and Strong Consistency: Acute Cloud Types

Maciej Kokociński, Tadeusz Kobus, and Paweł T. Wojciechowski

Abstract—In this article we study the properties of distributed systems that mix eventual and strong consistency. We formalize such systems through *acute cloud types* (ACTs), abstractions similar to conflict-free replicated data types (CRDTs), which by default work in a highly available, eventually consistent fashion, but which also feature strongly consistent operations for tasks which require global agreement. Unlike other mixed-consistency solutions, ACTs can rely on efficient quorum-based protocols, such as Paxos. Hence, ACTs gracefully tolerate machine and network failures also for the strongly consistent operations. We formally study ACTs and demonstrate phenomena which are neither present in purely eventually consistent nor strongly consistent systems. In particular, we identify *temporary operation reordering*, which implies interim disagreement between replicas on the relative order in which the client requests were executed. When not handled carefully, this phenomenon may lead to undesired anomalies, including circular causality. We prove an impossibility result which states that temporary operation reordering is unavoidable in mixed-consistency systems with sufficiently complex semantics. Our result is startling, because it shows that apparent *strengthening* of the semantics of a system (by introducing strongly consistent operations to an eventually consistent system) results in the weakening of the guarantees on the eventually consistent operations.

Index Terms—Eventual consistency, mixed consistency, fault-tolerance, acute cloud types, ACT

1 Introduction

The massive scalability and high availability of the complex (geo-replicated) distributed systems that power today's Internet often hinges on the use of eventually consistent data stores. These systems extensively employ specialized data structures, e.g., last-write-wins registers (LWW-registers), multi-value registers (MVRs), observed-remove sets (OR-sets) or other conflict-free replicated data types (CRDTs) [1], [2], [3]. These data structures are replicated on multiple machines (*replicas*) and can be read or modified independently on each replica without prior synchronization with other replicas. It means that replicas can promptly respond to the clients. The communication between the replicas happens solely using a gossip protocol. By design replicas are guaranteed to be able to converge to a single state, automatically resolving any inconsistencies between them.

Unfortunately, the semantics of such data structures are very limited. To provide high availability, low response times and eventual state convergence, these data structures require either that all operations commute, or that there exist commutative, associative, and idempotent procedures for merging replica states. This is why these mechanisms are not suitable for all use cases. For example, consider a simple non-negative integer counter. The addition operation can be trivially implemented in a conflict-free manner, as the addition operations are commutative. However, the subtraction operation requires global agreement to ensure that the value of the counter never drops below 0. In a similar way, in an auction system, concurrent bids can be considered independent operations and thus their execution does not need to be synchronized. However, the operation that closes the auction requires solving distributed consensus to select the single winning bid [4]. Due to the inherent shortcomings of CRDTs, recently there have been several attempts in the industry (e.g., [5], [6], [7], [8]) to enrich the semantics of the eventually consistent systems by allowing some operations to be performed with stronger consistency guarantees or by introducing (quasi) transactional support. Unfortunately, these attempts lack clearly stated semantics. For example, in Apache Cassandra using the *light weight transactions* on data that are accessed at the same time in the regular, eventually consistent fashion leads to undefined behaviour [9].

In this article we introduce *acute cloud types* (ACTs), a family of specialized mixed-consistency data structures designed primarily for high availability and low latency, but that also seamlessly integrate on-demand strongly consistent semantics. ACTs feature two kinds of operations: *weak operations*, targeted for unconstrained scalability and low response times (as operations in CRDTs), and *strong operations*, used when eventually consistent guarantees are insufficient. Strong operations require consensus-based inter-replica synchronization prior to execution.

Weak operations are guaranteed to progress, and are handled in such a way that the replicas eventually converge to the same state within each network partition, even when strongly consistent operations cannot complete due to network and process failures. On the other hand, strong operations can provide guarantees even as strong as linearizability [10] with respect to the already completed strong operations and a precisely defined subset of completed weak operations. Crucially,

• *The authors are with the Institute of Computing Science, Poznan University of Technology, 60-965 Poznań, Poland. E-mail: {Maciej.Kokocinski, Tadeusz.Kobus, Pawel.T.Wojciechowski}@cs.put.edu.pl.*

Manuscript received 27 Aug. 2020; revised 23 Mar. 2021; accepted 2 May 2021. Date of publication 17 June 2021; date of current version 26 Oct. 2021. (Corresponding author: Maciej Kokociński.) Recommended for acceptance by V. Cardellini. Digital Object Identifier no. 10.1109/TPDS.2021.3090318

strong operations are *non-blocking*: they can leverage efficient, quorum-based synchronization protocols, such as Paxos [11], and thus gracefully tolerate machine and network failures. Both weak and strong operations can be arbitrarily complex, but they must be deterministic.

Compared to other mixed-consistency solutions, our approach is more robust. Most notably, unlike classic *cloud types* [12] and *global sequence protocol* (GSP) [13], ACTs are *symmetrical* in the sense that they do not assume the existence of a server or servers that mediate all communication between remote replicas. Instead, ACTs utilize peer-to-peer communication model. This has several advantages: a failure of a replica or a group of replicas cannot impede the ability of other ACT replicas to execute weak operations and propagate the resulting updates. Also, ACTs can better tolerate network splits by allowing the replicas in all partitions to execute weak operations and exchange resulting updates. Furthermore, unlike the RedBlue consistency model [14] and similar approaches (e.g., [15], [16], [17]), ACTs support consistency guarantees weaker than causal consistency, so account for a wider range of systems. Causal consistency is known to be costly to achieve in practice [18], and is not always needed [19]. Crucially, ACTs do not require all replicas to be operational in order for the strong operations to complete, contrary to the approaches mentioned above. This latter trait has been fundamental to the design of ACTs.

In any run of an ACT, logically, there always exists a single global order S of all operations. Therefore, system traces can be reasoned about in terms of serial execution, which is the hallmark of strong consistency [10], [20], [21], as well as various weaker models, e.g., [22], [23], [24], [25]. During execution, strong operations are guaranteed to observe a prefix of S up to their position in S. A weak operation may observe a serialization S' of operations that diverges from S, but only by a finite number of elements. Thus weak and strong operations are interconnected in a non-trivial way, which intuitively ensures *write stabilization*: once a strong operation, during its execution, observes some weak operations op_i, op_j in that order, all subsequent strong operations, and eventually all weak operations, will also observe op_i, op_j in that order. Write stabilization allows ACTs to overcome limitations of models such as RedBlue consistency in which the effects of a weak operation could never be deemed final. It is so even though weak operations never have to directly synchronize with strong operations (e.g., by blocking on the completion of strong operations).

We propose a framework that enables formal reasoning about ACTs and their guarantees. We express the dependencies between operations through the *visibility* and *arbitration* relations, similarly as in [26], but we allow each operation to observe the arbitration in a temporarily inconsistent (but eventually convergent) form. In order to capture the unique properties of ACTs and write stabilization in particular, we define a novel correctness condition called *fluctuating eventual consistency* (FEC) that is strictly weaker than Burckhardt's Basic Eventual Consistency (BEC) [27].

By formally specifying ACTs, we uncovered several interesting phenomena unique to mixed-consistency systems (they are never exhibited by popular NoSQL systems, which only guarantee eventual consistency, nor by strongly consistent solutions). Crucially, some ACTs exhibit a phenomenon that we call *temporary operation reordering*, which happens when replicas temporarily disagree on the relative order in which the requests (modelled as operations) submitted to the system were executed. When not handled carefully, temporary operation reordering may lead to all kinds of undesired situations, e.g., circular causality among responses observed by the clients. As we formally prove, temporary operation reordering is not present in all ACTs but in some cases cannot be avoided. This impossibility result is startling, because it shows that apparent *strengthening* of the semantics of a system (by introducing strong operations to an eventually-consistent system) results in the weakening of the guarantees on the eventually-consistent operations.

In order to illustrate our concepts and analysis, we present an ACT for a non-negative counter and also revisit Bayou [23], a seminal, always available, eventually consistent data store. Bayou combines timestamp-based eventual consistency [28] and serializability [20] by speculatively executing transactions submitted by clients and having a primary replica to periodically *stabilize* the transactions (establish the final transaction execution order). We show how Bayou can be improved to form a general-purpose ACT.

1.1 Contribution Summary

1) We define *acute cloud types*, a family of specialized mixed-consistency data structures designed primarily for high availability and low latency, which also seamlessly integrate on-demand strongly consistent semantics achieved through quorum-based consensus protocols. Weak and strong operations in ACTs are interconnected in a non-trivial way, which intuitively ensures *write stabilization*.
2) We identify a range of traits unique to some ACTs. Most importantly, we define *temporary operation reordering*, a situation in which there is an interim disagreement between replicas on the relative order in which client requests were executed.
3) We propose a framework that enables formal reasoning about ACTs and their guarantees. In particular, our framework allows us to formalize temporary operation reordering and propose a correctness condition, called *fluctuating eventual consistency*, which adequately captures the guarantees provided by ACTs that exhibit this phenomenon.
4) We use our framework to prove a number of formal results regarding ACTs. As our main contribution, we show an impossibility result that states that temporary operation reordering, while not pertinent to all ACTs, in some ACTs cannot be avoided.
5) We revisit the seminal Bayou system, study its consistency guarantees, and show how it can be improved to form a general-purpose ACT.

1.2 Article Structure

The article is organized as follows. In Section 2 we explain ACTs through examples: an acute non-negative counter and an adaptation of Bayou that forms a general-purpose ACT. We formally define ACTs in Section 3, and introduce

the formal framework for reasoning about their correctness in Section 4. In Section 5 we define FEC, our new correctness criterion and prove the correctness of our example ACTs. Next, in Section 6, we give our impossibility result. We discuss related work in Section 7, and conclude in Section 8.

A brief announcement of this article appeared in [29].

2 Acute Cloud Types by Example

2.1 Acute Non-Negative Counter

As mentioned in Section 1, a non-negative integer counter cannot be implemented as a CRDT because the subtraction operation requires global coordination to ensure that the value of the counter never drops below 0. In Algorithm 1 we present an *acute non-negative integer counter* (ANNC), a simple ACT implementing such a counter. The add (line 5) and get (line 31) operations are weak and thus guarantee low response times, whereas subtract (line 11) is a strong operation to ensure the semantics of a non-negative counter. The crux of ANNC lies in using two complementary protocols for exchanging updates (a gossip one and one that establishes the ultimate operation serialization), and calculating the state of the counter by liberally counting add operations and conservatively counting the subtract operations.

To track the execution of weak and strong operations, each ANNC replica maintains three variables (line 2): one for subtraction operations (*strongSub*) and two for the addition operations (*weakAdd* and *strongAdd*). The replicas exchange the information about new ADD requests (weak updating operations) using a gossip protocol (modelled using *reliable broadcast, RB* [30]) as well as a protocol that involves inter-replica synchronization (modelled using *total order broadcast, TOB* [31], which can be efficiently implemented using quorum-based protocols, such as Paxos [11]; lines 9-10). The subtract operation, which does not commute unlike the add operation, solely uses TOB. Upon receipt of a TOB-cast SUBTRACT message, the subtract operation completes successfully only if we are certain that the value of the counter does not drop below 0, i.e., when the aggregated value of all confirmed addition operations (*strongAdd*) is greater or equal to the aggregated value of all subtract operations (*strongSub*) increased by *value* (lines 25-27).

We ensure that on any replica and for any ADD request r, the RB-deliver(r) event always happens before the TOB-deliver(r) event (lines 21–22). This way $weakAdd \geq strongAdd$. Hence, we solely use *weakAdd* as the approximation of the total value added to ANNC when calculating the return value for the get operations.

Using a gossip protocol allows us to achieve propagation of weak updating operations within network partitions, when synchronization which requires solving distributed consensus is not possible. On the other hand, when solving distributed consensus is possible, replicas can agree on the final order in which operations will be visible. This way weak operations add and get are highly available, i.e., they always execute in a constant number of steps and do not depend on waiting on communication with other replicas. Crucially, the return value of the get operation always reflects all the add operations performed locally and, eventually, all add operations performed within the network partition to which the replica belongs, if such a partition exists. On the other hand, the strong subtract operation is applied only if the replicas agree that it is safe to do so.

Algorithm 1. Acute Non-Negative Counter (ANNC) for Replica R_i.

```
 1: struct Req(type : {ADD, SUBTRACT}, value : int, id :
                pair⟨int, int⟩)
 2: var weakAdd, strongAdd, strongSub, currEventNo : int
 3: var reqsAwaitingResp : set⟨pair⟨int, int⟩⟩
 4: var rbDeliveredAdds : set⟨pair⟨int, int⟩⟩
 5: upon invoke add(value : int)                    // weak operation
 6:   currEventNo = currEventNo + 1
 7:   weakAdd = weakAdd + value
 8:   r = Req(ADD, value, (i, currEventNo))
 9:   RB-cast(r)
10:   TOB-cast(r)
11: upon invoke subtract (value : int)              // strong operation
12:   currEventNo = currEventNo + 1
13:   r = Req(SUBTRACT, value, (i, currEventNo))
14:   TOB-cast(r)
15:   reqsAwaitingResp = reqsAwaitingResp ∪ {r.id}
16: upon RB-deliver(r : Req(ADD, value, id))
17:   if r.id.first ≠ i ∧ r.id ∈ rbDeliveredAdds then
18:     rbDeliveredAdds = rbDeliveredAdds ∪ {r.id}
19:     weakAdd = weakAdd + value
20: upon TOB-deliver(r : Req(ADD, value, id))
21:   if r.id ∉ rbDeliveredAdds then
22:     trigger RB-deliver(r) // RB-deliver always before TOB-deliver
23:   strongAdd = strongAdd + value
24: upon TOB-deliver(r : Req(SUBTRACT, value, id))
25:   var res = strongAdd ≥ strongSub + value
26:   if res then
27:     strongSub = strongSub + value
28:   if id ∈ reqsAwaitingResp then
29:     reqsAwaitingResp = reqsAwaitingResp \ {id}
30:     return res to client
31: upon invoke get()                               // read-only, weak operation
32:   return weakAdd − strongSub to client
```

ANNC guarantees a property which is a conjunction of *basic eventual consistency* (BEC) [26], [27] for weak operations (add and get) and *linearizability* (Lin) [10] for strong operations (subtract). We formalize BEC and Lin in Sections 5.2 and 5.5, and prove the correctness of ANNC in Section 5.6.

2.2 Bayou

Bayou was an experimental system, so was never optimized for performance. However, due to its unique approach to speculative execution of transactions and their later *stabilization* (establishing the final transaction execution order by a primary replica), examining Bayou allows us to discuss various problematic phenomena that stem from having both weak and strong semantics in a single system. We improve Bayou to form a general-purpose, albeit not performance-optimized ACT.

2.2.1 Protocol Overview

Below we give a high-level description of the Bayou protocol. An interested reader may find a detailed description of Bayou (together with a pseudocode) in Appendix A.1, which can be

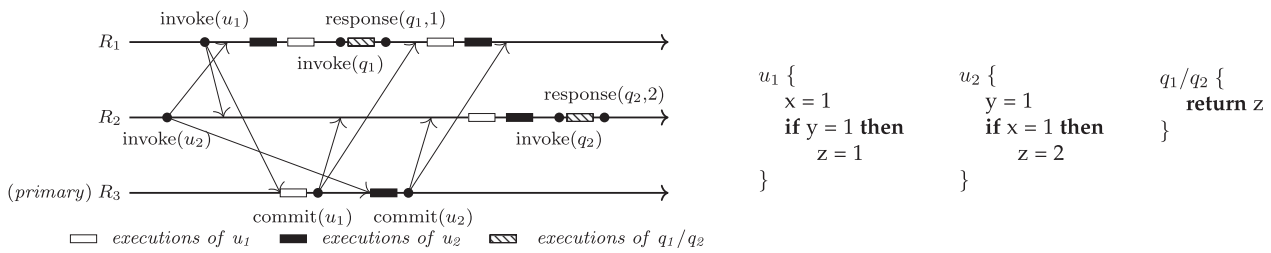

Fig. 1. Example execution of Bayou showing temporary operation reordering and circular causality.

found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2021.3090318. In order to make our analysis more general, we abstract certain aspects of the original protocol. Crucially, we allow clients to submit to Bayou replicas deterministic, arbitrarily complex (also as complex as, e.g., SQL transactions) operations that can provide the clients with a return value. Each operation is either *weak* or *strong*, similarly to operations in ANNC.

In Bayou, each replica speculatively total-orders all client operations, without prior agreement with other replicas, using a simple timestamp-based mechanism (a replica assigns a timestamp to an operation upon its submission). The requests (operations together with their timestamps) are disseminated to all replicas using a gossip protocol and each replica independently executes them sequentially according to their timestamps. When a replica delivers a request r with a timestamp lower than some already executed requests, the higher-timestamp requests are rolled-back and reexecuted after r. This way a single total order, consistent with operation timestamps, is always maintained by all replicas.

This approach has two major downsides. The first one concerns the performance: every time a replica receives a request with a relatively low timestamp (compared to the requests executed most recently), to maintain the correct execution order, many requests need to be rolled back and reexecuted. The second downside is related to the guarantees provided: a client that submitted an operation op and already received a response can never be sure that there will be no other operation op' with a lower timestamp than op, which will eventually cause op to be reexecuted, thus producing possibly a different return value.

To mitigate the two above problems, one of the replicas, called the *primary*, periodically commits a growing prefix of already executed operations, i.e., it decides to never rollback them again and broadcasts this decision to other replicas. Thus, it establishes the *final operation execution order* (also called the *committed order*). This order may occasionally differ from the timestamp order, e.g., when a message sent to the primary is delayed. Replicas always honour the order established by the primary, which may force them to rollback and reexecute some operations. However, once an operation is executed according to the committed order on a replica R, it will never be rolled back and reexecuted again on R (we then say that the operation is *stable* on R). Ultimately, all operations (weak or strong) are committed and become stable. However, since weak operations return results before this occurs, the results may be inconsistent.

Intuitively, the replicas converge to the same state, which is reflected by the prefix of operations established by the primary (called the $committed$ list of operations) and the sequence of other operations ordered according to their timestamps (the $tentative$ list of operations). More precisely, when the stream of operations incoming to the system ceases and there are no network partitions (the replicas can reach the primary), the $committed$ lists at all replicas will be the same, whereas the $tentative$ lists will be empty. On the other hand, when there are partitions, some operations might not be successfully committed by the primary, but will be disseminated within a partition using a gossip protocol. Then all replicas within the same partition will have the same $committed$ and (non-empty) $tentative$ lists.

2.2.2 Anomalies

Now we discuss the consequences to the semantics of Bayou resulting from having two, inconsistent with each other, ways in which operations are ordered (the timestamp order and the order established by the primary).

Consider the example in Fig. 1, which shows an execution of a three-replica Bayou system. Initially, replica R_1 executes updating operations u_1 and u_2 in order u_2, u_1, which corresponds to u_1's and u_2's timestamps. This execution order is observed by the client that issues query q_1. On the other hand, R_2 executes the operations according to the final execution order (u_1, u_2), as established by the primary replica R_3. Hence, the client that issued query q_2 observes a different execution order than the client that issued q_1. Note that replicas execute the operations with a delay (e.g., due to CPU being busy) and that R_1 reexecutes the operations once it gets to know the final order.

Clearly, the clients that issued the operations can infer from the return values the order in which Bayou executed the operations. The observed execution orders differ between the clients accessing R_1 and R_2. We call this anomaly *temporary operation reordering*, as only eventually operations will observe the same serialization of any two past operations. Interestingly, the anomaly is present even though both u_1 and u_2 are weak. Temporary operation reordering is directly related to the sheer ability of the system to execute strongly consistent operations. This behaviour is not present in strongly consistent systems, which ensure that a single global ordering of operation execution is always respected (e.g., [32], [33]). The majority of eventually consistent systems which trade consistency for high availability are also free of this anomaly, as they only use one method to order concurrent operations (e.g., [22], [27]), or support only commutative operations (as in *strong eventual consistency* [2], e.g., [3], [34]). There are also protocols that allow past operations to be perceived in different (but still legal) orders (e.g., [14], [35], [36]). But, unlike Bayou,

they do not require the replicas to eventually agree on a single execution order for all operations. Interestingly, temporary operation reordering is not present in ANNC, because weak updating operations (add) commute and do not provide return values to clients.

Bayou exhibits another anomaly, which comes as very non-intuitive, i.e., *circular causality*. By analysing the return values of queries q_1 and q_2 one may conclude that there is a circular dependency between u_1 and u_2: u_1 depends on u_2 as evidenced by q_1's response, while u_2 depends on u_1 as evidenced by q_2's response (the cycle of causally related operations can contain more operations). Interestingly, as we show later, circular causality does not directly follow from temporary operation reordering but is rather a result of the way Bayou rolls back and reexecutes some operations.

'In the original Bayou protocol, application-specific conflict detection and resolution is accomplished through the use of *dependency checks* and *merge procedure* mechanisms. Since we allow operations with arbitrary complex semantics, the dependency checks and the merge procedures can be emulated by the operations themselves, by simply incorporating *if-else* statements: the dependency check as the *if* condition, and the merge procedure in the *else* branch (as suggested in the original paper [23]). Hence, these mechanisms do not alleviate the anomalies outlined above.

2.2.3 Correctness Guarantees

Because of the phenomena described above, the guarantees provided by Bayou cannot be formalized using the correctness criteria used for contemporary eventually consistent systems. E.g., *basic eventual consistency* (BEC) by Burckhardt *et al.* [26], [27] directly forbids circular causality (see Section 5.2 for definition of BEC). BEC also requires the relative order of any two operations, as perceived by the client, to be consistent and to never change. Similarly, *strong eventual consistency* (SEC) by Shapiro *et al.* [2] requires any two replicas that delivered the same updates to have equivalent states.[1] Obviously, Bayou neither satisfies BEC nor SEC (as evidenced by Fig. 1). On the other hand informal definitions of eventual consistency which admit temporal reordering, such as [28], involve only liveness guarantees, which is insufficient. Hence we introduce a new correctness criterion, *fluctuating eventual consistency* (FEC), which can be viewed as a generalization of BEC (see Section 5.3 for definition). FEC relaxes BEC, so different operations can perceive different operation orders. However, we require that the different perceived operation orders converge to one final execution order. Hence, FEC is suitable for systems that feature temporary operation reordering.

Similarly to ANNC, Bayou also ensures linearizability for strong operations (a response of a strong operation *op* always reflects the serial execution of all stabilized operations up to the point of *op*'s commit). In Section 5.6 we formally prove that the Bayou-derived general-purpose ACT satisfies the above correctness criteria.

In Appendix A.2, available in the online supplemental material, an interested reader may find a brief analysis of Bayou's liveness guarantees.

2.2.4 Fault-Tolerance

Bayou's reliance on the primary means that it provides only limited fault-tolerance. Even though the primary may recover, when it is down, operations do not stabilize, and thus no strong operation can complete. Hence, the primary is the single point of failure. Alternatively, the primary could be replaced by a distributed commit protocol. If two-phase-commit (2PC) [37] is used, the phenomena illustrated in Fig. 1 are not possible. In this case each replica votes to commit a given request. A replica postpones the commit of a request with a higher timestamp to ensure that its requests with lower timestamps are committed first. However, in this approach, a failure of any replica blocks the execution of strong operations. On the other hand, if a *non-blocking* commit protocol, e.g., one that utilizes a quorum-based implementation of TOB is used (as in ANNC), the system may stabilize operations despite (a limited number of) failures.[2] As we prove later, ACTs (which do not depend on the synchronous communication with all replicas and thus can operate despite failures of some of them) with general-purpose semantics similar to Bayou, are necessarily prone to the temporary operation reordering.

2.2.5 The Improved Bayou Protocol

Bayou can be improved to make it more fault-tolerant and free of some of the phenomena described above.

First, we use TOB in place of the primary to establish the final operation execution order. More precisely, each time a replica receives an operation *op* from a client, it still disseminates *op* using a gossip protocol (so it can reach at least all replicas within the same network partition) but it also broadcasts the operation using TOB (in a similar way in which weak updating operations are handled in ANNC). Since TOB guarantees that all replicas deliver the same set of messages in the same order, all replicas will stabilize the same set of operations in the same order. As we argued earlier, TOB can be implemented in a way that avoids a single point of failure [11].

The second modification is aimed at eliminating circular causality in Bayou. To this end (1) strong operations are broadcast using TOB and never a gossip protocol, and (2) upon being submitted, a weak operation *op* is executed immediately on the current state to produce the return value, even when other requests with lower timestamps are queued for execution; however, eventually all requests are executed in the order consistent with their timestamps. In Appendix A.3, available in the online supplemental material, we formally prove that above changes to the protocol allow us to avoid circular causality.

With the above modifications the improved Bayou protocol becomes the general-purpose ACT, called AcuteBayou.

1. BEC can be seen as a refinement of SEC, which abstracts away from CRDTs implementation details and ensures that no return value is constructed out of thin air.

2. Sharded 2PC [38] can be considered non-blocking, if within each shard at least one process remains operational at all times. Then, in such a scheme not every process needs to be contacted to commit a transaction, thus it falls under the quorum-based category.

2.3 ANNC versus AcuteBayou

While ANNC implements a very specific narrow data type, we can consider AcuteBayou as a generic ACT, capable of executing any set of weak and strong operations. In fact we could trivially implement a non-negative integer counter using AcuteBayou by executing each counter operation as a separate AcuteBayou operation, albeit such an implementation would be suboptimal: in some cases the operations would have to be rolled back and temporary operation reordering would be possible again.

Despite the many differences between ANNC and AcuteBayou, they share several design assumptions, which are common to all ACT implementations. First, in order to facilitate high availability and low response times (which are essential in geo-replicated environments), frequently invoked operations should be declared weak and replicas should process them similarly to operations in CRDTs (automatically resolve conflicts between concurrent updates; converge to the same state within a network partition). To enforce this behaviour without resorting to distributed agreement, we impose the same assumptions as Attiya *et al.* for highly available eventually consistent data stores in [34] (see Section 3.3 for details). Second, when weak consistency guarantees are insufficient, strong operations can be used. Strong operations use a global agreement protocol for inter-replica synchronization, e.g., TOB. We require that strong operations do not block the execution of weak operations and that they do not require all replicas to be operational at all times in order to complete (as in 2PC).

ACTs constitute a modular abstraction layer that handles all the complexities of replication, while enabling flexibility, high performance and clear mixed-consistency semantics. In the next section we specify ACTs formally.

3 Acute Cloud Types

3.1 Definition

An *acute cloud type (ACT)* is an abstract data type, implemented as a replicated data structure, that offers a precisely defined set of operations, divided into two groups: weak and strong. The operations can be either updating or read-only (RO), and all operations are allowed to provide a return value (in Section 4 we show how the semantics of operations can be specified formally). We impose the following *implementation restrictions* over ACTs: *invisible reads*, *input-driven processing*, *op-driven messages*, *highly available weak operations* and *non-blocking strong operations*. The first four, are adapted from the definition of *write-propagating data stores* [34] and guarantee *genuine*, low-latency, eventually-consistent processing for weak operations (as in, e.g., CRDTs [2]). The last restriction guarantees that strong operations are implemented using a non-blocking agreement protocol, instead of a fault-prone approach requiring all the replicas to be operational. In Sections 3.2 and 3.3 we formalize the system model and provide precise definitions of the implementation restrictions.

3.2 System Model

3.2.1 Replicas and Clients

We consider a system consisting of $n \geq 2$ processes called *replicas*, which maintain full copies of an ACT and to which external clients submit requests in the form of operations to be executed.[3] Each operation invoked by a client is marked either *weak* or *strong*. Replicas communicate with each other through message passing. We assume the availability of a gossip protocol, which is used when ordering constraints are not necessary, and some global agreement protocol, used for tasks that require solving distributed consensus. For simplicity, as in Algorithm 1, we formalize these protocols using *reliable broadcast (RB)* [30], and TOB, respectively. Replicas can implement point-to-point communication simply by ignoring messages for which they are not the intended recipient. We model replicas as deterministic state machines, which execute atomic steps in reaction to external events (e.g., operation invocation or message delivery), and can execute internal events (e.g., scheduled processing of rollbacks). A specific event is *enabled* on a replica, if its preconditions are met (e.g., an $\text{RB-deliver}(m)$ event is enabled on a replica R, if m was previously RB-cast and R has not delivered message m yet). Replicas have access to a local clock, which advances monotonically, but we make no assumptions on the bound on clock drift between replicas.

We model crashed replicas as if they stopped all computation (or compute infinitely slowly). We say that a replica is *faulty* if it crashes (in an infinite execution it executes only a finite number of steps). Otherwise, it is *correct*.

3.2.2 Network Properties

In a fully asynchronous system, a crashed replica is indistinguishable to its peers from a very slow one, and it is impossible to solve the distributed consensus problem [39]. Real distributed systems which exhibit some amount of *synchrony* can usually overcome this limitation. For example, in a quasi-synchronous model [40], the system is considered to be synchronous, but there exist a non-negligible probability that timing assumptions can be broken. We are interested in the behaviour of protocols, both in the fully *asynchronous* environment, when timing assumptions are consistently broken (e.g., because of prevalent network partitions), and in a *stable* one, when the minimal amount of synchrony is available so that consensus eventually terminates. Thus, we consider two kinds of *runs*: *asynchronous runs* and *stable runs*. Replicas are not aware which kind of a run they are currently executing. In stable runs, we augment the system with the failure detector Ω (which is an abstraction for the synchronous aspects of the system). We do so implicitly by allowing the replicas to use TOB through the TOB-cast and TOB-deliver primitives. Since, TOB is known to require a failure detector at least as strong as Ω to terminate [41], we guarantee it achieves progress only in stable runs.

In both asynchronous and stable runs we guarantee the basic properties of reliable message passing [30], i.e.,:

- if a message is RB-delivered, or TOB-delivered, then it was, respectively, RB-cast, or TOB-cast, by some replica,
- no message is RB-delivered, or TOB-delivered, more than once by the same replica,
- if a correct replica RB-casts some message, then eventually it RB-delivers it,

3. We assume full replication for simplicity.

- if a correct replica RB-delivers some message, then eventually all correct replicas RB-deliver it,
- if any (correct or faulty) replica TOB-delivers some message, then eventually all correct replicas TOB-deliver it,
- messages are TOB-delivered by all replicas in the same total order.

We define $tobNo(m)$ as the sequence number of the TOB-$deliver(m)$ event (among other TOB-deliver events in the execution) on any replica (we leave it undefined, i.e., $tobNo(m) = \perp$, if m is never TOB-delivered by any replica).

Solely in stable runs, we also guarantee the following:

- if a correct replica TOB-casts some message, then eventually all correct replicas TOB-deliver it.
- if a message m was both RB-cast and TOB-cast by some (correct or faulty) replica, and m was RB-*deliver*ed by some correct replica, then eventually all correct replicas TOB-deliver it.

The last guarantee is non-standard for a total-order broadcast, but could be easily emulated by the application itself. We include it to simplify the presentation of certain algorithms, such as ANNC and AcuteBayou.

3.2.3 Fair Executions

An execution is *fair*, if each replica has a chance to execute its steps (all replicas execute infinitely many steps of each type of an enabled event, e.g., infinitely many RB-deliver events for infinitely many messages RB-cast).

We analyze the correctness of a protocol by evaluating a single arbitrary infinite fair execution of the protocol, similarly to [26] and [42]. If the execution satisfies the desired properties, then all the executions of the protocol (including finite ones and the ones featuring crashed replicas) satisfy all the safety aspects verified (*nothing bad ever happens* [43], [44]). Additionally, all fair executions of the protocol satisfy liveness aspects (*something good eventually happens*).

3.3 Implementation Restrictions

Below we state the five rules that ACTs need to adhere to.

1. Invisible Reads. Replicas do not change their state due to an invocation of a weak read-only operation. Formally, for each weak read-only operation op invoked on a replica R in state σ, the state of R after a response for op is returned is equal σ. Note that, the consequence of this is that weak read-only operations need to return a response to the client immediately in the invoke event, without executing any other steps. We allow strong read-only operations to change the state of a replica, because sometimes it is necessary to synchronize with other replicas, and then the replica needs to note down that a response is pending.

2. Input-Driven Processing. Replicas execute a series of steps only in response to some external stimulus, e.g., an operation invocation or a received message. A state σ of a replica R is *passive* if none of the internal events on the replica are enabled in σ. Initially each replica is in a passive state. An external event may bring a replica to an *active* state σ' in which it has some internal events enabled. Then, after executing a finite number of internal events (when no new external events are executed), the replica enters a passive state. More formally, for each replica R, we require that in a given execution, either there is only a finite number of internal events executed on R, or there is an infinite number of external events executed on R. We say that R is *passive*, if it is in a passive state, otherwise it is *active*.

3. Op-Driven Messages. RB or TOB messages are only generated and sent as a result of some non-read-only client operation, and not spontaneously or in response to a received message. More formally, a message can be RB-cast or TOB-*cast* by a replica R, if previously some non-read-only operation was invoked on R, and since then R did not enter a passive state.

4. Highly Available Weak Operations. Weak operations need to eventually return a response without communicating with other replicas. A weak operation op may remain pending only if the execution is finite, and the executing replica remains active since the invocation of op (in an infinite execution a pending weak operation is never allowed).

5. Non-Blocking Strong Operations. Strong operations need to eventually return a response if a global agreement has been reached. More formally, for a strong operation op invoked on a replica R, let $msgs$ be the set of all messages TOB-cast by R since the invocation of op but before R enters a passive state. Then, op may remain pending only if:

- the execution is finite, and R remains active since the invocation of op, or R remains active because of the delivery of any message $m \in msgs$, or
- there exists a message $m \in msgs$, which has not been TOB-delivered by R yet.

It means that in order to execute a strong operation replicas may synchronize by TOB-casting multiple messages, but once TOB completes, the response must be returned in a finite number of steps.

All the above requirements are commonly met by various eventually consistent data stores and CRDTs (when we consider them as ACTs with only weak operations and using our communication model[4]), see, e.g., [1], [2], [34], [42], [45], [46], [47], [48], [49]. Restrictions 1–4 are inspired by the ones defined for write-propagating data stores [34], but modified appropriately to accommodate for the more complex nature of ACTs. In particular, we allow implementations that do not execute each invoked operation in one atomic step, but divide the execution between many internal steps (e.g., see the pseudocode of Bayou in Appendix A.1, available in the online supplemental material). On the other hand, the 5th requirement concerns strong operations, and so is specific for ACTs. As discussed in [34], [42], requirements 1–4 preclude implementations which offer stronger consistency guarantees but do not provide a real value to the programmer (and still fall short of the guarantees possible to ensure if global agreement can be reached). For example, a register's implementation lacking invisible reads can return not the most recent value, but a stale one, unless the read operation was invoked earlier a certain number of times. Such an implementation is more restrictive compared to a classic

4. In case of geo-replicated systems which are weakly consistent between data centers, but feature state machine replication within a data center to simulate reliable processes, we can consider the whole data center as a single replica.

Property	Element-wise Definition $\forall x, y, z \in A:$	Algebraic Definition
symmetric	$x \xrightarrow{rel} y \Rightarrow y \xrightarrow{rel} x$	$rel = rel^{-1}$
reflexive	$x \xrightarrow{rel} x$	$id_A \subseteq rel$
irreflexive	$x \not\xrightarrow{rel} x$	$id_A \cap rel = \emptyset$
transitive	$(x \xrightarrow{rel} y \xrightarrow{rel} z) \Rightarrow (x \xrightarrow{rel} z)$	$(rel; rel) \subseteq rel$
acyclic	$\neg(x \xrightarrow{rel} ... \xrightarrow{rel} x)$	$id_A \cap rel^+ = \emptyset$
total	$x \neq y \Rightarrow (x \xrightarrow{rel} y \vee y \xrightarrow{rel} x)$	$rel \cup rel^{-1} \cup id_A = A \times A$

Property	Definition
natural	$\forall x \in A : \lvert rel^{-1}(x)\rvert < \infty$
partialorder	*irreflexive* $\wedge$ *transitive*
totalorder	*partialorder* $\wedge$ *total*
enumeration	*totalorder* $\wedge$ *natural*
equivalencerelation	*reflexive* $\wedge$ *transitive* $\wedge$*symmetric*

Fig. 2. Definitions of common properties of a binary relation $rel \subseteq A \times A$.

register, i.e., it admits fewer execution traces. Thus it satisfies a more stringent consistency guarantee, albeit not a very useful one. On the other hand, *with* the above restrictions, it is still possible to attain causal consistency and variants of it, such as *observable causal consistency* [34].

4 Formal Framework

Below we provide the formal framework that allows us to reason about execution histories and correctness criteria. We extend the framework by Burckhardt *et al.* [26], [27] (also used by several other researchers, e.g., [34], [42], [50], [51]).

4.1 Preliminaries

Relations. A binary relation rel over set A is a subset $rel \subseteq A \times A$. For $a, b \in A$, we use the notation $a \xrightarrow{rel} b$ to denote $(a, b) \in rel$, and the notation $rel(a)$ to denote $\{b \in A : a \xrightarrow{rel} b\}$. We use the notation rel^{-1} to denote the inverse relation, i.e., $(a \xrightarrow{rel^{-1}} b) \Leftrightarrow (b \xrightarrow{rel} a)$. Therefore, $rel^{-1}(b) = \{a \in A : a \xrightarrow{rel} b\}$. Given two binary relations rel, rel' over A, we define the composition $rel; rel' = \{(a, c) : \exists b \in A : a \xrightarrow{rel} b \xrightarrow{rel'} c\}$. We let id_A be the identity relation over A, i.e., $(a \xrightarrow{id_A} b) \Leftrightarrow (a \in A) \wedge (a = b)$. For $n \in \mathbb{N}_0$, we let rel^n be the n-ary composition $rel; rel \ldots; rel$, with $rel^0 = id_A$. We let $rel^+ = \bigcup_{n \geq 1} rel^n$ and $rel^* = \bigcup_{n \geq 0} rel^n$. For some subset $A' \subseteq A$, we define the restricted relation $rel|_{A'} = rel \cap (A' \times A')$. In Fig. 2 we summarize various properties of relations.

We define by $words(A)$ the set of all sequences (words) containing only elements from the set A. When not ambiguous we use A^* to denote $words(A)$ (i.e., when A is not a binary relation).

Let $rank(A, rel, a)$ denote the number of elements of set A that are in relation rel to element $a \in A$. Thus, $rank(A, rel, a) = |\{x \in A : x \xrightarrow{rel} a\}| = |rel^{-1}(a) \cap A|$.

We also define two operators *sort* and *foldl*. $A.sort(rel) \in A^*$ arranges in an ascending order the elements of set A according to the total order rel. $foldl(a_0, f, w) \in A$ reduces sequence $w \in B^*$ by one element at a time using the function $f : A \times B \to A$ and accumulator $a_0 \in A$:

$$foldl(a_0, f, w) = \begin{cases} a_0 & \text{if } w = \epsilon \\ f(foldl(a_0, f, w'), b) & \text{if } w = w'b \end{cases}.$$

Event Graphs. To reason about executions of a distributed system we encode the information about events that occur in the system and about various dependencies between them in the form of an *event graph*. An *event graph* G is a tuple $(E, d_1, \ldots, d_n)$, where $E \subseteq Events$ is a finite or countably infinite set of events drawn from universe $Events$, $n \geq 1$, and each d_i is an attribute or a relation over E. *Vertices* in G represent events that occurred at some point during the execution and are interpreted as opaque identifiers. *Attributes* label vertices with information pertinent to the corresponding event, e.g., operation performed, or the value returned. All possible operations of all considered data types form the $Operations$ set. All possible return values of all operations form the $Values$ set. *Relations* represent orderings or groupings of events, and thus can be understood as *arcs* or *edges* of the graph.

Event graphs are meant to carry information that is independent of the actual elements of $Events$ chosen to represent the events (the attributes and relations in G encode all relevant information regarding the execution). Let $G = (E, d_1, \ldots, d_n)$ and $G' = (E', d'_1, \ldots, d'_n)$ be two event graphs. G and G' are *isomorphic*, written $G \simeq G'$, if (1) for all $i \geq 1$, d_i and d'_i are of the same kind (attribute versus relation) and (2) there exists a bijection $\phi : E \to E'$ such that for all d_i, where d_i is an attribute, and all $x \in E$, we have $d_i(x) = d'_i(\phi(x))$, and such that for all d_i where d_i is a relation, and all $x, y \in E$, we have $x \xrightarrow{d_i} y \Leftrightarrow \phi(x) \xrightarrow{d'_i} \phi(y)$.

4.2 Histories

We represent a high-level view of a system execution as a *history*. We omit implementation details such as message exchanges or internal steps executed by the replicas. We include only the observable behaviour of the system, as perceived by the clients through received responses. Formally, we define a *history* as an event graph $H = (E, op, rval, rb, ss, lvl)$, where:

- $op : E \to Operations$, specifies the operation invoked in a particular event, e.g., $op(e) = write(3)$,
- $rval : E \to Values \cup \{\nabla\}$, specifies the value returned by the operation, e.g., $rval(e) = 3$, or $rval(e') = \nabla$, if the operation never returns (e' is *pending* in H),
- rb, the *returns-before* relation, is a natural partial order over E, which specifies the ordering of *non-overlapping* operations (one operation returns before the other starts, in real-time),
- ss, the *same session* relation, is an equivalence relation which groups events executed within the same session (the same client), and finally
- $lvl : E \to \{weak, strong\}$, specifies the consistency level demanded for the invoked operation.

We consider only *well-formed* histories, which satisfy:

- $\forall a, b \in E : (a \xrightarrow{rb} b \Rightarrow rval(a) \neq \nabla)$ (a pending operation does not return),
- $\forall a, b, c, d \in E : (a \xrightarrow{rb} b \wedge c \xrightarrow{rb} d) \Rightarrow (a \xrightarrow{rb} d \vee c \xrightarrow{rb} b)$ (rb is an *interval order*, i.e., it is consistent with a timeline interpretation, where operations correspond to segments [26], [52]),
- for each event $e \in E$ and its session $S = \{e' \in E : e \xrightarrow{ss} e'\}$, the restriction $rb|_S$ is an enumeration (clients issue operations sequentially).

4.3 Abstract Executions

In order to *explain* the history, i.e., the observed return values, and reason about the system properties, we need to extend the history with information about the abstract relationships between events. For strongly consistent systems typically we do so by finding a *serialization* [21] (an enumeration of all events) that satisfies certain criteria. For weaker consistency models, such as *eventual consistency* or *causal consistency*, it is more natural to reason about partial ordering of events. Hence, we resort to *abstract executions*.

An *abstract execution* is an event graph $A = (E, op, rval, rb, ss, lvl, vis, ar, par)$, such that:

- $(E, op, rval, rb, ss, lvl)$ is some history H,
- vis is an acyclic and natural relation,
- ar is a total order relation, and
- $par : E \to 2^{E \times E}$ is a function which returns a binary relation in E.

For brevity, we often use a shorter notation $A = (H, vis, ar, par)$ and let $\mathcal{H}(A) = H$. Just as serializations are used to explain and justify operations' return values reported in a history, so are the *visibility* (vis) and *arbitration* (ar) relations. *Perceived arbitration* (par) is a function which is necessary to formalize temporary operation reordering.

Visibility (vis) describes the relative influence of operation executions in a history on each others' return values: if a is visible to b (denoted $a \xrightarrow{vis} b$), then the effect of a is visible to the replica performing b (and thus reflected in b's return value). Visibility often mirrors how updates propagate through the system, but it is not tied to the low-level phenomena, such as message delivery. It is an acyclic, natural relation, which may or may not be transitive. Two events are *concurrent* if they are not ordered by visibility.

Arbitration (ar) is an additional ordering of events which is necessary in case of non-commutative operations. It describes how the effects of these operations should be applied. If a is arbitrated before b (denoted $a \xrightarrow{ar} b$), then a is considered to have been executed earlier than b. Arbitration is essential for resolving conflicts between concurrent events, but it is defined as a total-order over *all* operation executions in a history. It usually matches whatever conflict resolution scheme is used in an actual system, be it physical time-based timestamps, or logical clocks.

Perceived arbitration (par) describes the relative order of operation executions, as perceived by each operation ($par(e)$ defines the total order of all operations, as perceived by event e). If $\forall e \in E : par(e) = ar$, then there is no temporary operation reordering in A.

4.4 Correctness Predicates

A *consistency guarantee* $\mathcal{P}(A)$ is a set of conditions on an abstract execution A, which depend on the particulars of A up to isomorphism. For brevity we usually omit the argument A. We write $A \models \mathcal{P}$ if A *satisfies* $\mathcal{P}$. More precisely: $A \models \mathcal{P} \overset{\text{def}}{\iff} \forall A' : A' \simeq A : \mathcal{P}(A')$. A history H is *correct* according to some consistency guarantee $\mathcal{P}$ (written $H \models \mathcal{P}$) if it can be extended with some vis, ar relations and par function to an abstract execution $A = (H, vis, ar, par)$ that satisfies $\mathcal{P}$. We say that a system is correct according to some consistency guarantee $\mathcal{P}$ if all of its histories satisfy $\mathcal{P}$.

We say that a consistency guarantee $\mathcal{P}_i$ is at least as strong as a consistency guarantee $\mathcal{P}_j$, denoted $\mathcal{P}_i \geq \mathcal{P}_j$, if $\forall H : H \models \mathcal{P}_i \Rightarrow H \models \mathcal{P}_j$. If $\mathcal{P}_i \geq \mathcal{P}_j$ and $\mathcal{P}_j \not\geq \mathcal{P}_i$ then $\mathcal{P}_i$ is stronger than $\mathcal{P}_j$, denoted $\mathcal{P}_i > \mathcal{P}_j$. If $\mathcal{P}_i \not\geq \mathcal{P}_j$ and $\mathcal{P}_j \not\geq \mathcal{P}_i$, then $\mathcal{P}_i$ and $\mathcal{P}_j$ are incomparable, denoted $\mathcal{P}_i \lesseqgtr \mathcal{P}_j$.

4.5 Replicated Data Type

In order to specify semantics of operations invoked by the clients on the replicas, we model the whole system as a single replicated object (as in case of Algorithm 1). Even though we use only a single object, this approach is general, as multiple objects can be viewed as a single instance of a more complicated type, e.g., multiple registers constitute a single *key-value store*. Defining the semantics of the replicated object through a sequential specification [10] is not sufficient for replicated objects which expose concurrency to the client, e.g., multi-value register (MVR) [2] or observed-remove set (OR-Set) [3]. Hence, we utilize *replicated data types* specification [46].

In this approach, the state on which an operation $op \in Operations$ executes, called the *operation context*, is formalized by the event graph of the prior operations visible to op. Formally, for any event e in an abstract execution $A = (E, op, rval, rb, ss, lvl, vis, ar, par)$, the operation context of e in A is the event graph $context(A, e) \overset{\text{def}}{=} (vis^{-1}(e), op, vis, ar)$. Note that an operation context lacks return values, the returns-before relation, and the information about sessions. The set of previously invoked operations and their relative visibility and arbitration unambiguously defines the output of each operation. This brings us to the formal definition of a replicated data type.

A *replicated data type* $\mathcal{F}$ is a function that, for each operation $op \in ops(\mathcal{F})$ (where $ops(\mathcal{F}) \subseteq Operations$) and operation context C, defines the expected return value $v = \mathcal{F}(op, C) \in Values$, such that v does not depend on events, i.e., is the same for isomorphic contexts: $C \simeq C' \Rightarrow \mathcal{F}(op, C) = \mathcal{F}(op, C')$ for all op, C, C'. We say that $op \in ops(\mathcal{F})$ is a read-only operation (denoted $op \in readonlyops(\mathcal{F})$), if and only if, for any operation op', context $C = (E, op, vis, ar)$ and event $e \in E$, such that $op(e) = op$, $\mathcal{F}(op', C) = \mathcal{F}(op', C')$, where $C' = (E \setminus \{e\}, op, vis, ar)$. In other words, read-only operations can be excluded from any context C, producing C', and the result of any operation op' will not change.

In Fig. 3 we give the specification of three replicated data types: $\mathcal{F}_{MVR}$ (a multi-value register), $\mathcal{F}_{seq}$ (an append-only sequence), and $\mathcal{F}_{NNC}$ (a non-negative counter). We use $\mathcal{F}_{seq}$ in the subsequent sections to illustrate various consistency models.

$$\mathcal{F}_{MVR}(write(v),(E,op,vis,ar)) = ok$$
$$\mathcal{F}_{MVR}(read,(E,op,vis,ar)) = \{v : \exists e \in E : op(e) = write(v) \wedge \nexists e' \in E : op(e') = write(v') \wedge e \xrightarrow{vis} e'\}$$

$$\mathcal{F}_{seq}(append(s),(E,op,vis,ar)) = ok$$
$$\mathcal{F}_{seq}(read,(E,op,vis,ar)) = \begin{cases} \epsilon, & \text{if } appends(E) = \emptyset \\ s_1 \cdot s_2 \cdot \ldots \cdot s_n & \text{otherwise, where } \forall i \le n : \exists e \in E : rank(appends(E), ar, e) = i \\ & \qquad \wedge\ op(e) = append(s_i) \wedge s_i \in \$^* \end{cases}$$
$$\text{where } \$ = \{a, b, c, \ldots, z\} \text{ and } appends(E) = \{e \in E : \exists v \in \$^* : op(e) = append(v)\}$$

$$\mathcal{F}_{NNC}(add(v),(E,op,vis,ar)) = ok$$
$$\mathcal{F}_{NNC}(subtract(v),(E,op,vis,ar)) = \begin{cases} true & \text{if } foldl(0, f_{NNC}, E.sort(ar)) \ge v \\ false & \text{otherwise} \end{cases}$$
$$\mathcal{F}_{NNC}(get,(E,op,vis,ar)) = foldl(0, f_{NNC}, E.sort(ar))$$
$$\text{where } v \in \mathbb{N} \text{ and } f_{NNC} = \begin{cases} f_{NNC}(x, add(v)) & = x + v \\ f_{NNC}(x, subtract(v)) & = x - v \text{ if } x \ge v \text{ or } x \text{ otherwise} \\ f_{NNC}(x, get) & = x \end{cases}$$

Fig. 3. Formal specifications of a multi-value register data type $\mathcal{F}_{MVR}$, an append-only sequence data type $\mathcal{F}_{seq}$, and a non-negative counter $\mathcal{F}_{NNC}$. An instance of $\mathcal{F}_{MVR}$ stores multiple values when there are concurrent $write()$ operations ($write()$ operations not ordered by the vis relation). $\mathcal{F}_{seq}$ can be used to create a sequence of characters (a word), where the set of characters is limited to a through z. $\mathcal{F}_{seq}$ features two operations: $append(x)$, which appends x to the end of the sequence and returns $ok \in Values$, and $read()$, which returns a sequence (a word) $w \in \$^*$. $\mathcal{F}_{NNC}$ stores an integer value, that can be increased using the add operation or decreased using the $subtract$ operation, but only if the value of the counter will not decrease below 0. The get operation simply returns the current value of the counter. See the definition of operators $sort$ and $foldl$ in Section 4.1.

4.6 ACT Specification

To accommodate for the mixed-consistency nature of ACTs we extend replicated data type specification with the information on supported consistency levels for a given operation. Thus, we define *ACT specification* as a pair $(\mathcal{F}, lvlmap)$, where $\mathcal{F}$ is a replicated data type specification and $lvlmap : Operations \to 2^{\{weak,strong\}}$ is a function which specifies for each $op \in Operations$ with which consistency levels it can be executed. We assume that clients follow this contract, and thus, when considering a history $H = (E, op, rval, rb, ss, lvl)$ of an ACT compliant with the specification $(\mathcal{F}, lvlmap)$, we assume that for each $e \in E : lvl(e) \in lvlmap(op(e))$.

Then, ANNC's specification is $(\mathcal{F}_{NNC}, lvlmap_{NNC})$, where $lvlmap_{NNC}(add) = lvlmap_{NNC}(get) = \{weak\}$ and $lvlmap_{NNC}(subtract) = \{strong\}$.

5 Correctness Guarantees

In this section we define various correctness guarantees for ACTs. We define them as conjunctions of several basic predicates. We start with two simple requirements that should naturally be present in any eventually consistent system. For the discussion below we assume some arbitrary abstract execution $A = (E, op, rval, rb, ss, lvl, vis, ar, par)$.

5.1 Key Requirements for Eventual Consistency

The first requirement is the *eventual visibility* (EV) of events. EV requires that for any event e in A, there is only a finite number of events in E that do not observe e. Formally

$$\text{EV} \stackrel{\text{def}}{=} \forall e \in E : |\{e' \in E : e \xrightarrow{rb} e' \wedge e \not\xrightarrow{vis} e'\}| < \infty.$$

Intuitively, EV implies progress in the system because replicas must synchronize and exchange knowledge about operations submitted to the system.

The second requirement concerns avoiding circular causality, as discussed in Section 2.2.2. To this end we define two auxiliary relations: *session order* and *happens-before*. The session order relation $so \stackrel{\text{def}}{=} rb \cap ss$ represents the order of operations in each session. The happens-before relation $hb \stackrel{\text{def}}{=} (so \cup vis)^+$ (a transitive closure of session order and visibility) allows us to express the causal dependency between events. Intuitively, if $e \xrightarrow{hb} e'$, then e' potentially depends on e. We simply require *no circular causality*:

$$\text{NCC} \stackrel{\text{def}}{=} acyclic(hb).$$

In the following sections we add requirements on the return values of the operations in A. Formalizing the properties of ACTs which, similarly to AcuteBayou, admit temporary operation reordering, requires a new approach. We start, however, with the traditional one.

5.2 Basic Eventual Consistency

Intuitively, *basic eventual consistency* (BEC) [26], [27], in addition to EV and NCC, requires that the return value of each invoked operation can be explained using the specification of the replicated data type $\mathcal{F}$, which is formalized as follows:

$$\text{RVal}(\mathcal{F}) \stackrel{\text{def}}{=} \forall e \in E : rval(e) = \mathcal{F}(op(e), context(A, e)).$$

Then

$$\text{BEC}(\mathcal{F}) \stackrel{\text{def}}{=} \text{EV} \wedge \text{NCC} \wedge \text{RVal}(\mathcal{F}).$$

An example abstract execution A_{BEC} that satisfies BEC $(\mathcal{F}_{seq})$ is shown in Fig. 4. In A_{BEC}, replicas R_1 and R_2 concurrently execute two $append()$ operations, and then each replica executes an infinite number of $read()$ operations. Consider the $read()$ operations on R_2: the first one observes only $append(a)$ (which is in the operation context of $read()$), whereas the second observes only $append(b)$. BEC admits this kind of execution, because it does not make any requirements in terms of session guarantees [53]. Eventually, both $append(a)$ and $append(b)$ become visible to all subsequent $read()$ operations, thus satisfying EV.

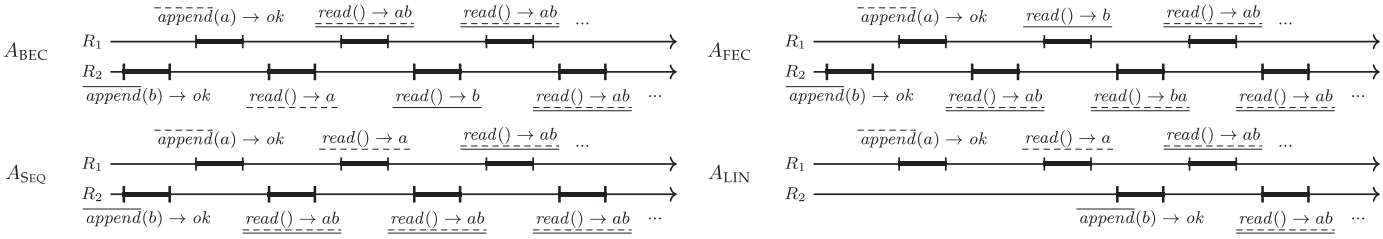

Fig. 4. Example abstract executions of systems with a list semantics that satisfy $\text{BEC}(\mathcal{F}_{seq})$, $\text{FEC}(\mathcal{F}_{seq})$, $\text{SEQ}(\mathcal{F}_{seq})$, and $\text{LIN}(\mathcal{F}_{seq})$ respectively (for brevity, we omit the level parameter l and assume that all operations belong to the same class l). We use solid and dashed underlines to depict which updating operations are visible (through relation vis) in A to the $read()$ operations (we assume that every $read()$ operation observes all other $read()$ operations that happened prior to it). In the arbitration order, $append(a)$ precedes $append(b)$, and both updates are followed by all the reads in the order they appear on the timeline.

By the definition of the $context$ function (Section 4.5), when A satisfies $\text{RVAL}(\mathcal{F})$, the return value of each operation is calculated according to the ar relation. It is then easy to see that there are executions of AcuteBayou (or other ACTs that admit temporary operation reordering) which do not satisfy $\text{RVAL}(\mathcal{F})$. It is because weak operations (as shown in Section 2.2.2) might observe past operations in an order that differs from the final operation execution order (ar). Hence AcuteBayou does not satisfy $\text{BEC}(\mathcal{F})$ for an arbitrary $\mathcal{F}$. But it could satisfy $\text{BEC}(\mathcal{F})$ for a sufficiently simple $\mathcal{F}$, such as a conflict-free counter, in which all operations always commute (as opposed to $\mathcal{F}_{NNC}$). It is so, because then, even if AcuteBayou reorders some operations internally, the final result never changes and thus the reordering cannot be observed by the clients.

5.3 Fluctuating Eventual Consistency

In order to admit temporary operation reordering, we give a slightly different definition of the $context$ function, in which the arbitration order *fluctuates*, i.e., it changes from one event to another. Let $fcontext(A, e) \stackrel{\text{def}}{=} (vis^{-1}(e), op, vis, par(e))$, which means that now we consider the operation execution order as perceived by e, and not the final one. The definition of the fluctuating variant of RVAL is straightforward:

$$\text{FRVAL}(\mathcal{F}) \stackrel{\text{def}}{=} \forall \in E : rval(e) = \mathcal{F}(op(e), fcontext(A, e)).$$

To define the fluctuating variant of BEC, that could be used to formalize the guarantees provided by ACTs we additionally must ensure that the arbitration order perceived by events is not completely unrestricted, but that it gradually *converges* to ar for each subsequent event. It means that each $e \in E$ can be temporarily observed by the subsequent events e' according to an order that differs from ar (but is consistent with $par(e')$). However, from some moment on, every event e' will observe e according to ar. To define this requirement, we use the $rank$ function (defined in Section 4.1). Let $E_e = \{e' \in E : e \xrightarrow{vis} e'\}$. This intuition is formalized by *convergent perceived arbitration*:

$$\text{CPAR} \stackrel{\text{def}}{=} \forall e \in E : |\{e' \in E_e : rank(vis^{-1}(e'), par(e'), e) \neq rank(vis^{-1}(e'), ar, e)\}| < \infty.$$

If A satisfies CPAR, then for each event e, the set of events e', which observe the position of e not according to ar is finite. Thus, the position of e in $par(e')$ for subsequent events e' stabilizes, and $par(e')$ eventually converges to ar.

Now we can define our new consistency criterion *fluctuating eventual consistency* (FEC):

$$\text{FEC}(\mathcal{F}) \stackrel{\text{def}}{=} \text{EV} \wedge \text{NCC} \wedge \text{FRVAL}(\mathcal{F}) \wedge \text{CPAR}.$$

An example abstract execution A_{FEC} that satisfies FEC is shown in Fig. 4. In A_{FEC}, replica R_2 temporarily observes the $append()$ operations in the order $append(b), append(a)$ which is different than the eventual operation execution order (as evidenced by the infinite number of $read() \to ab$ operations). We call this behaviour *fluctuation*.

It is easy to see that $\text{FEC}(\mathcal{F}) < \text{BEC}(\mathcal{F})$, in the sense that: for each $\mathcal{F}$, $\text{FEC}(\mathcal{F}) \leq \text{BEC}(\mathcal{F})$, and for some $\mathcal{F}$, $\text{FEC}(\mathcal{F}) < \text{BEC}(\mathcal{F})$. It is so, because FEC uses par instead of ar to calculate the return values of operation executions, but par eventually converges to ar. Hence, $\text{BEC}(\mathcal{F})$ is a special case of $\text{FEC}(\mathcal{F})$, when $\forall e \in E : par(e) = ar$. It is easy to see that A_{BEC} from Fig. 4 satisfies both BEC and FEC, whereas A_{FEC} satisfies only FEC.

5.4 Operation Levels

The above definitions can be used to capture the guarantees provided by a wide variety of eventually consistent systems. However, our framework still lacks the capability to express the semantics of mixed-consistency systems. ACTs offer different guarantees for different classes of operations (e.g., consistency guarantees stronger than BEC or FEC are provided in AcuteBayou or ANNC only for strong operations). Hence, we need to parametrize the consistency criteria with a level attribute (as indicated by the lvl function for each event). Since consistency level is specified per operation invocation, we need to assure that the respective operations' responses reflect the demanded consistency level.

Let us revisit BEC first. Let $L = \{e \in E : lvl(e) = l\}$ for a given l. Then

$$\text{EV}(l) \stackrel{\text{def}}{=} \forall e \in E : |\{e' \in L : e \xrightarrow{rb} e' \wedge e \not\xrightarrow{vis} e'\}| < \infty$$

$$\text{NCC}(l) \stackrel{\text{def}}{=} acyclic(hb \cap (L \times L))$$

$$\text{RVAL}(l, \mathcal{F}) \stackrel{\text{def}}{=} \forall e \in L : rval(e) = \mathcal{F}(op(e), context(A, e))$$

$$\text{BEC}(l, \mathcal{F}) \stackrel{\text{def}}{=} \text{EV}(l) \wedge \text{NCC}(l) \wedge \text{RVAL}(l, \mathcal{F}).$$

The above parametrized definition of BEC restricts the RVAL predicate only to events issued with the given consistency level l (the events that belong to the set L). It means that for any such event the response has to conform with the replicated data type specification $\mathcal{F}$, and with the vis and ar relations (as

defined by the definition of the $context$ function). For all other events this requirement does not need to be satisfied, so they can return arbitrary responses (unless restricted by other predicates targeted for these events). Similarly, for EV and NCC, the predicates are restricted to affect only the events from the set L. In case of EV, each event eventually becomes visible to the operations executed with the level l. In case of NCC, there must be no cycles in hb involving events from the set L.

The parametrized variant of FEC is formulated analogously. Let L be defined as before, and for any event $e \in E$, let $L_e = \{e' \in L : e \xrightarrow{vis} e'\}$ be the subset of events from L which observe e. Then

$$\begin{aligned}\text{FRVAL}(l,\mathcal{F}) &\stackrel{\text{def}}{=} \forall e \in L : rval(e) = \mathcal{F}(op(e), fcontext(A,e))\\ \text{CPAR}(l) &\stackrel{\text{def}}{=} \forall e \in E : |\{e' \in L_e : rank(vis^{-1}(e'), par(e'), e)\\ &\qquad \neq rank(vis^{-1}(e'), ar, e)\}| < \infty\\ \text{FEC}(l,\mathcal{F}) &\stackrel{\text{def}}{=} \text{EV}(l) \wedge \text{NCC}(l) \wedge \text{FRVAL}(l,\mathcal{F}) \wedge \text{CPAR}(l).\end{aligned}$$

As before, we restrict the return values only for the events from the set L. Additionally, we restrict the predicate CPAR, so that $par(e)$ converges towards ar only for events $e \in L$. Other events can differently perceive the arbitration of events (in principle, the observed arbitration can be completely different from the final one, specified by ar).

We can compare the parametrized variants of BEC and FEC as before: $\text{FEC}(l,\mathcal{F}) < \text{BEC}(l,\mathcal{F})$.

All of the strong consistency criteria which we are going to discuss next, we define already in the parametrized form with the given level l in mind, so they can be used, e.g., for strong operations in AcuteBayou and ANNC.

5.5 Strong Consistency

A common feature of strong consistency criteria, such as *sequential consistency* [21], or *linearizability* [10], is a single global serialization of all operations. It means that a history satisfies these criteria, if it is equivalent to some serial execution (*serialization*) of all the operations. Additionally, depending on the particular criterion, the serialization must, e.g., respect program-order, or real-time order of operation executions. Although we provide a serialization of all operations (through the total order relation ar, which is part of every abstract execution), the equivalence of a history to the serialization is not enforced in the correctness criteria we have defined so far. For example, given a sequence of three events $\langle a,b,c\rangle$, such that $a \xrightarrow{ar} b \xrightarrow{ar} c$, the response of c according to BEC, does not need to reflect neither a, nor b, as they may simply be not visible to c ($a \not\xrightarrow{vis} c \vee b \not\xrightarrow{vis} c$). Thus, to guarantee conformance to a single global serialization, we must enforce that for any two events $e_1, e_2 \in E$, $e_1 \xrightarrow{ar} e_2 \Leftrightarrow e_1 \xrightarrow{vis} e_2$ (unless e_1 is pending, since a pending operation might be arbitrated before a completed one, yet still be not visible). We express this through the *single order* predicate:

$$\begin{aligned}\text{SINORD} &\stackrel{\text{def}}{=} \exists E' \subseteq rval^{-1}(\nabla) : vis = ar \setminus (E' \times E)\\ \text{SINORD}(l) &\stackrel{\text{def}}{=} \exists E' \subseteq rval^{-1}(\nabla) : vis_L = ar_L \setminus (E' \times E)\\ &\text{where } vis_L = vis \cap (E \times L) \text{ and } ar_L = ar \cap (E \times L).\end{aligned}$$

Note that $rval^{-1}(\nabla)$ represents all pending events, while E' is a subset of these events. Thus, for certain pending events $e_1 \in E'$, $e_1 \xrightarrow{ar} e_2 \Leftrightarrow e_1 \xrightarrow{vis} e_2$ does not need to hold. In the parametrized form, the conformance to the serialization is required only for the events from the set L (but the serialization includes all the events).

In order to capture the eventual *stabilization* of the operation execution order, which happens in AcuteBayou and in ACTs similar to it, we now define two additional correctness criteria that feature SINORD.

Sequential Consistency. Informally, *sequential consistency* (SEQ) [21] guarantees that the system behaves as if all operations were executed sequentially, but in an order that respects the *program order*, i.e., the order in which operations were executed in each session. Hence, SEQ implies RVAL together with SINORD, and additionally, *session arbitration* (SESSARB). SESSARB simply requires that for any two events $e, e' \in E$, if $e \xrightarrow{so} e'$, then $e \xrightarrow{ar} e'$. In the parametrized form we are interested only in the guarantees for events in L, and thus we use $so_L = so \cap (E \times L)$ instead of so (see Section 5.1). SINORD together with SESSARB imply NCC and EV [26], however this does not hold for the parametrized forms of these predicates. Thus, we define SEQ by extending BEC (which explicitly includes EV, NCC and RVAL):

$$\begin{aligned}\text{SESSARB}(l) &\stackrel{\text{def}}{=} so_L \subseteq ar\\ \text{SEQ}(l,\mathcal{F}) &\stackrel{\text{def}}{=} \text{SINORD}(l) \wedge \text{SESSARB}(l) \wedge \text{BEC}(l,\mathcal{F}).\end{aligned}$$

An example abstract execution A_{SEQ} that satisfies SEQ is shown in Fig. 4. According to SEQ, since the $append()$ operations are arbitrated $append(a), append(b)$ (as evidenced by any $read()$ operation that observes both $append()$ operations), any $read()$ can either return ab or a, a non-empty prefix of ab.

Linearizability. The *linearizability* (LIN) [10] correctness condition is similar to SEQ but instead of program order it enforces a stronger requirement called *real-time order*. Informally, a system that is linearizable guarantees that for any operation op' that starts (in real-time) after any operation op ends, op' will observe the effects of op. We formalize LIN using the *real-time order* (RT) predicate, that uses the $rb_L = rb \cap (L \times L)$ relation in its parametrized form:

$$\begin{aligned}\text{RT}(l) &\stackrel{\text{def}}{=} rb_L \subseteq ar\\ \text{LIN}(l,\mathcal{F}) &\stackrel{\text{def}}{=} \text{SINORD}(l) \wedge \text{RT}(l) \wedge \text{BEC}(l,\mathcal{F}).\end{aligned}$$

Note that, SEQ and LIN are incomparable in their parametrized forms. While $\text{LIN}(l,\mathcal{F})$ requires any two events to be arbitrated according to real-time if they both belong to L, $\text{SEQ}(l,\mathcal{F})$ enforces real-time only within the same session, but only one of the events needs to belong to L. By using a stronger definition of $\text{RT}'(l)$ with $rb'_L = rb \cap (E \times L)$, we would force all operations to synchronize, which is incompatible with high availability of weak operations.

An example abstract execution A_{LIN} that satisfies LIN is shown in Fig. 4. According to LIN, since $append(a)$ ended before $append(b)$ started, the operations must be arbitrated $append(a), append(b)$ (as evidenced by any $read()$ operation that observes both $append()$ operations). If some $read()$

operation started after $append(a)$ ended but executed concurrently with $append(b)$ ($append(b)$ would start before $read()$ ended), $read()$ could return either a or ab.

5.6 Correctness of ANNC and AcuteBayou

Having defined BEC, FEC and Lin, we show four formal results: two regarding ANNC and two regarding AcuteBayou. The proofs of all four theorems can be found in Appendix A.5, available in the online supplemental material.

As we have discussed in Section 3.2.2, we are interested in the behaviour of systems, both in a fully asynchronous environment, when timing assumptions are constantly broken (e.g., because of prevalent network partitions), and in a stable one, when sufficient synchrony is available so that consensus eventually terminates. Thus, we consider two kinds of runs: asynchronous and stable.

Theorem 1. *In stable runs ANNC satisfies* BEC$(weak, \mathcal{F}_{NNC})$ $\wedge$ Lin$(strong, \mathcal{F}_{NNC})$.

Theorem 2. *In asynchronous runs ANNC satisfies* BEC$(weak, \mathcal{F}_{NNC})$ *and does not satisfy* Lin$(strong, \mathcal{F}_{NNC})$.

ANNC does not guarantee Lin$(strong, \mathcal{F}_{NNC})$ in asynchronous runs, because strong operations in general (for arbitrary $\mathcal{F}$) cannot be implemented without solving global agreement, and since in asynchronous runs TOB completion is not guaranteed, some of the operations may remain pending. It means that for some $e \in E$, such that $lvl(e) = strong$, $rval(e) = \nabla$, even though it is not allowed by $\mathcal{F}$ (recall from Section 3.2.3 that we consider fair executions).

By satisfying BEC$(weak, \mathcal{F}_{NNC})$, we prove that temporary operation reordering is not possible in ANNC. As we discussed in Section 2.2.2, it is not the case for AcuteBayou. However, we can prove that AcuteBayou satisfies our new correctness criterion FEC$(weak, \mathcal{F})$ (for arbitrary $\mathcal{F}$).

Theorem 3. *In stable runs AcuteBayou satisfies* FEC$(weak, \mathcal{F})$ $\wedge$ Lin$(strong, \mathcal{F})$ *for any arbitrary ACT specification* $(\mathcal{F}, lvlmap)$.

Theorem 4. *In asynchronous runs AcuteBayou satisfies* FEC$(weak, \mathcal{F})$ *and it does not satisfy* Lin$(strong, \mathcal{F})$ *for any arbitrary ACT specification* $(\mathcal{F}, lvlmap)$.

The observation that some undesired anomalies are not inherent to all ACTs leads to interesting questions that we plan to investigate more closely in the future, e.g., what are the common characteristics of the replicated data types with mixed-consistency semantics that can be implemented as ACTs that are free of temporary operation reordering.

6 Impossibility

Now we proceed to our central contribution: we show that there exists an ACT specification for which it is impossible to propose an ACT implementation that avoids temporary operation reordering.

If a mixed-consistency ACT that implements some replicated data type $\mathcal{F}$ could avoid temporary operation reordering, it would mean that it ensures BEC for weak operations and also provides at least some criterion based on SinOrd for strong operations (to ensure a global serialization of all operations). Hence we state our main theorem:

Theorem 5. *There exists an ACT specification* $(\mathcal{F}, lvlmap)$, *for which there does not exist an implementation that satisfies* SinOrd$(strong)$ $\wedge$ BEC$(strong, \mathcal{F})$ *in stable runs, and* BEC $(weak, \mathcal{F})$ *in both asynchronous and stable runs.*

To prove the theorem, we take $\mathcal{F}_{seq}$ (defined in Fig. 3) as an example replicated data type specification $\mathcal{F}$. We consider an ACT specification, which features $append$ and $read$ operations in both consistency levels, $weak$, and $strong$. Thus, $(\mathcal{F},\ lvlmap) = (\mathcal{F}_{seq}, lvlmap_{seq})$, where $lvlmap_{seq}(append) = lvlmap_{seq}(read) = \{weak, strong\}$.

Let us begin with an observation. Whenever any ACT implementation of $(\mathcal{F}_{seq}, lvlmap_{seq})$ that satisfies BEC$(weak, \mathcal{F}_{seq})$ in asynchronous runs, executes a weak $append$ operation, it has to RB-cast some message m. Since the implementation satisfies EV (through BEC$(weak, \mathcal{F}_{seq})$) we know that all replicas have to be informed about the invocation of $append$. The replica executing the $append$ operation may not postpone sending the message until some other invocation happens, because all the subsequent operation invocations on the replica may be operations, which do not grant the replica the right to send messages (e.g., RO operations, by the invisible reads requirement). Moreover, the replica may not depend on TOB-cast messages, because in asynchronous runs they are not guaranteed to be delivered to other replicas.[5] Thus, a message must be RB-cast. Since replicas cannot distinguish between asynchronous and stable runs, the same observation also holds for stable runs. We utilize this fact in our proof by considering asynchronous and stable executions and establishing certain invariants which need to hold in both kinds of runs.

We conduct the proof by contradiction using a specially constructed execution, in which a replica that executes a strong operation has to return a value without consulting all replicas. Thus, we consider an ACT implementation of $(\mathcal{F}_{seq}, lvlmap_{seq})$ that satisfies BEC$(weak, \mathcal{F}_{seq})$ in asynchronous runs, and BEC$(weak, \mathcal{F}_{seq})$ $\wedge$ SinOrd$(strong)$ $\wedge$ BEC$(strong, \mathcal{F}_{seq})$ in both the asynchronous and stable runs (see definition of $\mathcal{F}_{seq}$ in Fig. 3).

Proof. We give a proof for a system of three replicas R_1, R_2 and R_3. We begin with an empty execution represented by a history $H = (E, op, rval, rb, ss, lvl)$, which we will extend in subsequent steps. Initially all replicas are separated by a temporary network partition, which means that the messages broadcast by the replicas do not propagate (however, eventually they will be delivered once the partition heals). A weak $append(a)$ operation is invoked on R_1 in the event e_a and a weak $append(b)$ operation is invoked on R_2 in the event e_b. By input-driven processing and highly available weak operations both replicas return responses for the operations and become passive afterwards. Let $msgs_a^{RB}$ and $msgs_b^{RB}$ denote the set of messages RB-cast by, respectively, R_1 and R_2, until this point. Let $msgs_a^{TOB}$ and $msgs_b^{TOB}$ denote the set of messages TOB-cast by, respectively, R_1 and R_2, until this point. R_1 RB-delivers messages from the set $msgs_a^{RB}$, while R_2 RB-delivers messages from the set $msgs_b^{RB}$. No other messages are delivered by either replica

5. A replica may TOB-cast some messages due to the invocation of a weak $append$ operation, but its correctness cannot depend on their delivery.

(due to the temporary network partition). Subsequently replicas become passive (if $msgs_a^{TOB} \neq \emptyset$ or $msgs_b^{TOB} \neq \emptyset$, then these messages remain pending).

Consider another execution represented by history $H' = (E', op', rval', rb', ss', lvl')$ in which the network partition heals, and R_1 RB-delivers all messages in the set $msgs_b^{RB}$, R_2 RB-delivers all messages in the set $msgs_a^{RB}$, R_3 RB-delivers all messages in both the sets $msgs_a^{RB}$ and $msgs_b^{RB}$, and then a weak $read$ operation is invoked on R_1 in the event e'_c and a weak $read$ operation is invoked on R_2 in the event e'_d. By invisible reads and highly available operations, both replicas remain passive and immediately return a response.

Claim 1. *$rval'(e'_c) = rval'(e'_d) = v$, and $v = ab$ or $v = ba$.*

Proof. We extend H' with infinitely many weak $read$ invocations on each replica, in events e'_k, for $k \geq 1$. Similarly to e'_c and e'_d, the $read$ operations invoked in each e'_k return immediately and leave the replicas in the unmodified passive state. Since none of the $read$ operations generate any new messages, H' represents a fair infinite execution that satisfies all network properties of an asynchronous run. Then, by our base assumption, there exists an abstract execution $A' = (H', vis', ar', par')$, such that $A' \models \text{BEC}(weak, \mathcal{F}_{seq})$.

Because R_1 and R_2 remain in the same state since the execution of e'_c and e'_d, respectively, each $read$ operation invoked in e'_k on these replicas, returns the same response as e'_c or e'_d, depending on which replica the given event was executed on. By $\text{EV}(weak)$, the two updating events e_a and e_b have to be both observed by infinitely many of the e'_k events. Let e'_p be one such event executed on R_1 and e'_q be one such event executed on R_2, then $(e_a \xrightarrow{vis'} e'_p \wedge e_a \xrightarrow{vis'} e'_q \wedge e_b \xrightarrow{vis'} e'_p \wedge e_b \xrightarrow{vis'} e'_q)$. There is either: $e_a \xrightarrow{ar'} e_b$, or $e_b \xrightarrow{ar'} e_a$. Now, by the definition of read-only operations we can exclude the RO operations from the context of any operation without affecting the return value of all operations. Thus $\mathcal{F}_{seq}(read(), context(A', e'_p)) = \mathcal{F}_{seq}(read(), context(A', e'_q)) = v'$ for some v'. Because of $\text{RVAL}(weak, \mathcal{F}_{seq})$, $rval'(e'_p) = v' = rval'(e'_q)$. Therefore, all $read$ operations in H' return the same value v', including the earliest ones e'_c and e'_d, which means that $v = v'$. By the definition of $\mathcal{F}_{seq}$, either $v = ab$ or $v = ba$ (depending on whether $e_a \xrightarrow{ar'} e_b$, or $e_b \xrightarrow{ar'} e_a$). □

Without loss of generality, let us assume that v obtained in the history H' equals ab. Let us return to our main history H. We extend it similarly to the way we extended H', but we do not allow the network partition to heal completely. Instead, we just let $msgs_b^{RB}$ to reach R_1, which RB-delivers them exactly as in H'. Then, similarly to H', in H we invoke a weak $read$ operation on R_1 in an event e_r.

Claim 2. *In history H, $rval(e_r) = v = ab$.*

Proof. Since R_1 executes exactly the same steps in both H and H' up to the invocation of e_r and e'_c, respectively, and because replicas are deterministic, the current state of R_1 when executing e_r must be the same as it was in H' during the execution of e'_c. Thus, the return values of both operations are equal. □

Consider yet another execution represented by history $H'' = (E'', op'', rval'', rb'', ss'', lvl'')$ which is obtained from our main execution H by removing any steps executed by R_1. The events executed on R_2 and R_3 remain unchanged, since the replicas were all the time separated by a network partition, and no messages from R_1 reached neither R_2 nor R_3. We let the network partition heal. R_1 RB-delivers messages from the set $msgs_b^{RB}$, R_3 RB-delivers messages from both the sets $msgs_a^{RB}$ and $msgs_b^{RB}$, all replicas TOB-deliver messages from the set $msgs_b^{TOB}$, and afterward all replicas become passive.

We now extend H'' by infinitely many times applying the following procedure (for k from 1 to infinity):

1) invoke a *strong* $read$ on R_2 in the event e''_{3k},
2) let R_2 execute its steps until it becomes passive,
3) on each replica, RB-deliver and TOB-deliver all messages, respectively, RB-cast or TOB-cast, by R_2 in step 2,
4) let each replica reach a passive state,
5) invoke a *weak* $read$ on R_1 in the event e''_{3k+1},
6) invoke a *weak* $read$ on R_3 in the event e''_{3k+2}.

The resulting execution is fair and satisfies all the network properties of a stable run. Note that the strong $read$ operations executed on R_2 are not restricted by invisible reads and thus may freely change the state of R_2. Moreover, they can cause R_2 to RB-cast and TOB-cast messages. On the other hand, the weak $read$ operations executed on R_1 and R_3 are always executed on a passive state, and leave the replica in the same state. Moreover, R_1 and R_3 do not RB-cast, nor TOB-cast any messages. By non-blocking strong operations no strong $read$ operation may be pending in H''. This is so, because for each k, by step 4, there is no pending message not yet TOB-delivered on R_2, and R_2 is in a passive state.

Claim 3. *There exists an event $e''_x \in E''$, with $x = 3k$ for some natural k, such that $rval''(e''_x) = b$.*

Proof. By our base assumption, there exists an abstract execution $A'' = (H'', vis'', ar'', par'')$, such that $A'' \models \text{SINORD}(strong) \wedge \text{BEC}(strong, \mathcal{F}_{seq})$. Then, for each k, by $\text{RVAL}(strong, \mathcal{F}_{seq})$, $rval''(e''_{3k}) = \mathcal{F}_{seq}(read(), context(A'', e''_{3k}))$. Moreover, because of $\text{EV}(strong)$, e_b needs to be observed from some point on by every e''_{3k}. Thus, we let $e_b \xrightarrow{vis''} e''_x$. Since e_b is the only $append$ operation visible to e''_x (there are no other $append$ operations in A''), by definition of $\mathcal{F}_{seq}$, $rval''(e''_x) = b$. □

Let us return to our main history H. Note that, when we restrict H and H'' only to events on R_2, H constitutes a prefix of H''. Moreover, the state of R_2 at the end of H is the same as in H'' just before TOB-delivering messages from the set $msgs_b^{TOB}$ (if any) and executing the first strong $read$ operation. First, we allow the partition between R_2 and R_3 to heal (but R_1 remains disconnected). Then, we extend H in a few steps. We let R_3 RB-deliver messages from the set $msgs_b^{RB}$. Next, we TOB-deliver on R_2 and R_3 the messages from the

set $msgs_b^{TOB}$. Finally, we extend H with steps executed on R_2 and R_3 generated using the repeated procedure for H'', for k from 1 to $\frac{x}{3}$. We can freely omit the steps executed on R_1, since none of them influenced in any way the other replicas (neither R_2, nor R_3, RB-deliver, nor TOB-deliver any message from R_1). Thus, there exists an event $e_x \in E$ executed on R_2, an equivalent of the e''_x event from H'', such that $op(e_x) = read()$, $lvl(e_x) = strong$ and $rval\,(e_x) = b$.

Finally, we allow the network partition to heal completely. R_2 and R_3 RB-deliver messages from the set $msgs_a^{RB}$, and R_1 RB-delivers and TOB-delivers any outstanding messages generated by R_2 (naturally, R_1 TOB-delivers messages in the same order as R_2 and R_3 did). Then, we let the replicas reach a passive state, and in order to make our constructed execution fair, we extend it with infinitely many weak $read$ operations as we did with H'. By our base assumption, there exists an abstract execution $A = (H, vis, ar, par)$, such that $A \models \text{BEC}(weak, \mathcal{F}_{seq}) \wedge \text{SinOrd}(strong) \wedge \text{BEC}(strong, \mathcal{F}_{seq})$. There are only two $append$ operations invoked in A in the events e_a and e_b. Since $rval(e_r) = ab$ (which we have established in Claim 2), by $\text{RVal}(weak, \mathcal{F}_{seq})$ and the definition of $\mathcal{F}_{seq}$, it can be only that $e_a \xrightarrow{ar} e_b$. We also know that $rval(e_x) = b$ (e_x is a strong $read$ operation executed on R_2), which means that $e_b \xrightarrow{vis} e_x \wedge e_a \not\xrightarrow{vis} e_x$. By $\text{SinOrd}(strong)$, $e_b \xrightarrow{ar} e_x \wedge e_a \not\xrightarrow{ar} e_x$, and thus $e_x \xrightarrow{ar} e_a$. Therefore, a cycle forms in the total order relation ar: $e_a \xrightarrow{ar} e_b \xrightarrow{ar} e_x \xrightarrow{ar} e_a$, a contradiction. This concludes the proof. □

Since from Theorem 5 we know that there exists an ACT specification $(\mathcal{F}, lvlmap)$ for which we cannot propose (even a specialized) implementation that satisfies $\text{BEC}(weak, \mathcal{F})$, we can formulate a more general result about generic ACTs:

Corollary 1. *There does not exist a generic implementation that for an arbitrary ACT specification $(\mathcal{F}, lvlmap)$ satisfies $\text{SinOrd}(strong) \wedge \text{BEC}(strong, \mathcal{F})$ in stable runs, and $\text{BEC}(weak, \mathcal{F})$ both in asynchronous, and in stable runs.*

Theorem 5 shows that it is impossible to devise a system similar to AcuteBayou (for arbitrary $\mathcal{F}$) that never admits temporary operation reordering (so satisfies $\text{BEC}(weak, \mathcal{F})$ instead of $\text{FEC}(weak, \mathcal{F})$). Hence, admitting temporary operation reordering is the inherent cost of mixing eventual and strong consistency when we make no assumptions about the semantics of $\mathcal{F}$. Naturally, for certain replicated data types, such as $\mathcal{F}_{NNC}$, achieving both $\text{BEC}(weak, \mathcal{F})$ and $\text{Lin}(strong, \mathcal{F})$ is possible, as we show with ANNC.

In the next section we discuss several approaches that avoid temporary operation reordering, albeit at the cost of compromising fault-tolerance (e.g., by requiring all replicas to be operational), or sacrificing high availability (e.g., by forcing replicas to synchronize on weak operations).

7 Related Work

7.1 Symmetric Models With Strong Operations Blocking Upon a Single Crash

We start with *symmetric* mixed-consistency models, in which all replicas can process both weak and strong operations and communicate directly with each other (thus enabling processing of weak operations within network partitions), but either do not enable fully-fledged strong operations (there is no stabilization of operation execution order) or require all replicas to synchronize in order for a strong operation to complete. In turn, the way these models bind the execution of weak and strong operations can be understood as an asymmetric (1–n) variant of quorum-based synchronization. Hence, unlike in ACTs, strong operations cannot complete if even a single replica cannot respond (due to a machine or network failure), which is a major limitation.

Lazy Replication [15] features three operation levels: causal, forced (totally ordered with respect to one another) and immediate (totally ordered with respect to all other operations). In this approach, it is possible that two replicas execute a causal operation op_c and a forced operation op_f in different orders. Since op_c is required to commute with op_f, replicas will converge to the same state. However, the user is never certain that even after the completion of op_f, on some other replica no weaker operation op'_c will be executed prior to op_f. Hence the guarantees provided by forced operations are inadequate for certain use cases, which require write stabilization, e.g., an auction system [4] (see also Section 1). On the other hand, immediate operations offer stronger guarantees, but their implementation is based on three-phase commit [54], and thus requires all replicas to vote in order to proceed.

RedBlue consistency [14] extends Lazy Replication (with *blue* and *red* operations corresponding to the causal and forced ones), by allowing operations to be split into (side-effect free) *generator* and (globally commutative) *shadow* operations. This greatly increases the number of operations which commute, but red operations still do not guarantee write stabilization. To overcome this limitation, RedBlue consistency was extended with programmer-defined *partial order restrictions* over operations [17]. The proposed implementation, *Olisipo*, relies on a counter-based system to synchronize conflicting operations. Synchronization can be either symmetric (all potentially conflicting pairs of operations must synchronize, which means that weak operations are not highly available any more) or asymmetric (all replicas must be operational for strong operations to complete).

The formal framework of [16] can be used to express various consistency guarantees, including those of Lazy Replication and RedBlue consistency, but not as strong as, e.g., linearizability. Conflicts resulting from operations that do not commute are modelled through a set of tokens. On the other hand, in *explicit consistency* [55], stronger consistency guarantees are modelled through application-level invariants and can be achieved using multi-level locks (similar to readers-writer locks from shared memory).

All models mentioned so far assume causal consistency (CC) as the base-line consistency criterion and thus do not account for weaker consistency guarantees, such as FEC or BEC, as our framework. CC is argued to be costly to ensure in real-life [18], which makes our approach more general.

Finally, the model in [27] is similar to ours but treats strong operations as fences (barriers). It means that all replicas must vote in order for a strong operation to complete.

Temporary operation reordering is not possible in the models discussed so far. It is because they are either state-based (and thus their formalism abstracts away from the

operation return values which clients observe and interpret) and feature no write stabilization, or they require all replicas to vote in order to process strong operations (as explained in Section 2.2.4).

7.2 Symmetric Bayou-Like Models

In Section 2 we have already discussed the relationship between the seminal Bayou protocol [23] and ACTs.

In *eventually-serializable data service (ESDS)* [56], operations are executed speculatively before they are stabilized, similarly to Bayou. However, ESDS additionally allows a programmer to attach to an operation an arbitrary causal context that must be satisfied before the operation is executed. Zeno [57] is similar to Bayou but has been designed to tolerate Byzantine failures.

All three systems (Bayou, ESDS, Zeno) are eventually consistent, but ensure that eventually there exists a single serialization of all operations, and the client may wait for a notification that certain operation was stabilized. Since these systems enable an execution of arbitrarily complex operations (as ACTs), they admit temporary operation reordering.

Several researchers attempted a formal analysis of the guarantees provided by Bayou or systems similar to it. E.g., the authors of Zeno [57] describe its behaviour using I/O automata. In [58] the authors analyse Bayou and explain it through a formal framework that is tailored to Bayou. Both of these approaches are not as general as ours and do not enable comparison of the guarantees provided by other systems. Finally, the framework in [50] enables reasoning about eventually consistent systems that enable speculative executions and rollbacks and so also AcuteBayou. However, the framework does not formalize strong consistency models, which means it is not suitable for our purposes.

7.3 Asymmetric Models With Cloud as a Proxy

Contrary to our approach, the work described below assumes an *asymmetric* model in which external clients maintain local copies of primary objects that reside in a centralized (replicated) system, referred to as *the cloud*. Clients perform weak operations on local copies and only synchronize with the cloud lazily or to complete strong operations. Since the cloud functions as a communication proxy between the clients, when it is is unavailable (e.g., due to failures of majority of replicas or a partition), clients cannot observe even each others new weak operations. Hence, this approach is less flexible than ours. However, since the cloud serves the role of a single *source of truth*, conflicts between concurrent updates can be resolved before they are propagated to the clients, so temporary operation reordering is not possible.

In *cloud types* [12], clients issue operations on replicated objects stored in the *local revision* and occasionally synchronize with the *main revision* stored in the cloud, in a way similar as in version control systems. The synchronization happens either eagerly or lazily, depending on the used mode of synchronization. The authors use *revision consistency* [59] as the target correctness criterion. In a subsequent work [13] a *global sequence protocol (GSP)* was introduced, which refines the programming model of cloud types, and replaces revision consistency with an abstract data model, as revisions and revision consistency were deemed too complicated for non-expert users. *Global sequence consistency (GSC)* [60] is a consistency model that generalizes GSP and a few other approaches that assume external clients that either eagerly or lazily push or pull data from the cloud.

7.4 Asymmetric Master-Slave Models

There are systems which relax strong consistency by allowing clients to read stale data, either on demand (the client may forgo recency guarantees by choosing a weak consistency level for an operation), or depending on the replica localization (in a geo-replicated system the client accessing the nearest replica can read stale data that are pertinent to a different region). However, in such systems all updating operations (including the weak ones) must pass through the primary server designated for each particular data item. Thus, similarly to the *asymmetric, cloud as a proxy* models, in this approach weak operations are not freely disseminated among the replicas. Since all updates (of a concrete data item) are serialized by the primary, temporary operation reordering is not possible.

Examples of systems which follow this design and allow users to select an appropriate consistency level include PNUTS [61], Pileus [62], and also the widely popular contemporary cloud data stores, such as AmazonDB [5] and CosmosDB [6]. Systems that guarantee strong consistency within a single site and causal consistency between sites include Walter [63], COPS [48], Eiger [64] and Occult [65].

7.5 Other Approaches

Certain eventually consistent NoSQL data stores enable strongly consistent operations on-demand. E.g., Riak allows some data to be kept in *strongly consistent buckets* [8], which is a namespace completely separate from the one used for data accessed in a regular, eventually-consistent way. Apache Cassandra provides compare-and-set-like operations, called *lightweight transactions (LWTs)* [7], which can be executed on any data, but the user is forbidden from executing weakly consistent updates on that data at the same time. Concurrent updates and LWTs result in undefined behaviour [9], which means that mixed-consistency semantics of LWTs can be considered broken.

In Lynx [66] and Salt [67] mixed-consistency transactions are translated into a chain of subtransactions, each committed at a different primary site. Thus such transactions can block or raise an error if a specific site is unavailable.

Observable Atomic Consistency Protocol [68] is symmetric and supports strong operations via synchronization based on distributed consensus. However, unlike in ACTs, weak operations block when any strong operation is in progress, thus are not highly available.

Systems based on escrow techniques [69] enable strongly consistent operations to be executed simultaneously with weak operations, albeit in a non-fault-tolerant manner or by enforcing strong synchronization, at least within a single data center, also for weak operations [70].

Recently some work has been published on the programming language perspective of mixed-consistency semantics.

Since this research is not directly related to our work, we briefly discuss only a few papers. *Correctables* [71] are abstractions similar to futures, that can be used to obtain multiple, incremental views on the operation return value (e.g., a result of a speculative execution of the operation and then the final return value). Correctables are used as an interface for the modified variants of Apache Cassandra and ZooKeeper [72] (a strongly consistent system). In *MixT* [73] each data item is marked with a consistency level that will be used upon access. A transaction that accesses data marked with different consistency levels is split into multiple independently executed subtransactions, each corresponding to a concrete consistency level. The compilation-time code-level verification ensures that operations performed on data marked with weaker consistency levels do not influence the operations on data marked with stronger consistency levels. Understandably, the execution of a mixed-level transaction can be blocking. Finally, in [74] the authors advocate the use of the release-acquire semantics (adapted from low-level concurrent programming) and propose *Kite*, a mixed-consistency key-value store utilizing this consistency model. In Kite weak read operations occasionally require inter-replica synchronization and block on network communication, thus they are not highly available.

7.6 Comparison of FEC With Other Correctness Criteria

In Section 5 we discussed the relation of FEC with BEC [26], [27] by Burckhardt *et al*. FEC can be considered a relaxation of the Consistent Prefix [26] property (later also described as Monotonic Prefix Consistency [25]). In the latter properties events become visible to subsequent read operations only when their final arbitration is established.

In the framework of [24] FEC operations' ordering can be expressed as Capricious TOE (however, note that the model of [24] does not account for mixed consistency approaches). Thus, FEC bears some similarities with non-permissioned blockchains [75].

8 Conclusion

In this paper we defined *acute cloud types*, a class of replicated systems that aim at seamless mixing of eventual and strong consistency. ACTs are primarily designed to execute client-submitted operations in a highly available, eventually-consistent fashion, similarly to CRDTs. However, for tasks that cannot be performed in that way, ACTs at the same time support operations that require some form of distributed consensus-based synchronization.

We defined ACTs and the guarantees they provide in our novel framework which is suited for modeling mixed-consistency systems. We also proposed a new consistency criterion called *fluctuating eventual consistency*, which captures a common trait of many ACTs, namely *temporary operation reordering*. Interestingly, temporary operation reordering appears neither in systems that are purely eventually consistent (e.g., NoSQL data stores) nor purely strongly consistent (e.g., traditional DBMS). Moreover, it is not necessarily present in all ACTs, but as we formally prove, it cannot be avoided in ACTs that feature arbitrarily complex (but deterministic) semantics (e.g., arbitrary SQL transactions).

Acknowledgments

This work was supported by the Foundation for Polish Science, within the TEAM programme co-financed by European Union under European Regional Development Fund, under Grant No. POIR.04.04.00-00-5C5B/17-00. The work of Maciej Kokociński and Tadeusz Kobus was supported in part by the Polish National Science Centre under Grant No. DEC-2012/07/B/ST6/01230 and in part by the internal funds of the Faculty of Computing, Poznan University of Technology.

References

[1] G. DeCandia *et al*., "Dynamo: Amazon's highly available key-value store," *SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.
[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proc. Symp. Self-Stabilizing Syst.*, 2011, pp. 386–400.
[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," Inria–Centre Paris-Rocquencourt, Paris, France, *Tech. Rep. 7506*, 2011.
[4] N. M. Preguiça, C. Baquero, and M. Shapiro, "Conflict-free replicated data types," 2018, *arXiv:1805.06358*.
[5] Amazon DynamoDB documentation, "Consistent reads in DynamoDB," 2019. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html.
[6] Microsoft Azure documentation, "Consistency levels in Cosmos DB," 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels
[7] Apache Cassandra documentation, "Light weight transactions in Cassandra," 2019. [Online]. Available: https://docs.datastax.com/en/cql/3.3/cql/cql_using/useInsertLWT.html
[8] Basho documentation, "Consistency levels in Riak," 2019. [Online]. Available: https://docs.basho.com/riak/kv/2.2.3/developing/app-guide/strong-consistency
[9] Apache Cassandra Issues (Jira), "Mixing LWT and non-LWT operations can result in an LWT operation being acknowledged but not applied," 2019. [Online]. Available: https://jira.apache.org/jira/browse/CASSANDRA-11000
[10] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.
[11] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.
[12] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood, "Cloud types for eventual consistency," in *Proc. Eur. Conf. Object-Oriented Program.*, Jun. 2012, pp. 283–307.
[13] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich, "Global sequence protocol: A robust abstraction for replicated shared state," in *Proc. Eur. Conf. Object-Oriented Program.*, Jul. 2015, pp. 568–590.
[14] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, Oct. 2012, pp. 265–278.
[15] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, pp. 360–391, Nov. 1992.
[16] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro, "'Cause I'm strong enough: Reasoning about consistency choices in distributed systems," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, Jan. 2016, pp. 371–384.
[17] C. Li, N. Preguiça, and R. Rodrigues, "Fine-grained consistency for geo-replicated systems," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, Jul. 2018, pp. 359–371.
[18] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proc. Third ACM Symp. Cloud Comput.*, Oct. 2012, pp. 1–7.

[19] F. Houshmand and M. Lesani, "Hamsaz: Replication coordination analysis and synthesis," *Proc. ACM Program. Lang.*, vol. 3, Jan. 2019, Art. no. 74.
[20] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.
[21] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.
[22] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, Jun. 1979.
[23] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 29, pp. 172–182, Dec. 1995.
[24] M. Shapiro, M. S. Ardekani, and G. Petri, "Consistency in 3D," in *Proc. Int. Conf. on Concurrency Theory*, Aug. 2016, vol. 59, pp. 1–15.
[25] A. Girault, G. Gößler, R. Guerraoui, J. Hamza, and D. Seredinschi, "Monotonic prefix consistency in distributed systems," in *Proc. Int. Conf. Formal Techn. Distrib. Objects Components Syst.*, Jun. 2018, pp. 41–57.
[26] S. Burckhardt, "Principles of eventual consistency," *Foundations Trends Program. Lang.*, vol. 1, no. 1–2, pp. 1–150, Oct. 2014.
[27] S. Burckhardt, A. Gotsman, and H. Yang, "Understanding eventual consistency," Microsoft Research, Redmond, WA, USA, *Tech. Rep. MSR-TR-2013–39*, Mar. 2013.
[28] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.
[29] M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Brief announcement: On mixing eventual and strong consistency: Bayou revisited," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2019, pp. 458–460.
[30] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Program*. Berlin, Germany: Springer, 2011.
[31] F. Pedone, R. Guerraoui, and A. Schiper, "Exploiting atomic broadcast in replicated databases," in *Proc. Eur. Conf. Parallel Process.*, Sep. 1998, pp. 513–520.
[32] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[33] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication - Theory and Practice*. Berlin, Germany: Springer, 2010.
[34] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2015, pp. 1–10.
[35] R. Hansdah and L. Patnaik, " Update serializability in locking," in *Proc. Int. Conf. Database Theory*, Sep. 1986, pp. 171–185.
[36] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *Proc. IEEE 32nd Int. Conf. Distrib. Comput. Syst.*, Jun. 2012, pp. 455–465.
[37] P. A., Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley, 1987.
[38] G. V. Chockler and A. Gotsman, "Multi-shot distributed transaction commit," in *Proc. Int. Symp. Distrib. Comput.*, Oct. 2018, pp. 14:1–14:18.
[39] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.
[40] P. Verissimo and C. Almeida, "Quasi-synchronism: A step away from the traditional fault-tolerant real-time system models," *IEEE TCOS Bulletin*, vol. 7, no. 4, pp. 35–39, Dec. 1995.
[41] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, no. 4, pp. 685–722, Jul. 1996.
[42] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 141–155, Jan. 2017.
[43] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 2, pp. 125–143, Mar. 1977.
[44] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distrib. Comput.*, vol. 2, no. 3, pp. 117–126, Sep. 1987.
[45] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2013, pp. 761–772.
[46] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, "Replicated data types: Specification, verification, optimality," in *Proc. ACM SIGPLAN Notices*, vol. 49, pp. 271–284, Jan. 2014.
[47] N. Preguiça *et al.*, "SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine," 2014, *arXiv:1310.3107*.
[48] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in *Proc. ACM Symp. Operating Syst. Princ.*, Oct. 2011, pp. 401–416.
[49] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Closing the performance gap between causal consistency and eventual consistency," in *Proc. 1st Workshop Princ. Pract. Eventual Consistency*, Apr. 2014, pp. 1–5.
[50] A. Bouajjani, C. Enea, and J. Hamza, "Verifying eventual consistency of optimistic replication systems," in *Proc. Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, Jan. 2014, pp. 285–296.
[51] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surv.*, vol. 49, no. 1, pp. 19:1–19:34, Jun. 2016.
[52] T. L. Greenough, "Representation and enumeration of interval orders," Ph.D. dissertation, Dartmouth College, Hanover, NH, USA, 1976.
[53] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session guarantees for weakly consistent replicated data," in *Proc. Int. Conf. Parallel Distrib. Inf. Syst.*, 1994, pp. 140–150.
[54] D. Skeen, "Nonblocking commit protocols," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1981, pp. 133–142.
[55] V. Balegas *et al.*, "Putting consistency back into eventual consistency," in *Proc. 10th Eur. Conf. Comput. Syst.*, Apr. 2015, pp. 1–16.
[56] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman, "Eventually-serializable data services," in *Proc. ACM Symp. Princ. Distrib. Comput.*, May 1996, pp. 300–309.
[57] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis, "Zeno: Eventually consistent byzantine-fault tolerance," in *Proc. 6th USENIX Symp. Networked Syst. Des. Implementation*, Apr. 2009, pp. 169–184.
[58] A.-M. Bosneag and M. Brockmeyer, "A formal model for eventual consistency semantics," in *Proc. Int. Conf. Parallel Distrib. Comput. Syst.*, Jan. 2002, pp. 204–209.
[59] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv, "Eventually consistent transactions," in *Proc. Eur. Symp. Program.*, Mar. 2012, pp. 67–86.
[60] A. Gotsman and S. Burckhardt, "Consistency models with global operation sequencing and their composition," in *Proc. Int. Symp. Distrib. Comput.*, Oct. 2017, pp. 23:1–23:16.
[61] B. F. Cooper *et al.*, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endowment*, vol. 1, pp. 1277–1288, 2008.
[62] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh , "Consistency-based service level agreements for cloud storage," in *Proc. ACM Symp. Operating Syst. Princ.*, Nov. 2013, pp. 309–324.
[63] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proc. ACM Symp. Operating Syst. Princ.*, 2011, pp. 385–400.
[64] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proc. 10th Symp. Networked Syst. Des. Implementation*, Apr. 2013, pp. 313–328.
[65] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! Scalable causal consistency with no slowdown cascades," in *Proc. 14th Symp. Networked Syst. Des. Implementation*, Mar. 2017, pp. 453–468.
[66] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, "Transaction chains: Achieving serializability with low latency in geo-distributed storage systems," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, Nov. 2013, pp. 276–291.
[67] C. Xie *et al.*, "Salt: Combining ACID and BASE in a distributed database," in *Proc. 11th Symp. Operating Syst. Des. Implementation*, Oct. 2014, pp. 495–509.
[68] X. Zhao and P. Haller, "Replicated data types that unify eventual consistency and observable atomic consistency," *J. Logical Algebraic Methods Program.*, vol. 114, 2020, Art. 100561.
[69] P. E. O'Neil, "The escrow transactional method," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 405–430, Dec. 1986.

[70] V. Balegas *et al.*, "Extending eventually consistent cloud databases for enforcing numeric invariants," in *Proc. 34th Symp. Reliable Distrib. Syst.*, Sep. 2015, pp. 31–36.
[71] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Incremental consistency guarantees for replicated objects," in *Proc. Symp. Operating Syst. Des. Implementation*, Nov. 2016, pp. 169–184.
[72] Apache ZooKeeper documentation, "Zookeeper," 2019. [Online]. Available: https://zookeeper.apache.org
[73] M. Milano and A. C. Myers, "MixT: A language for mixing consistency in geodistributed transactions," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 226–241, 2018.
[74] V. Gavrielatos, A. Katsarakis, V. Nagarajan, B. Grot, and A. Joshi, "Kite: Efficient and available release consistency for the datacenter," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2020, pp. 1–16.
[75] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[76] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2000, p. 7.
[77] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.
[78] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[79] N. M. Preguiça, C. Baquero, P. S. Almeida, V. Fonte, and R. Gonçalves, "Dotted version vectors: Logical clocks for optimistic replication," 2010, *arXiv:1011.5808*.

Maciej Kokociński is currently working toward the PhD degree in computer science with the Institute of Computing Science, Poznan University of Technology, Poland. He is currently a research assistant with the Institute of Computing Science, Poznan University of Technology. He was a summer intern with Microsoft in Redmond and collaborated on research projects for Egnyte. His research interests include theory of distributed systems and transactional memory, and the design of efficient in-memory data stores.

Tadeusz Kobus received the PhD degree in computer science in 2017 from Poznan University of Technology, Poland. He is currently an assistant professor with the Institute of Computing Science, Poznan University of Technology. He was an intern with IBM T. J. Watson Research Center and worked on research projects for Egnyte. His research interests include fault-tolerant distributed algorithms, concurrent data structures, and, most recently, nonvolatile memory.

Paweł T. Wojciechowski received the Habilitation degree from Poznan University of Technology, Poland, in 2008, and the PhD degree in computer science from the University of Cambridge, in 2000. From 2001 to 2005, he was a postdoctoral researcher with the School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland. He is currently an associate professor with the Institute of Computing Science, Poznan University of Technology. He has led many research projects and coauthored dozens of papers. His research interests span topics in concurrency, distributed computing, and programming languages.