Maciej Kokociński

Correctness of Highly-Available Eventually-Consistent Replicated Systems

Doctoral Dissertation

Submitted to the Discipline Council
of Information and Communication Technology
of Poznań University of Technology

Thesis Advisor: Paweł T. Wojciechowski, Ph. D., Dr. Habil., Assoc. Prof.

Poznań · 2022

Maciej Kokociński

Poprawność Wysoko Dostępnych Ostatecznie Spójnych Systemów Zreplikowanych

Rozprawa doktorska

Przedłożono Radzie Dyscypliny
Informatyka Techniczna i Telekomunikacja
Politechniki Poznańskiej

Promotor: dr hab. inż. Paweł T. Wojciechowski, prof. PP

Poznań · 2022

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy in Computing Science.

Maciej Kokociński
Distributed Systems Research Group
Faculty of Computing and Telecommunications
Institute of Computing Science
Poznań University of Technology
maciej.kokocinski@cs.put.edu.pl

Typeset by the author in LaTeX.

Institute of Computing Science
Poznań University of Technology
Piotrowo 2, 60–965 Poznań, Poland
http://www.cs.put.poznan.pl

The research presented in this dissertation was partially funded from National Science Centre (NCN) funds granted by decision No. DEC-2012/07/B/ST6/01230, and from Foundation for Polish Science (FNP) funds, within the TEAM programme co-financed by the European Union under the European Regional Development Fund, granted by decision No. POIR.04.04.00-00-5C5B/17-00.

To my dearest Henryka.

Abstract

Global services that lie at the heart of today's Internet must remain operational at all times. To avoid downtimes caused by machine and network failures, highly available systems, such as NoSQL data stores, are utilized. Due to our increasing reliance on such systems, the study and verification of their correctness is of the utmost concern.

In this dissertation, we approach the problem of correctness of highly available systems from the theoretical point of view. We study the consistency and progress guarantees achievable under various assumptions. In particular, we investigate mixed-consistency systems, in which weakly consistent highly available operations are mixed with strongly consistent but not highly available ones. We also closely examine the behaviour of highly available systems in the presence of specific types of failures.

We devise new formal frameworks to reason about highly available systems, that allow us to uncover inherent limitations and tradeoffs in their correctness guarantees. We show that in a mixed-consistency system it is impossible to combine the best features of eventually consistent and strongly consistent systems without inflicting a penalty on the correctness guarantees. We also show that certain liveness guarantees are unachievable under specific failure models.

We formally identify undesirable phenomena that can occur under some conditions, such as *circular causality*, *temporary operation reordering*, *split brain syndrome* and *phantom operations*, and show how they can be mitigated. We also provide novel correctness criteria suitable for certain types of systems and environments. We propose *fluctuating eventual consistency* for systems in which temporary operation reordering is unavoidable, and we propose a family of *failure-aware correctness criteria* which precisely capture the achievable correctness guarantees in specific failure models.

Acknowledgements

I wish to express my sincere gratitude to my thesis advisor Prof. Paweł T. Wojciechowski, for supporting me, giving me the opportunity to pursue my research interests, and most importantly for teaching me the art of scientific research. His encouragement and guidance along the way are what made this thesis possible.

I would like to thank Tadeusz Kobus and Jan Kończak, my friends and research colleagues, for being an integral, indispensable part of this journey. Tadeusz collaborated with me on much of this work. He never failed to inspire me with his intelectual curiosity and work ethics. He taught me much about scripting, and his help with LaTeX has been invaluable to me. Tadek was a great companion on many conference trips. Jan taught me a lot about Linux, system administration and networking. I truly envy the depth of his technical knowledge, as well as his persistance in riding his bike in all weather conditions. I will always have fond memories of our countless talks, drinking coffee together, and fixing the coffee machine.

I also thank Prof. Jerzy Brzeziński, all the members of the Distributed Systems Research Group, and my fellow graduate students, Andrzej Stroiński, Anna Labijak-Kowalska, Dariusz Brzeziński, Dariusz Dwornikowski, Iwo Błądek, Kalina Jasińska-Kobus, Konrad Siek, Krzysztof Ciomek, Maciej Piernik, Mansureh Aghabeig, Marcin Szeląg, Michał Boroń, Michał Żurkowski, Paweł Kobyliński, Rafał Skowroński, Sylwia Kopczyńska, Wojciech Frohmberg, and Wojciech Wojciechowicz. Thank you, for all the memorable years.

I am grateful for the love and support of my family. In particular, I thank my parents, Ryszard and Gabriela, and my brother Jakub, for motivating me to work hard and pursue my goals.

Finally, I would like to thank my dearest Henryka, love of my life, for her patience and undestanding. Without her support, and all the inspiration I got from her, I would not be able to reach the place I am now.

List of publications

Journal papers:

M. Kokociński, T. Kobus, and P. T. Wojciechowski, "On mixing eventual and strong consistency: Acute cloud types," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, 2022

Conference papers:

M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Brief announcement: Eventually consistent linearizability," in *Proceedings of PODC '15: the 34th ACM Symposium on Principles of Distributed Computing*, July 2015

M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Brief announcement: On mixing eventual and strong consistency: Bayou revisited," in *Proc. of PODC '19*, July 2019

Outside of the main scope of the research, the author participated in the following work.

Book chapters:

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Introduction to transactional replication," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications* (R. Guerraoui and P. Romano, eds.), vol. 8913 of *Lecture Notes in Computer Science*, Springer, 2015

Journal papers:

P. T. Wojciechowski, T. Kobus, and M. Kokociński, "State-machine and deferred-update replication: Analysis and comparison," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, 2017

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Relaxing real-time order in opacity and linearizability," *Elsevier Journal on Parallel and Distributed Computing*, vol. 100, 2017

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Hybrid transactional replication: State-machine and deferred-update replication combined," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, July 2018

Conference papers:

P. T. Wojciechowski, T. Kobus, and M. Kokociński, "Model-driven comparison of state-machine-based and deferred-update replication schemes," in *Proceedings of SRDS '12: the 31st IEEE International Symposium on Reliable Distributed Systems*, Oct. 2012

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *Proceedings of ICDCS '13: the 33rd IEEE International Conference on Distributed Computing Systems*, July 2013

M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Make the leader work: Executive deferred update replication," in *Proceedings of SRDS '14: the 33rd IEEE International Symposium on Reliable Distributed Systems*, Oct. 2014

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "The correctness criterion for deferred update replication," in *Program of TRANSACT '15: the 10th ACM SIGPLAN Workshop on Transactional Computing*, June 2015

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Jiffy: A lock-free skip list with batch updates and snapshots," in *Proceedings of PPoPP '22: the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Apr. 2022

Patent applications:

P. T. Wojciechowski, T. Kobus, and M. Kokociński, "A fault-tolerant data processing computer system and method for implementing a distributed two-tier state machine," 2017. EPO patent no. EP 3193256 B1, July 7, 2017

P. T. Wojciechowski, T. Kobus, and M. Kokociński, "A fault-tolerant data processing computer system and method for implementing a distributed two-tier state machine," 2018. USPTO patent no. US 10135929 B2, Nov. 20, 2018

Contents

1 Introduction **1**

1.1 Motivations . . . 3
1.2 Aims and contributions . . . 4
1.2.1 Mixed-consistency semantics . . . 5
1.2.2 Correctness in the presence of failures . . . 7
1.3 Thesis Outline . . . 8

2 Acute Cloud Types **11**

2.1 Acute non-negative counter . . . 11
2.2 Bayou . . . 13
2.2.1 Protocol overview . . . 13
2.2.2 Detailed description . . . 14
2.2.3 Anomalies . . . 17
2.2.4 Correctness guarantees . . . 18
2.2.5 Progress guarantees . . . 19
2.2.6 Fault-tolerance . . . 20
2.2.7 AcuteBayou . . . 21
2.3 Acute non-negative counter vs AcuteBayou . . . 23
2.4 Formalizing Acute Cloud Types . . . 23
2.4.1 System model . . . 24
2.4.2 Design properties . . . 26

3 Formal framework for mixed-consistency systems **29**

3.1 Preliminaries . . . 29
3.1.1 Functions and tuples . . . 29
3.1.2 Relations . . . 29
3.1.3 Event graphs . . . 30
3.2 Histories . . . 31
3.3 Abstract executions . . . 32
3.4 Correctness predicates . . . 33

3.5 Replicated data type . . . 33
3.6 ACT specification . . . 35
3.7 Correctness criteria . . . 36
3.7.1 Key requirements for eventual consistency . . . 36
3.7.2 Basic Eventual Consistency . . . 36
3.7.3 Fluctuating Eventual Consistency . . . 38
3.7.4 Operation levels . . . 39
3.7.5 Strong consistency . . . 40
3.8 Correctness of ANNC and AcuteBayou . . . 42

4 Limitations of mixed-consistency 43
4.1 Other solutions . . . 47
4.1.1 Symmetric models with strong operations blocking upon a single crash . . . 48
4.1.2 Symmetric Bayou-like models . . . 49
4.1.3 Asymmetric models with cloud as a proxy . . . 49
4.1.4 Asymmetric master-slave models . . . 50
4.1.5 Other approaches . . . 50

5 Explicit failures modelling 53
5.1 Motivations and an example . . . 53
5.2 System model . . . 55
5.2.1 Replicas . . . 55
5.2.2 Clients . . . 56
5.2.3 Clients – replicas interactions . . . 57
5.2.4 Network properties . . . 58
5.2.5 Summary . . . 60
5.3 Formal framework . . . 60
5.3.1 Histories . . . 60
5.3.2 Abstract executions . . . 62
5.3.3 Correctness predicates and replicated data type . . . 62
5.3.4 Basic eventual consistency . . . 62

6 Client-side guarantees 65
6.1 Context preservation . . . 66

7 Correctness in the face of failures 69
7.1 Network partitions and state convergence . . . 69
7.2 Replica crashes and phantom operations . . . 73
7.3 Replica recovery and stable storage . . . 76
7.4 Summary . . . 82

8 Related work 83
8.1 High availability . . . 83
8.2 Eventual consistency . . . 84
8.3 Causal consistency and session guarantees . . . 85
8.4 Limitations of highly available systems . . . 86

8.5 Highly available system designs 87

9 Conclusions **89**

Bibliography **91**

A StateObject properties **103**

B Correctness proofs for ANNC and AcuteBayou **105**

B.1 ANNC correctness proofs 106

B.2 AcuteBayou correctness proofs 111

Streszczenie **121**

1 Introduction

The global services that lie at the heart of today's Internet, such as messaging applications, social media, e-commerce, banking, stock exchange, or online gaming, are powered by numerous complex distributed systems. In order to cope with the increasing traffic generated by millions of users these systems must be horizontally scalable, which means that their capacity can be increased by introducing more processing nodes. The massive scalability requirements are further complicated by the fact that these systems must stay operational at all times. As no computer or networking equipment is completely immune to hardware faults, the services themselves must be implemented in a way that lets them gracefully tolerate failures. That way the systems running these services can become *highly available*, i.e. they continue to serve client requests even when (partial) failures occur.

A common technique to increase system availability is *replication*, which consists of keeping multiple copies of service data and code, called *replicas*, on physically distinct nodes, often dispersed geographically. Besides providing fault-tolerance, replication facilitates scalable performance and lowers response times when replicas are located geographically close to clients. Traditional replication schemes, such as *state machine replication* [15, 16] or *primary-backup* [17], enforce strong consistency between replicas, i.e. the replicas coordinate their state changes so that the system as a whole appears to the clients as a centralized single server. However, keeping the replicas in sync is costly, as it usually entails solving distributed consensus. Thus, before a response can be returned to a client several messages need to be exchanged between replicas, which greatly amplifies response times. Moreover, maintaining replicas' consistency is impossible when network splits occur and the service ought to remain available, as stated in the famous CAP conjecture [18]. Thus, traditional strongly consistent replication schemes only guarantee availability in case of (a limited number of) replica crashes, but not in case of network failures resulting in lack of connectivity between groups of replicas.

To overcome the above limitations consistency requirements can be relaxed.

When some form of *eventual consistency* [19] is provided, the replicas need to synchronize their states only eventually. Thus, replicas can process client requests independently and disseminate state changes asynchronously. Thus, even when network splits occur, high availability can be maintained. To achieve this goal, eventually consistent systems feature a decentralized architecture and rely on peer to peer (asynchronous) communication protocols: a design pioneered by the seminal Amazon's Dynamo storage system [20], and followed in a plethora of popular NoSQL data stores (see, e.g., Apache Cassandra [21], Scylla [22], Riak [23], Voldemort [24], and Netflix's Dynomite [25]).

However, relaxed consistency models offer weaker guarantees and by design allow a certain degree of inconsistencies to occur. If not handled correctly, they may lead to undesired anomalies, including data loss. Thus, developers must carefully design the replicated services' code to cover all edge cases and account for the possible anomalies. In order to alleviate this burden, specialized data structures called conflict-free replicated data types (CRDTs) [26, 27, 28] are introduced. CRDTs can be implemented solely in an asynchronous manner and by design ensure eventual convergence of replica states. Popular CRDTs include Multi-Value Registers (MVRs), Last-Write-Wins Registers (LWW-registers), Positive-Negative-Counters (PN-counters), Observed-Remove Sets [27], and structures for collaborative text editing [29].

Unfortunately, the semantics of CRDTs are very limited. In order 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, implementing 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 [30].

Due to the inherent shortcomings of CRDTs, and eventual consistency in general, recently there have been several attempts both in the industry [31, 32, 33, 34], as well as in academia [35, 36, 37, 38, 39, 40, 41, 42], 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. Thus, a special class of highly available systems emerges, called *mixed-consistency systems*, in which only certain operations are required to be available. Operations that are executed with lower consistency guarantees, called *weak*, remain available in spite of failures, while operations executed in strongly consistent manner, called *strong*, may block due to, e.g., network splits. The analysis and formalization of mixed-consistency correctness properties are a major topic of this dissertation.

1.1 Motivations

As discussed, because of the growing importance of high availability in modern global services, highly available systems, including middleware solutions such as NoSQL data stores, proliferate in the Internet. Due to our increasing reliance on such systems, the study and verification of their correctness is of the utmost concern. Although there is already a significant body of research in the area, still there is much to be done.

For a long time eventual consistency has evaded being clearly defined and formalized. Several definitions had been proposed (see, e.g., [19, 43, 44, 45, 46, 2, 26, 37, 47, 48, 49]), which varied greatly both in the formalization techniques used, and in practical guarantees implied. On the other hand, strong consistency, which has been in use for decades (see, e.g., [50, 51, 52]), is much better understood. It is because strong consistency is based on a clear principle: a strongly consistent system executing requests in parallel should be indistinguishable from a one executing requests sequentially. Compared to strong consistency, eventual consistency provides guarantees that are not only much weaker, but also difficult to grasp due to their complexity or vagueness. For instance, the definition given by Vogels [19] stipulates that, when updates cease eventually all read operations return the same value, but it does not put any constraints on return values when updates never cease. Then again, *cloud types* [37] require the user to think in terms of revisions which can fork and join as in source control systems, a model called *revision consistency* [47], which was later abandoned due to excessive complexity [38]. As a result, proving correctness of some particular eventually consistent system, as well as reasoning about such systems in general, is more challenging. Moreover, when eventually consistent (weak) operations are mixed with strongly consistent (strong) ones, within the same mixed-consistency system, the confusion regarding the provided guarantees is amplified. In fact, currently there is no general consensus on the expected semantics of such systems.

Mixed-consistency systems employed in the industry lack clearly stated semantics, or have them severly restricted. For example, in Apache Cassandra [21] using the *light weight transactions* on data that are accessed at the same time in the regular, eventually consistent, fashion leads to undefined behaviour [53]. On the other hand, in Riak [23] data items accessed by strong and weak operations must be stored under separate namespaces (called *buckets*) [34], and thus there is no actual mixed-consistency semantics. Other systems [39, 31, 32] offer selectable consistency levels only for read-only operations by allowing clients to read stale or fresh data. On the other hand updates are always executed as strong operations.

All the known approaches that do provide mixed-consistency semantics feature some limitations in regard to their behaviour during failures. For example, in *cloud types* [37] as well as in *global sequence protocol* (GSP) [38] all the updating operations (both weak and strong) must pass through a centralized subsystem,

called *the cloud*, which disseminates the updates to all the nodes in an ordered stream. When the cloud is unavailable, e.g. due to failures of majority of servers within the cloud or a network split, an updating operation can still be performed and applied locally on some node, but its effect is not visible to other nodes. As a further example, in *lazy replication* [54], *RedBlue consistency* [35], and *partial order restrictions* [36] all replicas are required to be operational to execute strong operations. Thus, a single replica crash may block the system, with regard to strong operations, until the failure is fixed. Typical strongly consistent replicated systems utilizing *non-blocking* agreement protocols, such as Paxos [55], may tolerate up to half of replicas crash, and continue processing operations. Thus, the inability to tolerate even a single crash in a highly available system that ought to gracefully tolerate failures seems deeply unsatisfactory, even if the crash affects only strong operations.

The approaches mentioned above, when faced with failures, compromise either the progress of weak operations (by not propagating their effects), or the progress of strong operations (by blocking them). Such tradeoffs resulting from mixing weak and strong operations are worth exploring. In particular, an interesting question we try to answer in this dissertation is whether there exists a mixed-consistency system that handles strong operations in non-blocking manner (tolerates some number of replica crashes), while not inhibiting the progress of weak operations. Such a system would offer the best of both worlds: high availability and low latency in case of weak operations similarly as in eventually consistent systems, and strong guarantees and (limited) failure tolerance in case of strong operations similarly as in strongly consistent systems. The next question naturally concerns the correctness guarantees that such as a system can provide.

Whether a highly available system features additional strong operations or not, ensuring its correct behaviour when failures occur is critical. It is especially so since these systems are specifically designed for scenarios where failures are imminent. It comes as a surprise then that the majority of works concerning the correctness of highly available systems that can be found in the literature abstract away from replica or network failures altogether (see, e.g., [45, 35, 46, 56, 57, 58, 59, 60, 36, 61, 48, 49]). The analysis conducted that way can be deemed incomplete: a protocol that works correctly only when no failures occur does not necessarily work as expected when failures *do* happen. Thus, a comprehensive correctness analysis, which explicitly considers various failure models, may reveal new insights and inherent tradeoffs unnoticed so far by the research community and the industry.

1.2 Aims and contributions

Given the motivations presented above, we formulate our main thesis as follows:

Tradeoffs and limitations in highly available systems' correctness guarantees resulting from mixed-consistency operations and from machine and network failures can be formally identified and reasoned about.

We demonstrate the veracity of the thesis in two parts. Firstly, we identify and analyze the tradeoffs in correctness resulting from mixed-consistency semantics. Secondly, by taking a holistic approach to correctness analysis, which involves creating a faithful model of real-life client-server systems and explicit inclusion of failures, we precisely define the limitations on correctness of such systems resulting from machine and network failures.

Below we summarize the main contributions of this dissertation in detail.

1.2.1 Mixed-consistency semantics

In order to formally study the mixed-consistency systems and expose tradeoffs in their correctness we introduce a new abstraction called *acute cloud types* (ACTs). ACTs constitute 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 (unlike in cloud types [37]. On the other hand, strong operations can provide guarantees even as strong as linearizability [52] with respect to the already completed strong operations and a precisely defined subset of completed weak operations. Crucially, strong operations are *non-blocking*: they can leverage efficient, quorum-based synchronization protocols, such as Paxos [55], and thus gracefully tolerate machine and network failures (unlike RedBlue consistency [35]). Both weak and strong operations can be arbitrarily complex, but they must be deterministic.

Furthermore, unlike in the RedBlue consistency and in similar approaches (e.g., [54, 59, 36]), in which weak operations are causally consistent by design, 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 [62], and is not always needed [63].

We model ACTs as state automata that communicate with each other via an asynchronous network. Additionally, in the model we implicitly provide a failure detector which can be used to solve distributed consensus synchronization

necessary to deliver strongly consistent operations. In this part we consider machine and network crashes only implicitly. More precisely, we consider *stable* (failure-free) system runs, and *asynchronous* ones in which consensus-based synchronization does not terminate. We define ACTs through a set of *implementation restrictions* which enforce, among others, that the ACTs' state automata indeed do provide high availability for weak operations and non-blocking semantics for strong operations.

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 [51, 50, 52], as well as various weaker models, e.g., [64, 44, 65, 61]. During execution, strong operations are guaranteed to observe a prefix of S up to their position in S. Conversely, 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 [49], 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) [48].

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. This result represents our main contribution in regards to identifying correctness limitations of mixed-consistency

systems.

In order to illustrate our concepts and analysis, we present an ACT for a non-negative counter and also revisit Bayou [44], a seminal, always available, eventually consistent data store. Bayou combines timestamp-based eventual consistency [19] and serializability [51] 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.

1.2.2 Correctness in the presence of failures

As we mentioned earlier, the majority of the existing work on the correctness of highly available systems either abstract away from machine failures altogether [45, 35, 46, 57, 58, 59, 60, 36, 61], or admit machine failures or network splits but the correctness proofs only consider system runs in which no failures occur [48, 49]. In this dissertation we holistically approach the problem of correctness of highly available systems in the presence of failures. To this end we introduce a novel formal framework that explicitly considers hardware failures, such as transient or permanent machine crashes and network splits. In order to make our analysis complete we extend our system model to more closely follow systems that are actually deployed in real world. In particular we consider external clients of various types (stateless, stateful, sticky or mobile), load balancers that route requests from clients to replicas, and we admit replica recovery from stable storage after crash. By utilizing three machine failure models, and two network failure models, we consider a total of six combined failure models.

Since in highly available systems replicas serve client requests independently (without blocking communication with other replicas), many client-side correctness guarantees (called *session guarantees*) hinge on the ability of clients to remain permanently connected to the same replica, or to retain state to store some metadata and cache previous system responses. In general, due to machine and network failures, system designers cannot expect client requests to be routed to the same replica each time. Therefore, it is especially important to consider external clients which can connect to different replicas, and which are either stateful or stateless. Thus, we discuss when clients can be stateless and, if they do need to maintain some state, how to place requirements on their sessions. In particular, we define a novel session guarantee called *context preservation* (CP) for systems that expose the concurrency to the client (e.g., implement multi-value registers [26] or observed-removed sets [27]). CP is incomparable with the four classic session guarantees [66].

There are many possible failure scenarios that need to be considered in order to ensure that a protocol works as intended in real-life environments where failures are to be expected. Popular eventually consistent systems we are aware of do feature various anti-entropy mechanisms that prevent them from exhibiting undesirable anomalies [21, 23]. However, when the correctness of a protocol is proven using only system runs in which no failures occur, the robustness of such

mechanisms is not fully validated, as we explain below.

Correctness guarantees can be divided into safety (*nothing bad ever happens*) and liveness (*eventually something good happens*) properties [67] [68]. When proving a safety property of a protocol (which does not feature any post-crash recovery procedures) it is indeed sufficient to only consider failure-free runs. It is so because in finite executions crashed nodes are indistinguishable from very slow ones and network splits are indistinguishable from occurances of temporary communication delays. The same, however, does not hold for liveness guarantees, which can be violated only in infinite executions. Some strongly consistent correctness criteria, such as linearizability [52], are pure safety properties.[1] However, correctness criteria based on eventual consistency are an intersection of safety *and* liveness guarantees. Thus, in that case all possible system runs with failures need to be considered in a correctness proof. Our framework allows us to formally study both the safety and liveness aspects of the correctness of highly available systems when failures occur.

We show specific liveness guarantees that cannot be provided when certain failures occur, such as *eventual visibility* of all events. Thus, we formally identify a set of undesired phenomena, which are observable by the clients but, as we prove, are unavoidable in the considered environments. In particular, when unrecoverable replica crashes are possible, a successful execution of an operation *op* may be first acknowledged to the client that submitted it, but later *op* may appear as if it had never been executed by any replica. We call such operations *phantom operations*. We discuss, however, that with the proliferation of low-latency solid state drives (SSDs) and the advent of new technologies such as non-volatile memory (NVM, also called persistent memory) [70], phantom operations can be mitigated in many cases with minimal performance overhead.

Finally, we propose to relax the liveness requirements which cannot be satisfied in certain failure models. However, we posit that this relaxation occurs only due to failures and not arbitrarily, i.e. the guarantees should still hold completely in failure-free runs. Thus, we define a family of *failure-aware* correctness criteria, to adequately capture the behaviour of eventually consistent systems in failure-prone environments. We use our novel correctness criteria to systematize in a formal way the existing knowledge and intuitions regarding the correctness of highly available systems under failure conditions.

1.3 Thesis Outline

The thesis is organized as follows. In Chapter 2 we introduce acute cloud types and revisit Bayou. Then, in Chapter 3 we present our formal framework for mixed-consistency systems and define various correctness criteria including our fluctuating eventual consistency (FEC). Next, in Chapter 4 we explore the limitations of mixed-consistency systems. In Chapter 5 we extend our system model

[1] Assuming a deterministic or finitely undeterministic protocol [69].

to explicitly consider failures and we present the formal framework for reasoning about the correctness in the presence of failures. In Chapter 6 we discuss session guarantees and introduce context preservation (CP). Then, in Chapter 7 we analyze the achievable correctness guarantees in certain failure models and we introduce failure-aware correctness criteria. In Chapter 8 we revisit other work that is related to our research. Finally, we conclude in Chapter 9.

2
Acute Cloud Types

In this chapter we introduce ACTs. Before we define them formally, we begin with an overview of ACTs by discussing some examples. First, we propose an ACT protocol (or simply an ACT) for a non-negative counter. Then, we study the seminal Bayou protocol [44], which, with some modifications, can be considered a general ACT that enables execution of arbitrarily complex (deterministic) operations.

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 32) operations are weak and thus guarantee low response times, whereas subtract (line 12) 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.

In order 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* [71]) as well as a protocol that involves inter-replica synchronization (modelled using *total order broadcast, TOB* [72], which can be efficiently implemented using quorum-based protocols, such as Paxos [55]; lines 9-10). On the other hand, the

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

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 26-28).

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 22–23). 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.

ANNC guarantees a property which is a conjunction of *basic eventual consistency* (BEC) [48, 49] for weak operations (add and get) and *linearizability* (LIN) [52] for strong operations (subtract). We formalize BEC and LIN in Sections 3.7.2 and 3.7.5, and prove the correctness of ANNC in Section 3.8.

2.2 Bayou

Now let us discuss the seminal Bayou protocol. 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. In this section first we discuss the original protocol and then 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. 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 new 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.

The above 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 timestamps of the requests executed most recently), in order 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 splits (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 Detailed description

The pseudocode in Algorithm 2 specifies the Bayou protocol for replica R_i. Replicas are independent and communicate solely by message passing. When a client submits an operation op to a replica, op is broadcast within a Req message using a gossip protocol.[1] In our pseudocode, we use regular *reliable broadcast, RB* (line 12; we say that op has been $\mathrm{RB\text{-}cast}$). Through the code in line 13 we simulate immediate local $\mathrm{RB\text{-}delivery}$ of op.

Each Bayou replica totally-orders all operations it knows about (executed locally or received through RB). In order to keep track of the total order, a replica maintains two lists of operations: $committed$ and $tentative$. The $committed$ list encompasses the *stabilized* operations, i.e., operations whose final execution order has been established by the primary. On the other hand, the $tentative$ list encompasses operations whose final execution order has not yet been determined.

[1] In practice a read-only (RO) operation does not need to be broadcast to other replicas. For simplicity we omit this optimization in the pseudocode. See Section 2.2.7 for details on how to implement such an optimization.

Algorithm 2 The Bayou protocol for replica R_i

```
1: struct Req(timestamp : int, id : pair⟨int, int⟩, strongOp : boolean, op : ops(F))
2: operator <(r : Req, r' : Req)
3:     return (r.timestamp, r.id) < (r'.timestamp, r'.id)
4: var state : StateObject
5: var currEventNo : int
6: var committed, tentative : list⟨Req⟩
7: var executed, toBeExecuted, toBeRolledBack : list⟨Req⟩
8: var reqsAwaitingResp : map⟨Req, Resp⟩
9: upon invoke(op : ops(F), strongOp : boolean)
10:     currEventNo = currEventNo + 1
11:     var r = Req(currTime, (i, currEventNo), strongOp, op)
12:     RB-cast(ISSUE, r)
13:     insertIntoTentative(r)
14:     reqsAwaitingResp.put(r, ⊥)
15: procedure insertIntoTentative(r : Req)
16:     var previous = [x|x ∈ tentative ∧ x < r]
17:     var subsequent = [x|x ∈ tentative ∧ r < x]
18:     tentative = previous · [r] · subsequent
19:     var newOrder = committed · tentative
20:     adjustExecution(newOrder)
21: upon RB-deliver(ISSUE, r : Req)
22:     if r.id.first = i then                                  // r issued locally
23:         return
24:     if r ∉ committed then
25:         insertIntoTentative(r)
26: upon FIFO-RB-deliver(COMMIT, r : Req)
27:     if i = primary then
28:         return
29:     commit(r)
30: procedure commit(r : Req)
31:     committed = committed · [r]
32:     tentative = [x|x ∈ tentative ∧ x ≠ r]
33:     var newOrder = committed · tentative
34:     adjustExecution(newOrder)
35:     if reqsAwaitingResp.contains(r) ∧ r ∈ executed then
36:         return reqsAwaitingResp.get(r) to client
37:         reqsAwaitingResp.remove(r)
38: periodically primaryCommit()
39:     if i = primary ∧ tentative ≠ [] then
40:         var [head] · tail = tentative
41:         commit(head)
42:         FIFO-RB-cast(COMMIT, head)
43: procedure adjustExecution(newOrder : list⟨Req⟩)
44:     var inOrder = longestCommonPrefix(executed, newOrder)
45:     var outOfOrder = [x|x ∈ executed ∧ x ∉ inOrder]
46:     executed = inOrder
47:     toBeExecuted = [x|x ∈ newOrder ∧ x ∉ executed]
48:     toBeRolledBack = toBeRolledBack · reverse(outOfOrder)
49: upon toBeRolledBack ≠ []
50:     var [head] · tail = toBeRolledBack
51:     state.rollback(head)
52:     toBeRolledBack = tail
53: upon toBeRolledBack = [] ∧ toBeExecuted ≠ []
54:     var [head] · tail = toBeExecuted
55:     var response = state.execute(head)
56:     if reqsAwaitingResp.contains(head) then
57:         if ¬head.strongOp ∨ head ∈ committed then
58:             return response to client
59:             reqsAwaitingResp.remove(head)
60:         else
61:             reqsAwaitingResp.put(head, response)
62:     executed = executed · [head]
63:     toBeExecuted = tail
```

Algorithm 3 StateObject

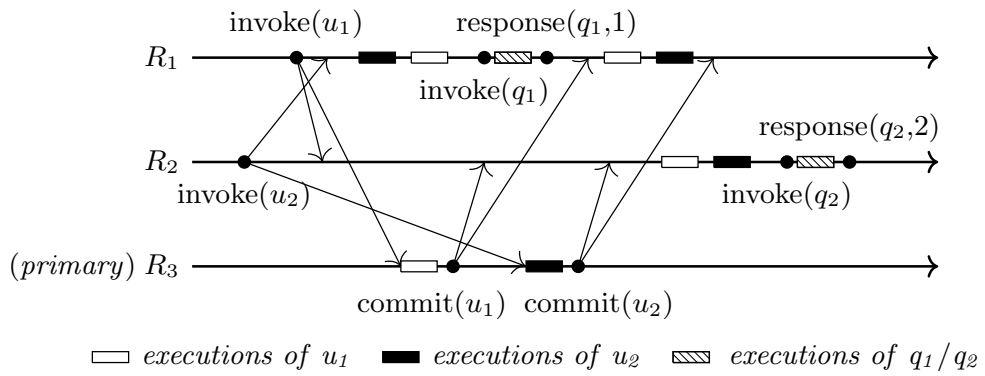
```
1: var db : map⟨Id, Value⟩
2: var undoLog : map⟨Req, map⟨Id, Value⟩⟩
3: function execute(r : Req)
4:     var undoMap : map⟨Id, Value⟩
5:     instrument r.op as below and execute line by line
6:     upon any x = read(id) replace with
7:         x = db[id]
8:     upon any write(id, v) replace with
9:         if undoMap[id] = ⊥ then
10:            undoMap[id] = db[id]
11:        db[id] = v
12:    upon any return response replace with
13:        undoLog[r] = undoMap
14:        return response
15: function rollback(r : Req)
16:    var undoMap = undoLog[r]
17:    for (k, v) ∈ undoMap do
18:        db[k] = v
19:        undoLog = undoLog \ (r, undoMap)
```

The operations on the *tentative* list are sorted using the operations' timestamps (to resolve any ties, the replica identifiers and per replica sequence numbers are used). A timestamp is assigned to an operation as soon as a Bayou replica receives it from a client.

A Bayou replica continually executes operations one by one in the order determined by the concatenation of the two lists: $committed \cdot tentative$ (line 55). The replica keeps additional data structures, such as *executed* and $toBeExecuted$, to keep track of its progress. An operation $op \in committed$, once executed, will not be executed again as its final operation execution order is determined. On the other hand, an operation in the *tentative* list might be executed and rolled back multiple times. It is because a replica adds operations to the *tentative* list (rearranging it if necessary; lines 18-16) as they are delivered by a gossip protocol. Hence, a replica might execute some operation op, and then, in order to maintain the proper execution order consistent with the modified *tentative* list, the replica might be forced to roll op back (line 51), execute a just received operation op' (which has lower timestamp than op), and execute op again. We maintain the $toBeRolledBack$ list of operations scheduled for rollback (operations are kept in the order reverse to the one in which they were executed, line 48). An operation execution can proceed only once all the scheduled rollbacks have been performed.

One of the replicas, called the *primary*, periodically *commits* operations from its *tentative* list by moving them to the end of the *committed* list, thus establishing their final execution order (line 38). The primary announces the commit of operations by RB-casting commit messages, so that each replica can also commit the appropriate operations. Note that the primary uses the FIFO variant of RB to ensure that all replicas commit the same set of operations in the same order.

Operations are executed on the *state* object (line 4), which encapsulates the state of the local database. At any moment, the value of *state* corresponds to




```
u1 {                u2 {                q1/q2 {
   x = 1               y = 1               return z
   if y = 1 then       if x = 1 then    }
      z = 1               z = 2
}                   }
```

Figure 2.1: Example execution of Bayou showing temporary operation reordering and circular causality.

a sequence s of the already executed operations on a replica given, where s is a prefix of $committed \cdot tentative$. Note that $state$ allows us to easily rollback a suffix of s (line 51). We discuss the properties of the $state$ object in more detail in Appendix A.

Algorithm 3 shows a pseudocode of a referential implementation of the StateObject for arbitrary operations of any sequential data type (a specialized one can be used to take advantage of specific data type's characteristics or to enable non-sequential semantics for certain replicated data types which expose concurrency to the client). We assume that each operation can be specified as a composition of read and write operations on registers (objects) together with some local computation. The assumption is sensible, as the operations are executed locally, in a sequential manner, and thus no stronger primitives than registers (such as CAS, fetch-and-add, etc.) are necessary. The StateObject keeps an undo log which allows it to revoke the effects of any operation executed so far (the log can be truncated to include only the operations on the $tentative$ list).

2.2.3 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 Figure 2.1, 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 operation 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 operation 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., [15, 73]). 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., [64, 48]), or support only commutative operations (as in *strong eventual consistency* [26], e.g., [27, 58]). There are also protocols that allow past operations to be perceived in different (but still legal) orders (e.g., [74, 75, 35]). 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 (in principle 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 [44]). Hence, these mechanisms do not alleviate the anomalies outlined above.

2.2.4 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.* [48, 49] directly forbids circular causality (see Section 3.7.2 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.* [26] requires any two replicas that delivered the same updates to have equivalent states.[2] Obviously, Bayou neither satisfies BEC nor SEC (as evidenced by Figure 2.1). On the other hand informal definitions of eventual consistency which admit temporal reordering, such as [19], involve only liveness guarantees, which is insufficient. Bayou fulfills the operational specification in [76]. However, we are interested in declarative specifications, similar in the style to popular consistency criteria, such as *sequential consistency* [50], or *serializability* [51], through which we can concisely define the behaviour of a wide class of systems. Hence we introduce a new correctness criterion, *fluctuating eventual consistency* (FEC), which can be viewed as a generalization of BEC (see Section 3.7.3 for definition). FEC relaxes BEC, so that 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 3.8 we formally prove that the Bayou-derived general-purpose ACT satisfies the above correctness criteria.

2.2.5 Progress guarantees

In a highly available system replicas are supposed to respond to a request even in the presence of network splits. However, this requirement can be differently formalized. In the model considered by Brewer [18], a network partition can last infinitely. Then, high availability can be formalized as wait-freedom [77], which means that each request is eventually processed by the system and the response is returned to the client. In the more commonly assumed model that admits only temporary network partitions (we also adopt this model for ACTs, similarly to, e.g., [48, 58]), that requirement is not strong enough, since a replica could trivially just wait until the partitions are repaired before executing a request and responding to the client. Therefore, in such a model the requirement of high availability must be formulated differently. It can be done as follows: a system is highly available if it executes each request in a finite number of steps even when no messages are exchanged between the replicas (the replica cannot indefinitely postpone execution of a request or returning the response to the client, see Section 2.4.2 for a formal definition). In this sense, weak operations in Bayou are highly available. However, this definition of high availability does not preclude situations in which, e.g., the number of steps the execution of each request takes grows over time and thus is unbounded. Hence, one could formulate a slightly stronger requirement, i.e., *bounded* wait-freedom [77], which

[2] 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.

states that there is a possibly unknown but bounded number of protocol steps that the replica takes before a response is returned to the client upon invocation of an operation. Interestingly, unlike many popular NoSQL data stores, such as [20] or [21], Bayou does not guarantee bounded wait-freedom even for weak operations, as we now demonstrate.

Consider a Bayou system with n replicas, one of which, R_s, processes requests slower compared to all other replicas. Assume also that every fixed period of time Δt there are n new weak requests issued, one on each replica, and the processing capabilities of all replicas are saturated. In every Δt, R_s should process all n requests (as do other replicas), but it starts to lag behind, with its backlog constantly growing. Intuitively, every new operation invoked on R_s will be scheduled for execution after all operations in the backlog, as they were issued with lower timestamps. Hence the response time will increase with every new invocation on R_s. One could try to overcome the problem of the increasing latency on R_s by artificially slowing the clock on R_s, thus giving unfair priority to the operations issued on R_s, compared to operations issued on other replicas. But then any operation invoked on R_s would appear on other replicas as an operation from a distant past. In turn, any such operation would cause a growing number of rollbacks on the other replicas.

Strong operations cannot be (bounded) wait-free simply because in order for them to complete, the primary must be operational, which cannot be guaranteed in a fault-prone environment.

Interestingly, in AcuteBayou (see Section 2.2.7) the execution of weak operations is trivially bounded wait-free, as they are executed immediately upon their invocations.

2.2.6 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) [78] is used, the phenomena illustrated in Figure 2.1 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 (as all the replicas need to be operational in 2PC in order to reach distributed agreement). This makes such a system even less resilient to failures than the original one. 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.[3] As we prove later, ACTs (which do not depend on the synchronous

[3]Sharded 2PC [79] can be considered non-blocking, under the assumption that within each shard at least one process remains operational at all times. Then, in such a scheme not every pro-

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.

2.2.7 AcuteBayou

Bayou can be improved to make it more fault-tolerant and free of some of the phenomena mentioned earlier. With the modifications described below the improved Bayou protocol becomes the general-purpose ACT, called AcuteBayou.

In Algorithm 4 we present the modifications to the Algorithm 2, which give us AcuteBayou. Note that, we also improve the execution of weak read-only (RO) operations (since any RO operation op does not change the logical state of the $state$, op can be executed only locally[4]).

Firstly, we use TOB in place of the primary to establish the final operation execution order. More precisely, every (weak, updating) operation is broadcast using RB (as before) as well as TOB (lines 13–14). When a replica TOB-delivers an operation op (line 20), it stabilizes op. Since TOB guarantees that all replicas TOB-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 have argued, TOB can be implemented in a way that avoids a single point of failure [55].

Further changes are aimed at eliminating circular causality in Bayou as well as improving the response time for weak operations. To this end (1) any strong operation is broadcast using TOB only (line 19), and (2) upon being submitted, any weak operation is executed immediately on the current state, and then rolled back (lines 11 and 12). It is easy to see that the modification (2) means the incoming stream of weak operations from other replicas cannot delay the execution of weak operations submitted locally. Below we argue why the two above modifications allow us to avoid circular causality in Bayou.

The change (1) means that for any pair of a strong s and a weak operation w, if the return value of any operation e depends on both s and w (e *observes* s and w), they will be observed in an order consistent with the final operation execution order. We prove it through the following observations:

1. for e to observe s, s must be committed (in the modified algorithm s never appears on the $tentative$ list),
2. if e is a strong operation, then w must also be committed, because upon execution strong operations do not observe operations on the $tentative$ list; hence both operations are observed according to their final execution order,
3. otherwise (e is a weak operation):
   a) w is updating (not RO), because otherwise it would not logically impact the return value of e,

cess needs to be contacted to commit a transaction, thus it falls under the quorum-based category.

[4]We assume that StateObject features an overloaded execute function which takes a plain operation as an argument, instead of a Req record, when executing RO operations.

Algorithm 4 Modifications to the Bayou protocol that produce AcuteBayou

```
    // redefines the Req record
 1: struct Req(timestamp : int, id : pair⟨int, int⟩, ctx : set⟨pair⟨int, int⟩⟩,
                  strongOp : boolean, op : ops(F))
    // replaces invoke(...) and primaryCommit()
 2: upon invoke(op : ops(F), strongOp : boolean)
 3:     if ¬strongOp ∧ op ∈ readonlyops(F) then
 4:         var response = state.execute(op)
 5:         return response to client
 6:     else
 7:         currEventNo = currEventNo + 1
 8:         var ctx = (⋃_{r∈executed} r.id) ∪ (⋃_{r∈toBeRolledBack} r.id)
 9:         var r = Req(currTime, (i, currEventNo), ctx, strongOp, op)
10:         if ¬strongOp then
11:             var response = state.execute(r)
12:             state.rollback(r)
13:             RB-cast(ISSUE, r)
14:             TOB-cast(COMMIT, r)
15:             insertIntoTentative(r)
16:             return response to client
17:         else
18:             reqsAwaitingResp.put(r, ⊥)
19:             TOB-cast(COMMIT, r)
    // replaces upon FIFO-RB-deliver(COMMIT, r : Req)
20: upon TOB-deliver(COMMIT, r : Req)
21:     commit(r)
```

b) if w is already committed, it is similar to case 2,

c) if w is not yet committed, e will observe the operations in the order s, w; on the other hand, once w is delivered by TOB and committed, it will appear on the *committed* list after s, and so e also observes s and w in the same order s, w.

The change (2) is necessary to prevent circular causality between two (or more) weak operations (the case depicted in Figure 2.1. It is because the modified algorithm executes a weak (updating) operation op without waiting for the RB-cast/TOB-cast message to arrive. It means that no concurrent operation op' will be executed prior to the first execution of op, whose return value observes the client. Otherwise op could observe op' even though the final execution order is op, op'.

Finally, we redefine the Req record to include the *execution context* ctx, i.e., the identifiers of requests already executed upon the invocation of the current operation and which have influenced the $state$ object (those on the $executed$ list and those on the $toBeRolledBack$ list). Note that in practice such identifiers can be efficiently represented using *Dotted Version Vectors* [80]. With the augmented Req record the implementation of StateObject can take advantage of the relative visibility between operations to achieve the non-sequential semantics of such replicated data types as MVRs or ORsets.

Note that the modified variant of Bayou does not ensure that subsequent operations invoked by the same client observe the effects of previous ones, even if all of them are issued on the same replica (the *read-your-writes* session guarantee,

RYW [66]). In the original Bayou system, clients were collocated with replicas and RYW was naturally provided. However, modern eventually consistent systems eschew sessions guarantees for performance reasons. More precisely, providing RYW without synchronous replication between servers either forces the client to always connect to the same server (which is typically avoided because of the use of load-balancers and stateless, thin, mobile clients), or to maintain a significant state on the client-side. See also Chapter 6 for an in depth discussion on session guarantees.

2.3 Acute non-negative counter vs 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. Firstly, in order to facilitate high availability and low response times (which are essential in geo-replicated environments), frequently invoked operations should be defined as weak operations 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 [58] (see Section 2.4.2 for details). Secondly, 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 are meant to provide the programmer with 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.

2.4 Formalizing Acute Cloud Types

An acute cloud type 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 Chapter 3 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* [58]. They guarantee *genuine*, low-latency, eventually-consistent processing for weak operations (as in, e.g., CRDTs [26]). The last restriction guarantees that strong operations are implemented using a non-blocking agreement protocol, instead of a non-fault-tolerant approach requiring all the replicas to be operational. In Sections 2.4.1 and 2.4.2 we formalize the system model and provide precise definitions of the implementation restrictions.

2.4.1 System model

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.[5] 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)* [71], 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). We say that a specific event is *enabled* on a replica, if its preconditions are met (e.g., an $\mathrm{RB\text{-}deliver}(m)$ event is enabled on a replica R, if m was previously $\mathrm{RB\text{-}cast}$ 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*.

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 [81]. Real distributed systems which exhibit some amount of *synchrony* can usually overcome this limitation. For example, in a quasi-synchronous

[5] We assume full replication for simplicity.

model [82], 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 splits), 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 $\mathrm{TOB\text{-}cast}$ and $\mathrm{TOB\text{-}deliver}$ primitives. Since, TOB is known to require a failure detector at least as strong as Ω to terminate [83], we guarantee it achieves progress only in stable runs.

In both asynchronous and stable runs we guarantee the basic properties of reliable message passing [71], i.e.:

- if a message is $\mathrm{RB\text{-}delivered}$, or $\mathrm{TOB\text{-}delivered}$, then it was, respectively, $\mathrm{RB\text{-}cast}$, or $\mathrm{TOB\text{-}cast}$, by some replica,
- no message is $\mathrm{RB\text{-}delivered}$, or $\mathrm{TOB\text{-}delivered}$, more than once by the same replica,
- if a correct replica $\mathrm{RB\text{-}casts}$ some message, then eventually it $\mathrm{RB\text{-}delivers}$ it,
- if a correct replica $\mathrm{RB\text{-}delivers}$ some message, then eventually all correct replicas $\mathrm{RB\text{-}deliver}$ it,
- if any (correct or faulty) replica $\mathrm{TOB\text{-}delivers}$ some message, then eventually all correct replicas, $\mathrm{TOB\text{-}deliver}$ it,
- messages are $\mathrm{TOB\text{-}delivered}$ by all replicas in the same total order.

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

Solely in stable runs, we also guarantee the following:

- if a correct replica $\mathrm{TOB\text{-}casts}$ some message, then eventually all correct replicas $\mathrm{TOB\text{-}deliver}$ it.
- if a message m was both $\mathrm{RB\text{-}cast}$ and $\mathrm{TOB\text{-}cast}$ by some (correct or faulty) replica, and m was $\mathrm{RB\text{-}delivered}$ by some correct replica, then eventually all correct replicas $\mathrm{TOB\text{-}deliver}$ 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 presentation of certain algorithms, such as ANNC and AcuteBayou.

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 [49] and [60]. 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* [67, 68]). Additionally, all fair executions of the protocol satisfy liveness aspects (*something good eventually happens*).[6]

2.4.2 Design properties

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

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.

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. We say that 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*.

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

[6] In Chapter 5 we use a relaxed definition of a fair execution to account for faulty replicas.

previously some non-read-only operation was invoked on R, and since then R did not enter a passive state.

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).

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.

Summary

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[7]), see, e.g., [84, 28, 20, 85, 86, 26, 87, 58, 60]. Restrictions 1–4 are inspired by the ones defined for write-propagating data stores [58], 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 Section 2.2.2). On the other hand, the 5th requirement concerns strong operations, and so is specific for ACTs. As discussed in [58, 60], 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 register, i.e., it admits fewer execution traces. Thus it satisfies a more stringent consistency

[7] 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.

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* [58].

3

Formal framework for mixed-consistency systems

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.* [48] (also used by several other researchers, e.g., [57, 58, 60, 88], see also [49] for a textbook tutorial).

3.1 Preliminaries

3.1.1 Functions and tuples

We use standard notations for functions: $A \to B$ denotes the set of all functions from A to B, and $f : A \to B$ means that $f \in (A \to B)$. For any function $f : A \to B$ we denote by $f^{-1} : B \to A$ the inverse of f: for any $b \in B$, $f^{-1}(b) = \{a \in A | f(a) = b\}$. We may treat functions as relations, thus $f(a) = b \Leftrightarrow (a, b) \in f$. In case of partial functions we use the following notation to indicate that $a \in A$ does not belong to the domain of f: $f(a) = \perp$. We use the $\perp$ symbol also to denote an undefined value in other contexts such as in *pairs* or *tuples*. If the structure of a tuple is well defined, e.g. $a = (x, y, z)$, we often write $a.x$ to simply denote x.

3.1.2 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., $(a \xrightarrow{id_A}$

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*

Figure 3.1: Definitions of common properties of a binary relation $rel \subseteq A \times A$.

$b) \Leftrightarrow (a \in A) \wedge (a = b)$. For $n \in \mathbb{N}_0$, we let rel^n be the n-ary composition $rel; rel...; rel$, 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 Figure 3.1 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 $a_0 \in A$:

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

3.1.3 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 $(E, d_1, ..., d_n)$, where $E \subseteq Events$ is a finite or countably infinite set of events drawn from universe $Events$ (the set of all possible events that we may encounter in a concrete situation), $n \geq 1$, 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 inter-

preted 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 $G = (E, d_1, ..., d_n)$ and $G' = (E', d'_1, ..., d'_n)$ 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 vs. 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 $x \xrightarrow{d_i} y \Leftrightarrow \phi(x) \xrightarrow{d'_i} \phi(y)$.

3.2 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 operation invoked in the event.

We consider only *well-formed* histories for which the following holds: 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 [89, 49]),

- 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).

3.3 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* [50] (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, and 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.

3.4 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 $\mathcal{P}_i \lesseqgtr \mathcal{P}_j$.

3.5 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 [52] is not sufficient for replicated objects which expose concurrency to the client, e.g. multi-value register (MVR) [26] or observed-remove set (OR-Set) [27]. Hence, we utilize *replicated data types* specification [28].

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, as well as the information about sessions. The set of previously invoked operations, coupled with their relative visibility and arbitration, unambiguously defines the output of the operation invoked in the considered event. 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

$$\mathcal{F}_{reg}(write(v), (E, op, vis, ar)) = ok$$

$$\mathcal{F}_{reg}(read(), (E, op, vis, ar)) = \begin{cases} v & \text{if } \exists e \in E : (op(e) = write(v) \wedge \nexists e' \in E : (op(e') = write(v') \wedge e \xrightarrow{ar} e')) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{F}_{MVR}(write(v), (E, op, vis, ar)) = ok$$

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

$$\mathcal{F}_{orset}(add(v), (E, op, vis, ar)) = ok$$

$$\mathcal{F}_{orset}(remove(v), (E, op, vis, ar)) = ok$$

$$\mathcal{F}_{orset}(read(), (E, op, vis, ar)) = \{v | \exists e \in E : op(e) = add(v) \wedge \nexists e' \in E : op(e') = remove(v) \wedge e \xrightarrow{vis} e'\}$$

$$\mathcal{F}_{seq}(append(s), (E, op, vis, ar)) = ok$$

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

where $\$ = \{a, b, c, \ldots, z\}$ and $appends(E) = \{e \in E : \exists v \in \$^* : op(e) = append(v)\}$

$$\mathcal{F}_{NNC}(add(v), (E, op, vis, ar)) = ok$$

$$\mathcal{F}_{NNC}(subtract(v), (E, op, vis, ar)) = \begin{cases} true & \text{if } foldl(0, f_{NNC}, E.sort(ar)) \geq v \\ false & \text{otherwise} \end{cases}$$

$$\mathcal{F}_{NNC}(get, (E, op, vis, ar)) = foldl(0, f_{NNC}, E.sort(ar))$$

$$\text{where } v \in \mathbb{N} \text{ and } f_{NNC} = \begin{cases} f_{NNC}(x, add(v)) & = x + v \\ f_{NNC}(x, subtract(v)) & = x - v \text{ if } x \geq v \text{ or } x \text{ otherwise} \\ f_{NNC}(x, get) & = x \end{cases}$$

Figure 3.2: Formal specifications of replicated data types for a last-write-wins register $\mathcal{F}_{reg}$, a multi-value register $\mathcal{F}_{MVR}$, an observed-remove set $\mathcal{F}_{orset}$, an append-only sequence $\mathcal{F}_{seq}$, and a non-negative counter $\mathcal{F}_{NNC}$.

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 Figure 3.2 we give the specification of five replicated data types: $\mathcal{F}_{reg}$ (a last-write-wins register), $\mathcal{F}_{MVR}$ (a multi-value register), $\mathcal{F}_{orset}$ (an observed-remove set), $\mathcal{F}_{seq}$ (an append-only sequence), and $\mathcal{F}_{NNC}$ (a non-negative counter). $\mathcal{F}_{reg}$ uses the last-write-wins policy [64] to select the most recently written value as specified by the ar relation. An instance of $\mathcal{F}_{MVR}$ stores multiple values when there are concurrent $write()$ operations ($write()$ operations not ordered by the vis relation). An $\mathcal{F}_{orset}$ removes an element x from the set only if it has been observed previously (when the $add(x)$ operation is visible to the $remove(x)$ operation). $\mathcal{F}_{seq}$ 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 3.1.

We use $\mathcal{F}_{seq}$ in the subsequent sections to illustrate various consistency models.

3.6 ACT specification

To accommodate for the mixed-consistency nature of ACTs we have to 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 $e \in E : lvl(e) \in lvlmap(op(e))$.

The ACT specification of ANNC presented in Algorithm 1 is $(\mathcal{F}_{NNC}, lvlmap_{NNC})$, where:

$$lvlmap_{NNC}(add) = \{weak\},$$
$$lvlmap_{NNC}(get) = \{weak\}, \text{and}$$
$$lvlmap_{NNC}(subtract) = \{strong\}.$$

3.7 Correctness criteria

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 $A = (E, op, rval, rb, ss, lvl, vis, ar, par)$.

3.7.1 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 2.2.3. 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 Acute-Bayou, admit temporary operation reordering, requires a new approach. We start, however, with the traditional one.

3.7.2 Basic Eventual Consistency

Intuitively, *basic eventual consistency* (BEC) [48, 49], 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

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

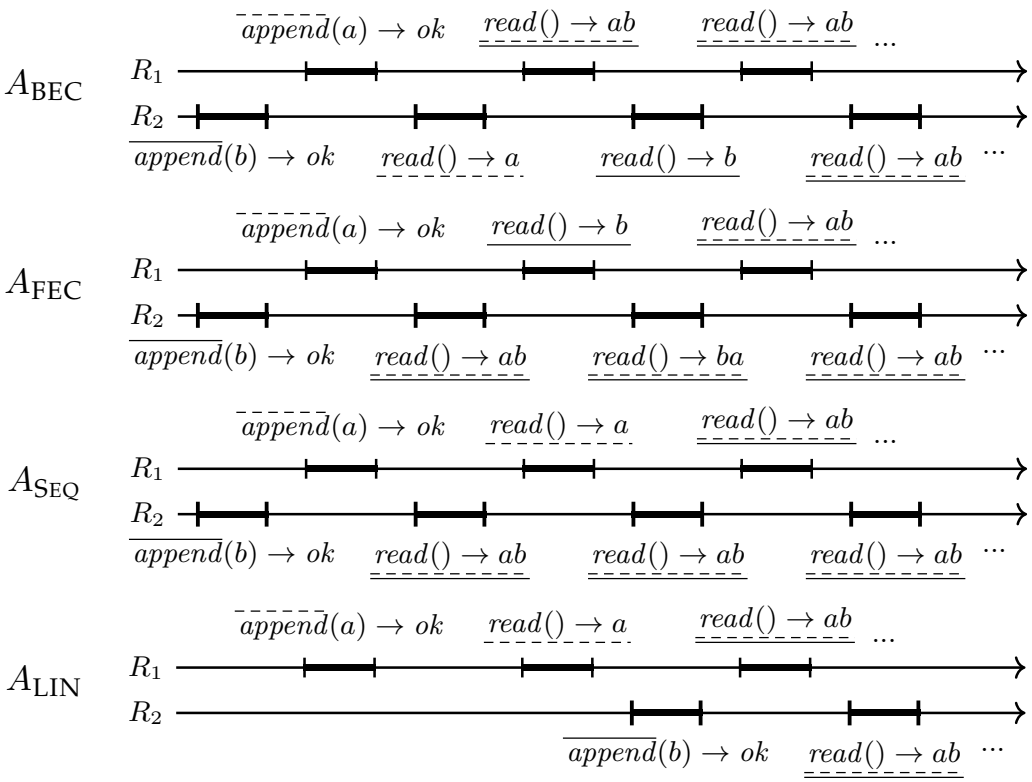


Figure 3.3: Example abstract executions of systems with a list semantics that satisfy BEC($\mathcal{F}_{seq}$), FEC($\mathcal{F}_{seq}$), SEQ($\mathcal{F}_{seq}$), and LIN($\mathcal{F}_{seq}$) respectively (for brevity, we omit the level parameter l and assume that all operations belong to the same class 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.

An example abstract execution A_{BEC} that satisfies BEC($\mathcal{F}_{seq}$) is shown in Figure 3.3. 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 [66]. Eventually, both $append(a)$ and $append(b)$ become visible to all subsequent $read()$ operations, thus satisfying EV.

By the definition of the $context$ function (Section 3.5), when A satisfies 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 RVAL($\mathcal{F}$). It is because weak operations (as shown in Section 2.2.3) might observe past operations in an order that differs from the final operation execution order (ar). Hence AcuteBayou does not satisfy BEC($\mathcal{F}$) for an arbitrary $\mathcal{F}$. But it could satisfy 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.

3.7.3 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:

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

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 3.1). 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 Figure 3.3. In A_{FEC}, replica R_2 temporarily observes the $append()$ operations in the order $append(b), append(a)$ which is different then the eventual operation execution order (as evidenced by the infinite number of $read() \rightarrow ab$ 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 Figure 3.3 satisfies both BEC and FEC, whereas A_{FEC} satisfies only FEC.

3.7.4 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

$$\begin{aligned}
\text{EV}(l) &\stackrel{\text{def}}{=} \forall e \in E : |\{e' \in L : e \xrightarrow{rb} e' \wedge e \not\xrightarrow{vis} e'\}| < \infty \\
\text{NCC}(l) &\stackrel{\text{def}}{=} acyclic(hb \cap (L \times L)) \\
\text{RVAL}(l, \mathcal{F}) &\stackrel{\text{def}}{=} \forall e \in L : rval(e) = \mathcal{F}(op(e), context(A, e)) \\
\text{BEC}(l, \mathcal{F}) &\stackrel{\text{def}}{=} \text{EV}(l) \wedge \text{NCC}(l) \wedge \text{RVAL}(l, \mathcal{F})
\end{aligned}$$

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.

3.7.5 Strong consistency

A common feature of strong consistency criteria, such as *sequential consistency* [50], or *linearizability* [52], 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) [50] 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 3.7.1). SINORD together with SESSARB imply NCC and EV [49], 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):

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

An example abstract execution A_{SEQ} that satisfies SEQ is shown in Figure 3.3. 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) [52] 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:

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

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

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 Figure 3.3. 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.

3.8 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 B.

As we have discussed in Section 2.4.1, 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 2.4.1 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 2.2.3, 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.

4
Limitations of mixed-consistency

We proceed to our central contribution: identifying key limitations of mixed-consistency systems represented as ACTs. In the previous chapter, we have shown that using Bayou one can obtain an ACT for any replicated data type $\mathcal{F}$. Naturally, a question arises whether it is possible to provide such a generic solution, but which avoids temporary operation reordering. Unfortunately, the answer is *no*. 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 Figure 3.2) 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.[1] Thus, a message must be RB-cast. Naturally, a replica may RB-cast several messages (although in practice a single message per operation should suffice) and may also TOB-cast some messages. 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 now proceed with the proof of the theorem. 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 Figure 3.2).

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 (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 $v = ba$.*

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

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. ■

Let us now 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* $read$ operation 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 $A'' \models \textsc{SinOrd}(strong) \wedge \textsc{BEC}(strong, \mathcal{F}_{seq})$. Then, for each k, by $\textsc{RVal}(strong, \mathcal{F}_{seq})$, $rval''(e''_{3k}) = \mathcal{F}_{seq}(read(), context(A'', e''_{3k}))$. Moreover, because of $\textsc{EV}(strong)$, 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. Firstly, 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 Acute-Bayou (for arbitrary $\mathcal{F}$) but one 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 showed earlier with ANNC.

In the following 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 in order to complete weak operations).

4.1 Other solutions

We use ACTs as an abstraction to represent mixed-consistency systems. We have specifically designed ACTs to hold the best features of both eventually consistent and strongly consistent systems. However, not all mixed-consistency solutions share the same characteristic. Thus, our impossibility result can be mitigated when some of these desirable features are traded off. Below we discuss some such solutions that can be found in the literature. We classify them in five broad categories.

4.1.1 Symmetric models with strong operations blocking upon a single crash

We start with *symmetric* mixed-consistency models, i.e., 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 which 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 [54] 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 [30] (see also Chapter 1). On the other hand, immediate operations offer stronger guarantees, but their implementation is based on three-phase commit [90], and thus requires all replicas to vote in order to proceed.

RedBlue consistency [35] 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 [36]. 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 [59] 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* [91], 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 [62], which makes our approach more general.

Finally, the model in [48] 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 2.2.6).

4.1.2 Symmetric Bayou-like models

In Section 2.2 we have already discussed the relationship between the seminal Bayou protocol [44] and ACTs.

In *eventually-serializable data service (ESDS)* [76], 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 [92] 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 [92] describe its behaviour using I/O automata. In [93] 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 [57] 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.

4.1.3 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 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* [37], 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* [47] as the target correctness criterion. In a subsequent work [38] 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)* [94] 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.

4.1.4 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 [39], Pileus [40], and also the widely popular contemporary cloud data stores, such as AmazonDB [31] and CosmosDB [32]. Systems that guarantee strong consistency within a single site and causal consistency between sites include Walter [95], COPS [86], Eiger [96] and Occult [97].

4.1.5 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* [34], which is a namespace completely separate from the one used for data accessed in a regular, eventually-consistent way. But then, the data kept in regular and strongly consistent buckets are completely isolated, which means that Riak lacks any meaningful mixed-consistency semantics (we could as well use two separate systems: one strongly consistent, one eventually consistent).

Apache Cassandra provides compare-and-set-like operations, called *light-weight transactions (LWTs)* [33], 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 [53], which means that mixed-consistency semantics of LWTs can be considered broken.

Various SQL database management systems (DBMS) feature several isolation levels, such as *read uncommitted*, *read committed*, *repeatable read*, *snapshot isolation* and *serializable* [98, 99], which correspond to different sets of guarantees on the types of phenomena that a transaction can witness when executing concurrently with other transactions. Hence DBMS can be considered mixed-consistency systems. However, these systems typically use extensive inter-replica synchronization based on locks to provide the aforementioned guarantees, and thus they are not highly available.

In Lynx [100] and Salt [101] 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 [102] 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 [103] 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 [104].

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* [105] 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 [106] (a strongly consistent system). In *MixT* [41] 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 [42] 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 thus block on network communication, thus they are not highly available.

5
Explicit failures modelling

In this chapter we extend the system model and provide a new formal framework, which allow us to formally study the correctness of highly available systems in the presence of failures. We begin with a simple example that justifes our approach.

5.1 Motivations and an example

Consider a simple system, in which each replica runs an implementation of a *last-write-wins register* $\mathcal{F}_{reg}$ (also called an *epidemic register* [49]; see the replicated data type specification in Figure 3.2, and the pseudocode in Algorithm 5). Clearly, the presented implementation is highly available as each replica responds to a client request immediately, without waiting for communication with other replicas. Every replica of the system has a copy of the register and allows clients to invoke two operations: $write(v)$, which stores a new value v in the register, and $read()$, which returns the current value of the register. When a replica's state changes, it sends a message to other replicas, so they can update their state accordingly. In order to guarantee that eventually the replicas converge to the same state, the replicas use timestamps and the last-write-wins policy [64]. More precisely, when $write(v)$ is invoked on some replica R_i (line 5), the replica saves v in its copy of the register, together with a unique timestamp that comprises of a logical clock [15] and R_i's identifier (line 12). Then R_i uses best-effort broadcast [107] to distribute v and the timestamp among other replicas (line 7). A replica updates its copy of the register only if the received timestamp is greater than the one corresponding to the current value stored by the replica (line 13). Many existing NoSQL data stores, such as Apache Cassandra (in its default configuration) [21] rely on a similar principle of operation.

It is easy to see that when neither replica crashes nor network splits are possible, this implementation indeed ensures eventual consistency, as defined by

Algorithm 5 Naïve Single Distributed Register Protocol for replica R_i

```
1: struct RegRec(clock : integer, rid : integer, value : Value)
2: operator <(o : RegRec, o' : RegRec)
3:     return (o.clock < o'.clock) ∨ (o.clock = o'.clock ∧ o.rid < o'.rid)
4: var myReg : RegRec
5: upon invoke write(value : Value)
6:     myReg = RegRec(myReg.clock + 1, i, value)
7:     BE-cast myReg
8:     return ok to client
9: upon invoke read()
10:    return myReg.value to client
11: upon BE-deliver(update : RegRec)
12:    if myReg < update then
13:        myReg = update
14:        // BE-cast update                                  // protocol fix
```

Vogels [19]: when updates cease eventually all $read()$ operations return the same value. Formally one could show that the implementation ensures BEC. However, as clearly follows from the executions in Figure 5.1, when replica crashes (top execution) or network splits (bottom example) are possible, the implementation no longer satisfies even the simple Vogels' definition of eventual consistency (not even for each network partition considered separately). In order to facilitate correct (intended by the programmer) behaviour when failures can occur some additional logic is necessary in the form of an anti-entropy protocol. In our example a replica could simply forward the received messages when applying the update, thus achieving *reliable broadcast* [107] for the update messages (see the fix in line 14). It is easy to see that the fix neither impacts safety (*nothing bad ever happens*) nor liveness (*eventually something good happens*) guarantees [67] [68] of the protocol when considering only failure free runs (the two versions, *with* and *without* the fix, are indistinguishable from the perspective of the clients, when no failures occur). However, when even a single failure might happen, the two versions of the protocol behave in a very different manner. More precisely, in all cases both versions satisfy the safety requirements (the reads return some correct values written earlier), but in spite of failures only the latter one satisfies the liveness requirements (convergence of the returned values).[1]

Our simple example showcases just two of many possible failure scenarios that need to be considered in order to ensure the protocol works as intended in real-life environments where failures are to be expected. The correctness analysis of such systems is further complicated when external clients, which can be mobile or stateless, are accounted for. Popular eventually consistent systems that we are aware of do feature various anti-entropy mechanisms that prevent them from exhibiting anomalies depicted in Figure 5.1. The framework introduced in this chapter allows us to formally study the correctness of highly avail-

[1] Note that if a system satisfies some safety guarantee in all failure-free runs, it can be shown that the system satisfies the guarantee also when failures do occur. It is because in finite executions crashed nodes are indistinguishable from very slow ones and network splits are indistinguishable from occurances of temporary communication delays. The same, however, does not hold for liveness guarantees, which can be violated only in infinite executions. Moreover, defining meaningful liveness guarantees that hold when failures occur is non-trivial.

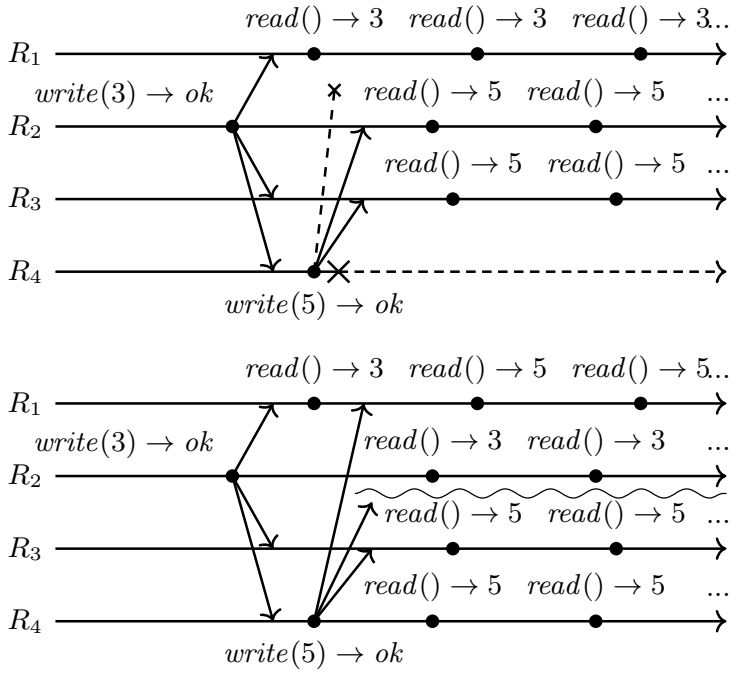


Figure 5.1: Example executions of a system implementing Algorithm 5. Solid arrows originating in events (dots) represent sent messages. Top execution: eventual transmission of a message is guaranteed only for the three correct (never crashed) processes. A crash of R_4 leads to inconsistent states of replicas. Bottom execution: a network split between $\mathcal{R} = \{R_1, R_2\}$ and $\mathcal{R}' = \{R_3, R_4\}$ (depicted using a wavy line) results in inconsistent states of replicas in $\mathcal{R}$.

able systems and, in particular, detect defects, such as as the one in Algorithm 5.

5.2 System model

Since our goal is to realistically model highly available systems facing failures, our approach somewhat deviates from the classic one. We consider a system consisting of *service replicas* (or simply *replicas*), connected via an asynchronous network, and external clients, which are routed to the replicas through load balancers. The key feature of our model is that a client's requests cannot be guaranteed to be always routed to the same replica. Below we outline our assumptions together with rationales behind them.

5.2.1 Replicas

Replicas form a set $\mathcal{R} = \{R_1, R_2, ..., R_n\}$, which can be divided into disjoint subsets $G_1, G_2, ...$, called partition groups. The groups represent sets of replicas located physically close to each other, e.g. in the same region, datacenter, or within the same availability zone inside of a datacenter (the significance of this

division will be discussed later). Replicas communicate with each other solely through message passing. Each replica has access to its own volatile memory as well as stable storage.[2] Data stored in the latter survive crashes and can be used by the replica for recovery. The replicas issue regular or synchronous writes to stable storage. Both kinds of writes can be interrupted by a crash, however the latter blocks the code execution on the replica until the written data is guaranteed to be persisted. For simplicity we assume full replication of application data (each replica holds all data necessary to serve any particular request). We later discuss an extension to a partial replication setting.

We consider three *replica failure models*: the *no-crash* (NC) model, in which no replica ever crashes, the *crash-stop* (CS) model, in which a replica can crash by stopping execution and ceasing all communication but never recovers, and the *crash-recovery* (CR) model, in which a replica can recover after crash by using, e.g., the data saved in its stable storage. In the CR model we discern between *transient* and *fatal failures*: after the latter one a replica never recovers. A hardware failure that causes permanent restarts that prevent the replica from completing any meaningful computation is also treated as a fatal failure. Formally, a replica that never crashes or experiences a finite number of (transient) failures is *correct*, otherwise it is *faulty*. Any number of replicas can be faulty. We expect the system to remain available (1) even if only a single replica is correct, and (2) in the case of *sticky clients* (explained later), if only a single replica in each partition group is correct. Once a replica receives a client request it starts to execute it, and unless the replica subsequently crashes, it returns a response without waiting for any external events.

We make no assumptions on relative speeds of the replicas and we assume no bounds on replica clocks skew. We consider *fair* infinite executions: each correct replica executes an infinite number of steps of the implemented algorithm and receives a never ending stream of client requests.

5.2.2 Clients

Clients are the abstractions through which users interact with the system, and which are responsible for passing user requests (possibly with some metadata) to the replicas in an appropriate format.[3] A client issues a single request at a time. A series of requests issued by a client forms a *session*. Sessions allow us to track potential dependencies between requests issued by the same user of the system. Clients may be *stateful*, or *stateless*. In particular a stateful client may represent, e.g., a desktop application maintaining a stable connection with the system. On the other hand, a stateless client may represent an application that cannot store state by design (e.g., due to performance considerations), or because of technical limitations (e.g., a web app in a browser with local storage and cookies disabled). A client may also loose state when the user's device is

[2] Stable storage may comprise of any technology which allows the replicas to persist data, such as HDDs, SSDs, or non-volatile/persistent memory (NVM/PM) [70].

[3] A client may also be used by another service. In that case, the service is the user of the system.

restarted without saving the state to stable storage, or when the user switches devices during a session.

In a classic replicated system model (e.g. [60]) all participating processes are equal, and there is no distinction between replicas and clients, or the clients are deemed to reside on the same physical nodes as the replicas and always communicate only with the local replica. In the system model in Section 2.4.1 we considered clients to be external to replicas, but we did not elaborate any further about their properties. Such treatment of clients was sufficient since we abstracted away from persistent communication blockages as well as we did not consider client-side guarantees based on sessions (session guarantees). On the other hand, in this chapter we elect to represent clients as external, completely independent entities to better reflect the client-server architecture used in practice. Moreover, such an approach has important consequences from the correctness point of view when client-side guarantees are considered. E.g., it is known that causal consistency is achievable in highly available systems in the classic model, whereas it is not achievable with external clients [108]. It is so even when the clients are stateful and cache all their requests and responses.[4] To achieve such guarantees clients would need to continously exchange information with replicas and other clients about other clients' requests, which would render them full replicas and which is impractical. Moreover, a client may not be able to maintain permanent connections with other processes.

We distinguish *mobile* and *sticky* clients. Mobile clients may issue their requests to any of the replicas from $\mathcal{R}$. On the other hand, sticky clients are associated with a single partition group G_i and issue their requests only to replicas from G_i. Our notion of sticky clients is different than in other works (e.g. [109]), where a sticky client issues all requests always to exactly the same replica. We discuss this difference below.

5.2.3 Clients – replicas interactions

Although a client issues only a single request at a time, when the replica does not respond fast enough the request may timeout allowing the client to issue the same request again to a different replica. Such a mechanism is necessary in a failure prone environment because otherwise, in case of a fatal crash of a replica, the client would remain blocked, which is against the spirit of high availability.[5] This is one of the reasons why we exclude from this model a notion of sticky clients which always connect to the same replica.

In practical replicated systems client requests are routed to the replicas through (hardware or software) load balancers, which are either external (load balancing through external DNS servers) or internal (dedicated devices or replicas themselves balance the load). The load balancers can be stateless (treating

[4]The same applies to some combinations of eventual consistency and the four classic session guarantees [44], and when the clients are stateless none of the classic session guarantees can be achieved.

[5]To satisfy high availability as defined in [18], [110] or [49], only correct replicas need to eventually respond, and clients connected to a crashed replica may remain blocked infinitely.

each request independently and assigning the requests to replicas in round-robin fashion), or stateful (maintaining information about client connections and routing requests of a given client always to the same replica). In general, stateful load balancing only works if the client maintains a stable connection with the system, but even then there is a problem with this scheme: the load balancers themselves may crash and lose state, or become unavailable. Thus, it is impossible in a highly available system to guarantee that every request from the same session reaches the same replica.[6] In practice, load balancing is often stateless by default and no attempt is made to route all requests within a session to the same replica (e.g., this is the case in Apache Cassandra [21], where replicas serve as the load balancers). This is the second reason why we exclude the notion of sticky clients issuing all requests to the same replica. We model load balancers only implicitly by their effect on request routing – i.e., the lack of control on the client side which replica will serve the issued request.[7]

However, we do discern between mobile, completely unbounded clients and sticky clients, which always connect to the same group of replicas, e.g. from the same geographical region. Such behaviour can be achieved by geo-sensitive load-balancing under the assumption that users do not cross geographical boundaries.

Let us now discuss a possible extension of the model to accommodate partial replication. If only certain replicas hold the necessary data to serve a specific request, then load balancing needs to take this fact into account. This can be achieved statelessly by utilizing *consistent hashing* [111] [20]. However, care needs to be taken with partial replication, because in each partition group G_i at least a single replica must hold the relevant data for each request. Moreover, since the single replica could crash, multiple replicas are required. Even then, the system is less resilient than a fully replicated one (which can tolerate up to $n - 1$ faulty replicas and remain available) and requires additional assumptions about failure patterns and the maximum allowed number of faulty replicas.

5.2.4 Network properties

We have so far strayed away from the network properties. In a typical asynchronous system model it is assumed that fair-loss links are available [107], which means that certain messages may be lost, but by utilizing stubborn retransmission it is possible to eventually contact every process. On the other hand, in the CAP conjecture [18] [110] a network is allowed to lose arbitrarily many messages. Thus, in the former approach only temporary network splits (or partitionings) can be modelled, whereas in the latter permanent network partitionings are also possible.

[6] Note that if the load balancers use a replicated state machine [55] and a consensus protocol to maintain the client connections data, then such an approach is not highly available. It is because before routing a new client's request to a replica, the load balancer would have to first consult other load balancers, and would block during a network split.

[7] More precisely, the system may try to route all requests from the same session to the same replica by maintaining stable client-replica connections, but there is no guarantee that this will succeed.

Note that, as we discuss in Section 2.2.5, with only temporary network splits, replicas might wait for any split to heal and communicate with each other before responding to a client request. In the system model in Section 2.4.1 we assume the existence of fair-loss links, and thus we admit only temporary network splits. However, we define high availability differently than in CAP. We formalize it as a replica design property (in a similar way as in [49, 58, 60]), which means a replica must be designed in such a way so as not to depend on communication with other replicas to generate a response to a client request. Thus, the impossibility postulated by CAP still holds. Still, distinguishing between the two types of network splits (temporary and permanent) can be useful as we discuss below.

Although in practice network splits are rare, they *do* occur, as shown by several studies that quantify network reliability in a rigorous manner [112] [113] [114] [115] (see also [109] for discussion on some high-profile cases). While most network failures are short-lived, some can take hours to resolve. Network splits may be caused by hardware failures or software issues, such as misconfigurations. The split may occur between datacenters or within a datacenter. In reality, network failure patterns can be complex. E.g., *partial partitionings* [115] [116] occur, in which two groups of replicas cannot communicate with each other, but are otherwise reachable from a third group of replicas (or by external clients).

We choose to model short-lived and long-lived network splits separately, as temporary and permanent, respectively. Even a single message loss can be considered a very short-lived network split. As systems are designed to cope with such minor failures, we classify a network split as temporary if its duration is short enough not to break any liveness expectations of the users. On the other hand, when the downtime caused by a network split is long enough to adversely affect client sessions (users' experience is compromised and as a result users decide to finish their sessions early), we classify the split as permanent (from the perspective of the interrupted session the split seems to last forever).

Thus, we consider the following two *network failure models*: the *temporary network partitionings* (TNP) model and the *permanent network partitionings* (PNP) model. The former corresponds to fair-loss links [107], while the latter is similar to the model assumed in the CAP conjecture, in which arbitrarily many messages can be lost. In the PNP model the set $\mathcal{R}$ of replicas can be divided into disjoint sets of replicas, $P_1, P_2, ..., P_k$, called *partitions* (or *final network partitions*). Replicas within a single partition maintain fair-loss links with each other. On the other hand, communication between replicas from different partitions may be possible for some time, but from some point on all messages will be lost. Thus, partitions represent the final state of connectivity between replicas in the limit at infinity.

The replicas may utilize point-to-point communications, as well as any other communication primitives that can be built upon them, such as best-effort broadcast, reliable broadcast, etc. (taking into account the limitations in connectivity between replicas). On the other hand, the replicas may not use communication primitives that require solving distributed consensus, such as total order broadcast.

When considering sticky clients which always connect to a specific partition group G_i, in the PNP model we assume that there exists P_j, such that $G_i \subseteq P_j$. In other words, network partitionings do not cross partition groups. Thus, if we define partition groups to represent separate datacenters, we model network splits between datacenters, but not inside of them. Then, the model allows us to compare guarantees provided to (1) clients that switch between replicas which cannot communicate with each other (mobile clients), and (2) clients that stick to replicas which can communicate with each other, but not with the rest of replicas (sticky clients).[8] Also, we assume that within each partition group G_i there is at least a single correct replica reachable by external clients.

5.2.5 Summary

By having three replica failure models and two network failure models, in total we consider six *failure models*: NC-TNP, NC-PNP, CS-TNP, CS-PNP, CR-TNP, CR-PNP. Additionally, we separately consider mobile and sticky, as well as stateful and stateless, clients.

5.3 Formal framework

Below we provide the formal framework that allows us to reason about correctness of highly available systems in executions in which failures occur. In a similar way as in Section 3 we extend the framework by Burckhardt *et al.* [48, 49]. To avoid repetition we cover below only the parts unique to this framework and we refer to Section 3 for common elements and preliminary definitions, such as e.g. *event graphs*.

5.3.1 Histories

To facilitate reasoning about failures we need to introduce two major changes to the way we represent execution *histories*. Firstly, we add the information about failures. Note that, a history represents a high-level view of a system as perceived by the clients. While the failures are not directly observable by clients,[9] we rely on this information to formalize the expected behaviour of the system. Secondly, we augment the way we represent client sessions to reflect their more complicated nature (such as timeouts on operations issued by clients). Instead of the *same session* equivalence relation ss, we utilize *session order* so, in which we allow a limited degree of branching to occur.[10] We also drop the lvl function, as

[8] If network splits can occur, e.g., between datacenters and inside of a datacenter, but not within a rack of servers, partition groups need to be adequately defined. On the other hand, if in this scenario certain clients are sticky at the regional or datacenter level, but not at the rack level, they need to be treated as mobile, and not sticky.

[9] E.g., a timeout on a client's operation does not necessarily result from a replica failure.

[10] Thus, so is more complex than simple intersection of *returns-before* and *same session* ($rb \cap ss$).

mixed-consistency semantics is orthogonal to the failure modelling we pursue in this chapter.

Formally, a *history* is an event graph $H = (E, op, rval, rb, so, sp, crash)$, 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),
- so, the *session order* relation, is a natural partial order over E, which specifies the ordering of operations executed within the same session,
- sp, the *same-partition* relation, is an equivalence relation, which groups events according to the final network partition in which they occurred,
- $crash : E \to \{true, false\}$, specifies if a particular event was executed on a replica that subsequently crashed ($true$), or not ($false$).

Note that, for some event $e \in E$, $crash(e) = true$ does not mean that, the replica on which e was executed, crashed during, or immediately after, the execution of e. Similarly, for any two events $a, b \in E$, $a \not\approx_{sp} b$ does not mean that replicas, which executed a and b, could not communicate with each other at the time these events were executed; the final permanent network split that separated the replicas might have happened later. Our definition of a history does not include the information about which event was executed on which replica. We rather give only indirect information, regarding which network partition was the replica located in, and whether the replica subsequently crashed or not.

Since replicas may crash shortly after receiving a request, just before the request is executed, or before the response is returned to the client, we need to consider a few edge cases. $rval(e) \neq \nabla$ means the response was generated by the replica, but this is fact is independent of whether the response was actually received by the client or not (e.g., because the message carrying the response was dropped and the replica did not retransmit it due to a crash, or because the client already issued the request to a different replica and was no longer interested in the response). On the other hand, $rval(e) = \nabla$ means that the replica has already started processing the operation, perhaps sending some messages to other replicas, but did not manage to produce any output (e.g., because of a crash). Finally, if a client sent a request, but the replica never received it, then there is no $e \in E$ pertaining to this particular request. Consecutive operations issued by the same client are ordered by the session order relation. If an operation timeouts, and the client issues the operation to another replica, there can be two concurrent operations within the same session, but the former one is abandoned and forms a *dead-end*, i.e., it is not followed by further operations in the session order.

We consider only *well-formed* histories, for which the following holds:

- if $|E| \not< \infty$, then $\forall e \in E : (\neg crash(e) \Rightarrow rval(e) \neq \nabla)$ (in infinite executions replicas, which do not crash, eventually respond),
- $\forall a, b \in E : (a \xrightarrow{rb} b \Rightarrow rval(a) \neq \nabla)$ (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 [49] [89]),
- $so \subseteq rb$ (session order respects the returns-before order; a client issues a request only when the previous one has finished),
- for all $e \in E$, the set $so^{-1}(e)$ is well ordered by the relation so (so is a union of trees),
- $\forall a, b, c \in E : (a \xrightarrow{so} b \wedge a \xrightarrow{so} c) \Rightarrow (op(b) = op(c) \wedge (so(b) = \emptyset \vee so(c) = \emptyset))$ (there is only limited branching in so due to timeouts),
- $\forall a, b, c \in E : (a \xrightarrow{so} b \wedge a \xrightarrow{so} c \wedge so(b) = \emptyset \wedge so(c) \neq \emptyset) \Rightarrow (c \xrightarrow{rb} b)$ (a client issues a request again only if it has abandoned the previous attempt),
- $|/ \approx_{sp}| < \infty$ (there is only a finite number of permanent network partitions).

5.3.2 Abstract executions

In accordance to the changes we applied to histories, we also modify *abstract executions* (we add sp relation and $crash$ function, but we drop the irrelevant in this context par).

An *abstract execution* is an event graph $A = (E, op, rval, rb, so, sp, crash, vis, ar)$, such that $(E, op, rval, rb, so, sp, crash)$ is some history H, vis is an acyclic and natural relation, and ar is a total order relation. For brevity, we often use a shorter notation $A = (H, vis, ar)$ and let $\mathcal{H}(A) = H$. The meaning of *visibility* (vis) and *arbitration* (ar) relations is the same as in Section 3.3.

5.3.3 Correctness predicates and replicated data type

We retain all the defintions pertaining to correctness predicates and replicated data types from the Sections 3.4 and 3.5 adjusted to the new definitions of histories and abstract executions. Specifically, the *operation context* is still a four element tuple which lacks return values, the returns-before relation, the information about sessions, and now it also lacks information about failures as well.

5.3.4 Basic eventual consistency

We now revisit the definition of *basic eventual consistency* (BEC) from Section 3.7.2 and adapt it to the new framework (other correctness criteria can be adapted analogously). We assume some abstract execution $A = (E, op, rval, rb, so, sp, crash, vis, ar)$.

Recall that BEC consists of three simple requirements:

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

The first requirement is the *eventual visibility* (EV) of events. The regular variant of EV requires that for any operation executed in an event $e \in E$, there is only a finite number of events in E that do not observe e. Due to replica crashes some operations may become pending, which means that the executing replicas did not manage to produce return values before a crash. Then, for such an operation executed in an event e, $rval(e) = \nabla$ even in an infinite history. Since the return value was not passed to the client it is not necessary to require the eventual visibility of this particular event e. On the other hand, it may be that before the crash the replica managed to share the information about the pending operation issued in e with some other replicas (in that case e may become visible to other events). To enforce convergence we require that once such an event e becomes visible to some other events eventually it has to become visible to all subsequent ones from some point on. Thus, the adapted definition of EV is formulated as follows:

$$\text{EV} \stackrel{\text{def}}{=} \forall e \in E : \big((rval(e) \neq \nabla \vee vis(e) \neq \emptyset) \Rightarrow |\{e' \in E : e \xrightarrow{rb} e' \wedge e \not\xrightarrow{vis} e'\}| < \infty\big)$$

The second requirement concerns avoiding circular causality. We define the auxiliary *happens-before* relation as before: $hb \stackrel{\text{def}}{=} (so \cup vis)^+$ (the transitive closure of session order and visibility), using the revised definition of so. We simply require *no circular causality*:

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

Finally, we specify *return value consistency* (RVAL). In the adapted version we release the pending operations from the obligation to return a value that can be explained using the specification of the replicated data type $\mathcal{F}$ and the operation's context:

$$\text{RVAL}(\mathcal{F}) \stackrel{\text{def}}{=} \forall e \in E : \big(rval(e) \neq \nabla \Rightarrow rval(e) = \mathcal{F}(op(e), context(A, e))\big)$$

BEC, as the name suggests, provides only very basic guarantees. It treats each operation independently, so it can be described as *client session agnostic*. If a client issues two operations op and op', op does not need to be visible to op'. Moreover, op (and op') might be visible to some subsequent operations, and then not be visible again. BEC only requires that after some time, there will be no more operations which fail to observe op (and op'). This mimics how stateless clients, switching between different replicas, may observe the incomplete process of update propagation. Even though BEC is such a weak correctness criterion, as we later show in Chapter 7, it cannot be satisfied when failures occur (even with the adaptations discussed above). Thus, it adequately captures the guarantees provided by eventually consistent systems only in the best case,

failure-free scenarios.

6
Client-side guarantees

In this chapter we discuss additional client-side guarantees called *session guarantees*. The four classic session guarantees [66] facilitate an intuitive and pragmatic programming model that builds on top of basic eventual consistency. Assuming some abstract execution $A = (E, op, rval, rb, so, sp, crash, vis, ar)$, they can be expressed in our model as:

1. *read your writes*: RYW $\stackrel{\text{def}}{=} so \subseteq vis$
2. *monotonic reads*: MR $\stackrel{\text{def}}{=} (vis; so) \subseteq vis$
3. *monotonic writes*: MW $\stackrel{\text{def}}{=}$ MWA $\wedge$ MWV, where
   - *monotonic writes in arbitration*: MWA $\stackrel{\text{def}}{=} so \subseteq ar$
   - *monotonic writes in visibility*: MWV $\stackrel{\text{def}}{=} (so; vis) \subseteq vis$
4. *writes follow reads*: WFR $\stackrel{\text{def}}{=}$ WFRA $\wedge$ WFRV, where
   - *writes follow reads in arbitration*: WFRA $\stackrel{\text{def}}{=} (vis; so^*) \subseteq ar$
   - *writes follow reads in visibility*: WFRV $\stackrel{\text{def}}{=} (vis; so^*; vis) \subseteq vis$

System architects typically optimize for *read your writes* (RYW) and *monotonic reads* (MR), as these are mostly anticipated by the users [117]. Thus, we mostly focus on these two session guarantees. RYW guarantees that each event e is visible to events that follow e in the same session. MR guarantees that events which are visible to any event e are also visible to events that follow e in the same session.

Unfortunately, as we have discussed in Section 5.2, in highly available systems classic session guarantees are difficult to provide (and cannot be provided altogether when the clients are stateless). Moreover, it is debatable just how important such guarantees are to the users and system architects. E.g., MR is usually described as important in the context of a webmail client: whenever a user has seen an email in their inbox, after a page refresh the email should not disappear. However, as common experience teaches us, the mainstream webmail

clients often forgo this guarantee and it is possible for a once visible message to become temporarily not available. On the other hand, it *is* imperative that if a message was once seen, that it *eventually* becomes visible, which is exactly the guarantee that BEC provides. In Section 7.1 we provide variants of RYW and MR that are ensured only eventually, called *eventual session guarantees*.

6.1 Context preservation

Although eventual session guarantees (implied by BEC) seem attractive, as they can be provided easily with stateless clients, they are not always sufficient for certain replicated data types, as we discuss below.

Let us start by considering a system that implements an observed-remove set (OR-set; see the specification of $\mathcal{F}_{orset}$ in Figure 3.2). An OR-set functions like a regular set but binds the $remove()$ operations with the $read()$ operations, so that a client can remove only those elements from the set, which it has observed. This way the client cannot accidentally remove elements that were concurrently added by other clients. Assume that a client received x in a response to a $read()$ operation ($read() \rightarrow S$ and $x \in S$) and then attempts to remove it. If the $read()$ and $remove(x)$ operations issued by the client are executed by two different replicas R_i and R_j, it might happen that the operation $add(x)$ that added x to the OR-set was visible to $read()$, but not to $remove(x)$ (R_j has not yet received the relevant update message). Then $remove(x)$ will take no effect (because according to the specification of $\mathcal{F}_{orset}$ only *observed* elements can be removed). Thus, without some form of a session guarantee the use of an OR-set leads to unintended behaviour of the system. In practice, this problem can be easily solved using client state. All elements added to the set can be tagged with some unique identifier. These identifiers are returned as metadata in $read()$ operations and stored as part of the client's state. When a client issues a $remove(x)$ operation, it passes x's identifier to the executing replica, which thus learns about x's existence (the $add(x)$ operation becomes visible to the $remove(x)$ operation).

This problem is even more evident in case of a system that implements a multi-value register (MVR; see the specification of $\mathcal{F}_{MVR}$ in Figure 3.2 and the example systems Dynamo [20] and Riak [23]). Unlike in a typical register, in an MVR concurrent $write()$ operations do not lead to a race condition. Instead, all values written concurrently (called *siblings* in Riak) are stored in the MVR and are returned to the client as a set in a $read()$ operation. A $write()$ operation that follows a $read()$ operation *logically* overwrites all siblings returned in $read()$, thus resolving the conflicts resulting from previous concurrent $write()$ operations. Clearly, a stateless client cannot bind the invocation of $read()$ and $write()$ operations on an MVR and each $write()$ creates a new sibling. Again some form of a session guarantee is necessary so an MVR can be used as intended. Such a session guarantee requires client state for metadata storage.[1]

[1]Such metadata can be efficiently maintained by using, e.g., dotted version vectors [80] [118],

Interestingly not always more is better: if a system provides classic session guarantees (or causal consistency), unintended behaviour may also ensue. Consider a system that, similarly to Dynamo and Riak, implements multiple MVR registers (a key-value store with MVRs as values), and an abstract execution $A = (E, op, rval, rb, so, sp, crash, vis, ar)$ in which two clients concurrently issue operations regarding registers x and y. The first client issues a following chain of operations: $read(x) \rightarrow \{u\}$, $read(y) \rightarrow \{v\}$, $write(x, u')$, in events e_1, e_2 and e_3, respectively (which means, e.g., that $op(e_1) = read(x)$ and $rval(e_1) = \{u\}$). The second client issues: $read(x) \rightarrow \{u\}$, $write(x, u'')$, in events e_4 and e_5. Both clients read u from x and want to overwrite it with a different value. Now, if e_5 occurs before e_2, it is possible that $e_5 \xrightarrow{vis} e_2$. This does not influence the return value in e_2, but the first client may obtain metadata that include information about u'' in x. Then, if MR is to be satisfied, $e_5 \xrightarrow{vis} e_3$ must hold. Thus, the write of u' to x will overwrite not only u, but also u'', and subsequent reads on x will return $\{u'\}$ instead of the intended $\{u', u''\}$.

Clearly, the relative visibility of events in case of an MVR needs to be carefully managed. The set of writes visible to some other write must correspond exactly to the writes that were visible to the previous read executed in the same session (and when considering multiple MVRs, on the same register). If additional writes are visible, they will be errorneously overwritten, and if insufficient writes are visible, unncessary siblings will be created. Assuming some abstract execution $A = (E, op, rval, rb, so, sp, crash, vis, ar)$, we can express this requirement through a new predicate that we call *context preservation* (CP):

$$\begin{aligned}\mathrm{CP}(\mathcal{F}_{MVR}) \stackrel{\text{def}}{=} \forall e, e' \in E, v \in \mathit{Values} :\\ (op(e) = read \wedge op(e') = write(v) \wedge e \xrightarrow{so} e'\\ \wedge \nexists e'' \in E : \big(op(e'') = read \wedge e \xrightarrow{so} e'' \xrightarrow{so} e'\big)\\ \Rightarrow vis^{-1}(e') = vis^{-1}(e) \cup \{e\})\end{aligned}$$

$\mathrm{CP}(\mathcal{F}_{MVR})$ explicitly defines the set of events visible to a $write()$ operation as the set of events visible to the most recent $read()$ operation performed by the same client, as well as the $read()$ operation itself. CP is incomparable with classic session guarantees (it is neither stronger, nor weaker). Note that slightly different definitions of CP are needed, e.g., for a key-value store of multiple MVRs, or an OR-set. Additionally, observe that for certain data types, no such guarantee is necessary. E.g., in a distributed last-write-wins register (see the specification of $\mathcal{F}_{reg}$ in Figure 3.2, and the implementation in Algorithm 5) all operations are independent and concurrency is never exposed to the client. Thus, an last-write-wins register can work correctly with stateless clients.

as in Riak.

7 Correctness in the face of failures

In this chapter we present a formal analysis of the behaviour of highly available replicated systems in the presence of failures. In Section 5.3.4 we have discussed that BEC is indeed a very weak correctness criterion. It may come as a surprise, then, that BEC is too strong to be satisfied when failures occur, as we discuss below. We also define correctness criteria that can be satisfied in a failure-prone environment, and then we show how to mitigate some of the undesired phenomena that are present in certain failure models.

For our discussion we assume some arbitrary system implementing a *non-trivial* replicated data type $\mathcal{F}$, i.e., $\mathcal{F}$ features at least one read-only operation, and one updating operation, which is *detectable* through the read-only operation. Formally, assuming some abstract execution $A = (E, op, rval, rb, so, sp, crash, vis, ar)$:

$$\exists op_r \in readonlyops(\mathcal{F}), op_u \in updateops(\mathcal{F}), e \in E : \\ (op(e) = op_u \ \wedge \ \mathcal{F}(op_r, (\emptyset, op, vis, ar)) \neq \mathcal{F}(op_r, (\{e\}, op, vis, ar)))$$

All practical types (including the ones in Figure 3.2) are non-trivial.

7.1 Network partitions and state convergence

We begin with the simple case of permanent network partitions and assume that every replica is correct and no crashes occur (the NC-PNP model). Recall the example from Section 5.1. Clearly, the implementation of a register provided in Algorithm 5 does not guarantee eventual visibility (EV) in case of network splits. Hence, it does not ensure system-wide state convergence as well, even with the proposed fix in line 14. The fix allows the replicas to converge within each network partition, but still some events executed within one (final) network partition will never become visible to events in other network partitions. Thus,

Algorithm 5 does not satisfy EV and, in consequence, also BEC($\mathcal{F}_{reg}$). A similar argument can be made for any system implementing a non-trivial replicated data type $\mathcal{F}$ (we include CS-PNP and CR-PNP models for completeness):

Theorem 6. *For any non-trivial $\mathcal{F}$, in the NC-PNP, CS-PNP and CR-PNP models, it is impossible to implement a highly available system that satisfies* BEC($\mathcal{F}$).

Proof. We conduct the proof by contradiction. Consider a system featuring two replicas R_1, R_2 and an execution with a network split that separates the two replicas from the very beginning. Client c_1 connects to R_1, while client c_2 connects to R_2. Firstly, c_1 issues an updating operation op_u. Then, both c_1 and c_2, take turns to issue in a continuous fashion a series of read-only operations op_r, such that op_u is detectable through op_r (c_1 issues operations on R_1, c_2 issues operations on R_2). Since no crashes occur, each invoked operation returns a response to the appropriate client. Let us call the depicted scenario the history $H = (E, op, rval, rb, so, sp, crash)$. If the system satisfies BEC($\mathcal{F}$), then there exists an abstract execution $A = (H, vis, ar)$, that satisfies BEC($\mathcal{F}$).

There is a single event $e_0 \in E$, such that $op(e_0) = op_u$ and infinitely many events $e_i \in E$, with $i \geq 1$, such that $op(e_i) = op_r$. For all $e_i, e_j \in E$, $i < j \Leftrightarrow e_i \xrightarrow{rb} e_j$. For each $e_i \in E$, with $i \geq 1$, let $i \equiv 1 \pmod 2$ if the operation executed in e_i was issued by c_2, and $i \equiv 0 \pmod 2$ if the operation executed in e_i was issued by c_1. Because of EV, there exists some $k \geq 1$, such that for each $i \geq k$, $e_0 \xrightarrow{vis} e_i$. Then, for each $e_i \in (E \setminus vis(e_0))$, $rval(e_i) = \mathcal{F}(op_r, context(A, e_i)) = \mathcal{F}(op_r, (\emptyset, op, vis, ar)) = v$, and for each $e_j \in vis(e_0)$, $rval(e_j) = \mathcal{F}(op_r, context(A, e_j)) = \mathcal{F}(op_r, (\{e_0\}, op, vis, ar)) = v'$, because of RVAL($\mathcal{F}$), and because read-only operations can be removed from a context without changing the expected return values. Note that, $v \neq v'$.

Now, let us consider an alternative history $H' = (E', op', rval', rb', so', sp', crash')$, in which client c_1 did not issue operation op_u, and the events initiated by c_2 are exactly the same as in H. Thus, $e_0 \notin E'$, each $e_{i>0} \in E'$, and op', $rval'$, rb', so', sp', $crash'$ when restricted to $E'' = \{e_i \in E' | i \equiv 1 \pmod 2\}$, are equivalent to their counterparts in H when similarly restricted, i.e., the events executed on R_2 are exactly the same. In particular for every $e'' \in E''$, $rval'(e'') = rval(e'')$. Such a history H' must exist, because the replicas in H were separated by a network split, and the events on R_2 occured independently of R_1 and c_1. Clearly, it must be possible for the system to produce history H', if it produces H, since H and H' are indistinguishable to R_2. Then, there must also exist an abstract execution $A' = (H', vis', ar')$, that satisfies BEC($\mathcal{F}$). By RVAL($\mathcal{F}$) and properties of read-only operations, for each $e' \in E'$, $rval(e') = \mathcal{F}(op_r, context(A', e')) = \mathcal{F}(op_r, (\emptyset, op', vis', ar')) = v$. But clearly, for some $e'' \in E'' \subset E'$, $rval(e'') = v' \neq v$. A contradiction. ∎

The above result is clearly related to the CAP theorem [18] [110], which states that it is impossible to achieve strong consistency in highly available systems in the presence of network splits. The proof provided by Gilbert and Lynch [110] uses linearizability [52] as the consistency criterion. On the other hand, our result concerns the impossibility of satisfying BEC, a much weaker correctness

criterion, and thus can be viewed as a strenghtening of the CAP theorem. Moreover, we consider arbitrary non-trivial types, and not only registers, as in CAP. In terms of proof techniques, Gilbert and Lynch base their proof on violation of a safety guarantee (linearizability's real-time requirement), whereas we rely on violation of a liveness property (EV). Thus, Gilbert and Lynch's proof requires only finite executions, whereas our proof utilizes infinite ones.

Interestingly, Burckhardt considers a variant of the CAP theorem in [49] implicitly assuming only temporary network partitions (so the NC-TNP model) and arrives at a different conclusion. He shows that for certain data types such as $\mathcal{F}_{reg}$ not only it is possible to achieve BEC($\mathcal{F}_{reg}$), but even sequential consistency (SEQ($\mathcal{F}_{reg}$)). However, he shows that under these assumptions it is impossible to satisfy linearizability (LIN($\mathcal{F}_{reg}$)). The above highlights just how important it is to clearly state all the assumptions. Since temporary network partitions do not constitute substantial obstacles for eventually consistent highly available systems, we do not focus on the TNP model alone (NC-TNP), but only in combination with CS or CR models.

The crux of the proof of our Theorem 6 is the impossibility to satisfy EV. Hence, we cannot expect the replica states to ever converge, as shown in Figure 5.1. However, in this system not only the states of replicas in different network partitions never converge, but the replica states never converge even within the same network partition. This behaviour could be easily prevented if Algorithm 5 featured the proposed fix in line 14.

Below we propose a variant of BEC with a weakened EV requirement, which can be satisfied under permanent network partitions, but which directly prohibits the unnecessary phenomena present in Algorithm 5 without the fix. We first formalize the weakened version of EV, called *partition-aware eventual visibility* (PAEV):

$$\begin{aligned}\text{PAEV} \stackrel{\text{def}}{=} \forall e \in E : (|\{e' \in [e]_{sp} : e \xrightarrow{rb} e' \wedge e \not\xrightarrow{vis} e'\}| < \infty \\ \wedge \forall p \in E/\approx_{sp} : (p \cap vis(e) \neq \emptyset \Rightarrow |\{e' \in p : e \xrightarrow{rb} e' \wedge e \not\xrightarrow{vis} e'\}| < \infty))\end{aligned}$$

PAEV states that for any event e, (1) if e is not pending,[1] it has to be visible to every event that happened from some point on within the same network partition as e, and (2) within every other network partition, if e has been observed by some event $e' \in p$ executed in that partition, it has to be visible to every event in p that happened from some point on. Note that if we consider an execution with only one network partition (no network splits), PAEV reduces to EV. Then, the weakened variant of BEC, called *partition-aware basic eventual consistency* (PABEC), is obtained by simply substituting EV with PAEV:

$$\text{PABEC}(\mathcal{F}) \stackrel{\text{def}}{=} \text{PAEV} \wedge \text{NCC} \wedge \text{RVAL}(\mathcal{F})$$

[1] On the other hand, when e is pending then by the definition of well-formed histories there are no events succeeding it in rb and thus $\{e' \in [e]_{sp} : e \xrightarrow{rb} e' \wedge e \not\xrightarrow{vis} e'\} = \emptyset$.

Clearly, our new criterion allows us to differentiate between algorithms, which strive to achieve state convergence without waiting for network splits to heal, from the ones that do not (so, e.g., Algorithm 5 with and without the fix, respectively).

Note that network splits lead to a phenomenon called *split brain syndrome*, in which mobile clients that switch between replicas from different network partitions can observe diverging system replies concerning some data. On the other hand, sticky clients are not directly affected by this phenomenon, but if they communicate with each other *outside of the system*, they can observe it indirectly.

Let us now briefly discuss the eventual variants of two key session guarantees: *eventually read your writes* (ERYW) and *eventually monotonic reads* (EMR). Below we state them formally:

$$\text{ERYW} \stackrel{\text{def}}{=} \forall e \in E : |\{e' \in so(e) : e \stackrel{vis}{\not\longrightarrow} e'\}| < \infty$$
$$\text{EMR} \stackrel{\text{def}}{=} \forall e \in E\ \forall e' \in vis^{-1}(e) : |\{e'' \in so(e) : e' \stackrel{vis}{\not\longrightarrow} e''\}| < \infty$$

ERYW requires that for each event e the number of events that follow e in the same session and which do not observe e is finite. On the other hand, EMR requires that when an event e observes some other event e', there is only a finite number of events e'' that follow e in the same session ($e'' \in so(e)$) that do not observe e'. Both ERYW and EMR are implied by BEC, but not by PABEC. Neither of them can be provided for stateless mobile clients in the PNP model:

Theorem 7. *For any non-trivial $\mathcal{F}$, in the NC-PNP, CS-PNP and CR-PNP models, it is impossible to implement a highly available system that ensures* $\text{PABEC}(\mathcal{F}) \wedge \text{ERYW}$ *or* $\text{PABEC}(\mathcal{F}) \wedge \text{EMR}$ *for stateless mobile clients.*

Proof. The proof is similar to the one for Theorem 6. Consider a history H as outlined in the proof of Theorem 6, with the difference that all operations are issued by the same client ($c_1 = c_2$). Note that, the client alternately issues operations to R_1 and R_2, even though they are in two different network partitions. This is naturally allowed, as we explicitly consider mobile clients. For all $e_i, e_j \in E$, $i < j \Leftrightarrow e_i \stackrel{so}{\longrightarrow} e_j$. Note that $so(e_0) = E \setminus \{e_0\}$. If the system satisfies $\text{PABEC}(\mathcal{F})$ with either of the two session guarantees, then there exists an abstract execution $A = (H, vis, ar)$, that satisfies $\text{PABEC}(\mathcal{F})$ with the respective session guarantee.

Because of PAEV, there exists some $k' \geq 1$, such that for each $i' \geq k' \wedge i' \equiv 0 \pmod 2$, $e_0 \stackrel{vis}{\longrightarrow} e_{i'}$ ($e_{i'}$ was executed on R_1).

If $A \models \text{ERYW}$, then there exists some $k \geq 1$, such that for each $i \geq k$, $e_0 \stackrel{vis}{\longrightarrow} e_i$ (e_0 has to be eventually visible, as the set $\{e' \in so(e_0) : e_0 \stackrel{vis}{\not\longrightarrow} e'\}$ has to be finite, and all operations are issued within the same session).

If $A \models \text{EMR}$, then there exists some $k \geq k'$, such that for each $i \geq k$, $e_0 \stackrel{vis}{\longrightarrow} e_i$ (since e_0 is observed by some $e_{i'}$, where $i' \geq k' \wedge i' \equiv 0 \pmod 2$, e_0 can be not observed only by a finite number of events $e_i \in so(e_{i'})$).

Now, we can conclude (similarly to the way we did in the proof of Theorem 6) that there is infinitely many events e_j, such that $e_0 \stackrel{vis}{\longrightarrow} e_j$, and that their return value v' is different than v (the return value for e_i such that $e_0 \stackrel{vis}{\not\longrightarrow} e_i$).

When we consider an alternative history H', where op_u has not been invoked, we can see it is indistinguishable from H for R_2: the stateless client issues the same operations, passing exactly the same information, and no messages from the other replica are delivered. Thus, the return values for operations executed on R_2 are the same in H and H', and infinitely many of them are v'. However, according to PABEC($\mathcal{F}$), and more specifically RVAL($\mathcal{F}$), the only allowed response is $v \neq v'$. A contradiction. ■

On the other hand, it can be shown that for sticky clients, which always connect to replicas from the same network partition, ERYW and EMR trivially hold when PABEC is satisfied. Moreover, stateful clients can even achieve the classic RYW and MR guarantees by caching requests and responses [108]. Note also, that context preservation (CP; see Section 6.1) can also be achieved in the PNP model, but using less resources than in case of RYW or MR.

7.2 Replica crashes and phantom operations

We continue our analysis by introducing replica crashes but not yet allowing crashed replicas to recover (the CS-TNP and CS-PNP models). Even without network splits, certain phenomena can occur, which we call *phantom operations*. We explain them first using an example.

Assume that a client issues an update operation to a faulty replica R_i, which responds with some return value (e.g., ok), and soon afterwards crashes. R_i might have tried to propagate the update to other replicas before the crash, but it could only do so asynchronously. Because of fair-loss links there is no guarantee of successful dissemination of messages when the sender fails. The client is then misled that the update operation was successfully completed (the system acknowledged the execution of the update), but there is no guarantee that it will be included in any future state of the replicas. Moreover, the client could communicate *outside of the system* with other entities and spread invalid information based on conviction that the update will eventually become visible. We call such an operation a *phantom operation*.

Phantom operations do not need to be confined to a single faulty replica, nor a single client. Multiple clients can observe the effects of a phantom operation op by, e.g., performing operations on the faulty replica after op was executed but before the replica crashed. Furthermore, a faulty replica R_i can manage to propagate the update to some other faulty replica R_j which crashes before successfully propagating the update to other replicas.

The common trait of the situations described above is that some update is acknowledged or observed, but then permanently lost due to a failure. In our framework this can be expressed as eventual *invisibility* of a particular event e: only a finite number of events, out of infinitely many, observe e. When there are no phantom operations (in an infinite abstract execution) the following predicate

(which we could call *phantom-freedom*) holds:

$$\text{XEV} \stackrel{\text{def}}{=} |E| \not< \infty \Rightarrow \forall e \in E : (rval(e) \neq \nabla \vee vis(e) \neq \emptyset \Rightarrow |vis(e)| \not< \infty)$$

Note that this visibility predicate is much weaker than EV (or even PAEV). Whereas EV requires that an event becomes visible to *all* subsequent events from some point on, XEV only requires the event to be visible to *some* (infinitely many) events, but still infinitely many other events may not observe it. We can define the weakest variant of BEC that is phantom-free as:

$$\text{XBEC}(\mathcal{F}) \stackrel{\text{def}}{=} \text{XEV} \wedge \text{NCC} \wedge \text{RVAL}(\mathcal{F})$$

Some acknowledged updates that were not propagated due to crashes are benign. E.g., if an update a was overwritten by a subsequent update b, then no information is lost. This is mirrored in the definition above, as for such an update a, we can always add artificial visibility arcs in the abstract execution (pretending that a was visible to every event which already observed b), in effect making a not a phantom anymore. Nonetheless, in principle we cannot avoid phantom operations if crashes occur in the CS model, as we state formally below:

Theorem 8. *For any non-trivial $\mathcal{F}$, in the CS-TNP and CS-PNP models, it is impossible to implement a highly available system that satisfies* XBEC($\mathcal{F}$).

Proof. The proof is similar to the one for Theorem 6. Consider a history H as outlined in the proof of Theorem 6, with the difference that all replicas are in the same partition (no network splits), and R_1 crashes immediately after sending the response for op_u back to the client c_1 (also c_1 does not issue the subsequent op_r operations). If R_1 has sent any messages to other replicas, we drop them in accordance with the properties of fair-loss links. We can now follow the same logic as in the proof of Theorem 6, and show that in any abstract execution A, the event $e_0 \in E$ needs to become visible to infinitely many $e_i \in E$, for $i \geq 1$, as otherwise visibility requirements would be violated (in this case e_0 would be a phantom operation). This eventually leads to contradiction, because R_2 cannot know about the existence of the event e_0. We skip the repetitive steps. ■

Corollary 2. *For any non-trivial $\mathcal{F}$, in the CS-TNP and CS-PNP models, it is impossible to implement a highly available system that satisfies* PABEC($\mathcal{F}$).

Corollary 3. *For any non-trivial $\mathcal{F}$, in the CS-TNP and CS-PNP models, it is impossible to implement a highly available system that satisfies* BEC($\mathcal{F}$).

The corollaries above follow directly from Theorem 8 and the definitions of PABEC($\mathcal{F}$) and BEC($\mathcal{F}$), respectively.

If BEC and PABEC cannot be satisfied by a highly available system in the face of failures, then what guarantees can such a system provide? The answer is that we cannot rule out all phantom operations, but we could require that there are no phantoms which are *not* caused by a replica failure. We formalize this intuition by proposing *crash-aware basic eventual consistency* (CABEC) for the TNP

model, and the more general *failure-aware basic eventual consistency* (FABEC) for the PNP model. They are based on the *crash-aware eventual visibility* (CAEV) and *failure-aware eventual visibility* (FAEV) predicates, respectively:

$$\begin{aligned}\text{CAEV} \stackrel{\text{def}}{=} \forall e \in E : (|\{e' \in E : e \xrightarrow{rb} e' \wedge e \not\xrightarrow{vis} e'\}| < \infty \\ \vee (crash(e) \wedge crash^{-1}(false) \cap vis(e) = \emptyset)) \\ \text{FAEV} \stackrel{\text{def}}{=} \forall e \in E \, \forall p \in E/\approx_{sp} : (|\{e' \in p : e \xrightarrow{rb} e' \wedge e \not\xrightarrow{vis} e'\}| < \infty \\ \vee (p \cap crash^{-1}(false) \cap (vis(e) \cup \{e\}) = \emptyset))\end{aligned}$$

CAEV implies that, unless an event e was executed on a replica which subsequently crashed and e was not observed by any other event on some replica that did not crash, it has to be eventually visible. FAEV means that for each event e and each network partition either e is eventually visible in that partition, or no event $e' \in p$ that occurred on a replica from that partition, and which does not crash, can observe e (or be e itself). Once e is observed by any event e'' that occurred on a replica which does not crash, e has to become eventually visible in the network partition in which e'' was executed. Then

$$\begin{aligned}\text{CABEC}(\mathcal{F}) &\stackrel{\text{def}}{=} \text{CAEV} \wedge \text{NCC} \wedge \text{RVAL}(\mathcal{F}) \\ \text{FABEC}(\mathcal{F}) &\stackrel{\text{def}}{=} \text{FAEV} \wedge \text{NCC} \wedge \text{RVAL}(\mathcal{F})\end{aligned}$$

When all replicas are correct CABEC reduces to BEC, and FABEC reduces to PABEC.

Note that because of crashes and phantom operations, it is impossible to ensure ERYW and EMR for stateless clients (both sticky and mobile), even when no network splits occur.

Theorem 9. *For any non-trivial $\mathcal{F}$, in the CS-TNP and CS-PNP models, it is impossible to implement a highly available system that ensures* CABEC$(\mathcal{F}) \wedge$ ERYW *or* CABEC$(\mathcal{F}) \wedge$ EMR *for stateless clients.*

Proof. The proof is based on the same principles as the proofs for Theorem 6, Theorem 7 and Theorem 8, but is more complex. We consider a history H as in the proof of Theorem 6, but with $c_1 = c_2$, and R_1 crashing after serving the response for op_u. However, contrary to the proof of Theorem 8, we do not crash R_1 immediately after executing op_u. Instead, we allow it to serve multiple op_r operations, all the time keeping R_1 and R_2 in two (temporary) network partitions, dropping all messages exchanged between them. We proceed to show that from some point on the op_r operations executed on R_1 start returning $v' \neq v$. To understand why this is the case we need to consider a couple alternative histories.

First, let us consider a history $H' = (E', op', rval', rb', so', sp', crash')$, in which R_2 crashes immediately without executing any event, while R_1 executes e_0 and infinitely many e_i, for $i \equiv 0 \pmod 2$ as in H from the proof of Theorem 6. Note that R_1 does not crash in H', so e_0 cannot be a phantom operation. Since $crash(e_i) = false$ for all $e_i \in E'$ (there are no events executed on the crashed R_2),

and because CABEC reduces to BEC in such cases, $H' \models \text{BEC}(\mathcal{F})$. Then, following the logic from the proof of Theorem 6, we can show that from some point on all op_r operations will return v'. Let us denote by e_k the first such event. Now, we create another alternative history H'' from H', by crashing R_1 immediately after e_k. Both R_1 and R_2 crash in H'', and thus it is a finite history.

Now we construct our target history H from H'' (which includes only events executed on R_1), by revoking the crash of R_2 and adding infinitely many executions of op_r on R_2 in the events e_i, for $i \equiv 1 \pmod 2$. H is indistinguishable from H'' for R_1, because in both histories the stateless client issues the same operations to R_1, and no messages are exchanged between replicas. Thus in H, (1) R_1 executes a finite number of events with the last e_k being an op_r returning v', and then crashes; (2) for the entire duration R_1 and R_2 are separated by (temporary) network split, and all messages between them are dropped; and (3) R_2 executes infinitely many events.

By the same logic as in the proof of Theorem 7, we can show that the ERYW and EMR session guarantees require e_0 to be eventually visible (through the event e_k in case of EMR), forcing all e_i from some point on to return v'. Then, we can show as in the proof of Theorem 6, that the only possible response for each e_i, for $i \equiv 1 \pmod 2$ is $v \neq v'$. Which concludes the proof with a contradiction. ■

7.3 Replica recovery and stable storage

Theorem 8 shows that phantom operations are unavoidable when a replica crashes after serving an operation submitted by the client, but before propagating the information about the operation to at least one correct replica. Naturally, in the CS model phantom operations cannot be avoided unless we sacrifice high availability and let the replica synchronize with other replicas before returning a response to the client. However, in the CR model, where replicas can recover after crash, we can avoid some of the phantom operations if only the information about the operations performed can be recovered after crash. To this end a replica has to perform a synchronous write to stable storage before returning a response to the client. Of course, a replica recovery is not possible in case of a fatal failure of the replica (e.g., a failure of the stable storage unit itself or the replica crashing and recovering infinitely many times). We formalize this intuition in the following three theorems:

Theorem 10. *For any non-trivial $\mathcal{F}$, in the CR-TNP and CR-PNP models, it is impossible to implement a highly available system that satisfies* XBEC($\mathcal{F}$).

Proof. The impossibility is due to fatal failures only, because as we argue later, transient failures can be tolerated. There are two kinds of fatal failures in the CR model: a crash after which the replica does not recover; and infinitely many crashes and recoveries of the same replica. In the former case, the same reasoning applies as in Theorem 8 for the CS model. In the latter case we can choose

the crashes to happen soon after recovery and drop all messages exchanged with that replica, thus forcing it into an infinite restart loop, in which it is unable to make any progress. Again, the same reasoning can be applied. However, note that it is sufficient to consider only the former case to prove the theorem. ■

Theorem 11. *For any non-trivial $\mathcal{F}$, in the CR-TNP and CR-PNP models, it is impossible to implement a highly available system that ensures* CABEC($\mathcal{F}$) ∧ ERYW *or* CABEC($\mathcal{F}$) ∧ EMR *for stateless clients.*

Proof. Just as in Theorem 10 the impossibility is due to fatal crashes, and we can apply the same reasoning as before. For a fatal crash where the replica does not recover, we follow the proof of Theorem 9 for the CS model. ■

Theorem 12. *For any non-trivial $\mathcal{F}$, in the CR-TNP and CR-PNP models, if the replicas do not issue synchronous writes to stable storage during the execution of some operations, but before returning the responses to the clients, it is impossible to implement a highly available system that satisfies* XBEC($\mathcal{F}$)*, even when no fatal failures occur.*

Proof. Assume that the system either does not issue synchronous writes to stable storage during the execution of any operation, or that it does so only after returning the response. Consider the same scenario as in the proof of Theorem 8 (R_1 crashes immediately after returning the response for op_u). We know that R_1 did not use stable storage synchronously during the execution of the event e_0, and if it did asynchronously, we declare all the issued writes to had not been persisted before the crash. There is no knowledge about the operation op_u issued by client c_1, neither in the stable storage of R_1, nor in the state of any other replica (no messages from R_1 were successfully transmitted, and the client c_1 communicated only with R_1). Now we can add to the execution R_1's recovery. Due to the lack of any recorded information about op_u, the state of R_1 after recovery is the same as if op_u was never issued by client c_1. Now, we can follow the same logic as in the proof of Theorem 8 to reach a contradiction. ■

Note that Theorem 12 *does not* require the system to issue synchronous writes to stable storage during the execution of *all* operations. The set of operations that require persistence depends on the semantics of $\mathcal{F}$. Still, synchronous writes are unavoidable in general and thus they constitute the inherent cost of eliminating phantom operations caused by transient failures.

Persistent storage solutions available in today's data centers primarily comprise of network storage devices based on magnetic disks (HDDs) and solid state drives (SSDs) (see, e.g., [119]). The HDD-based storage devices, which can handle around 7500-15000 random input/output operations per second (IOPS), are simply too slow to enable frequent synchronous writes. However, the SSD-based storage, which in recent years became much more affordable, can achieve 20-40 times the IOPS of HDD-based storage and a few times higher bandwidth (especially for write operations). It means that now the cost of performing synchronous writes for each client operation served is no longer prohibitive (unless a service running in the replicated environment must guarantee extremely

Algorithm 6 Protocol implementing replicated data type $\mathcal{F}$, for replica R_i

```
1: struct OpRec(clock : integer, rid : integer, op : ops(F))
2: operator <(o : OpRec, o' : OpRec)
3:     return (o.clock < o'.clock) ∨ (o.clock = o'.clock ∧ o.rid < o'.rid)
4: function opFun(o : OpRec)
5:     return o.op
6: function makeContext(operations : set⟨OpRec⟩, vis : set⟨OpRec × OpRec⟩)
7:     var ar = {(o, o')|o, o' ∈ operations ∧ o < o'}
8:     return (operations, opFun, vis, ar)
9: var operations : set⟨OpRec⟩
10: var visible : set⟨OpRec × OpRec⟩
11: upon invoke(op : ops(F))
12:     var rval = F(op, makeContext(operations, visible))
13:     var o = OpRec(max(operations).clock + 1, i, op)
14:     visible = visible ∪ (operations × {o})
15:     operations = operations ∪ {o}
16:     write operations and visible synchronously to stable storage
17:     BE-cast(UPDATE, operations, visible)
18:     return rval to client
19: upon BE-deliver(UPDATE, recOperations : set⟨OpRec⟩, recVisible : set⟨OpRec × OpRec⟩)
20:     if recOperations \ operations ≠ ∅ then
21:         operations = operations ∪ recOperations
22:         visible = visible ∪ recVisible
23:         write operations and visible synchronously to stable storage
24:         BE-cast (UPDATE, recOperations, recVisible)
25: upon recovery
26:     initialize operations and visible from stable storage
27:     BE-cast (RECOVERY, operations, visible)
28: upon BE-deliver(RECOVERY, recOperations : set⟨OpRec⟩, recVisible : set⟨OpRec × OpRec⟩)
29:     if operations \ recOperations ≠ ∅ then
30:         BE-cast (UPDATE, operations, visible)
31:     if recOperations \ operations ≠ ∅ then
32:         operations = operations ∪ recOperations
33:         visible = visible ∪ recVisible
34:         write operations and visible synchronously to stable storage
```

low latencies in serving client requests). The performance penalty due to frequent writes to stable storage is likely to further drop with the adoption of novel technologies, such as byte-addressable non-volatile memory (also called persistent memory) [70] which promises performance that is almost on par with RAM [120].

We make one final observation for a system that uses stable storage to avoid some phantoms:

Theorem 13. *For any $\mathcal{F}$, it is possible to implement a highly available system that, if no fatal failures occur, satisfies* BEC($\mathcal{F}$) *in the CR-TNP, and* PABEC($\mathcal{F}$) *in the CR-PNP model, and if fatal failures occur, satisfies* CABEC($\mathcal{F}$) *in the CR-TNP, and* FABEC($\mathcal{F}$) *in the CR-PNP model.*

Proof. In order to prove the above, we need to show that we can propose an implementation that satisfies FABEC for a generic type $\mathcal{F}$. As an illustration, consider Algorithm 6, which shows a pseudocode of a generic protocol for an arbitrary $\mathcal{F}$. Note that, Algorithm 6 is overly simplistic and not optimized for

performance (e.g., it does not distinguish between read-only and updating operations, and does not minimize the size of exchanged messages, nor the amount of data kept in stable storage).

An $OpRec$ represents the quanta of information about invocation of a single operation (line 1). An $OpRec$ is a tuple which consists of the operation op invoked, the identifier rid of the replica that executes op, and $clock$ (the value of the a logical clock maintained by the replica at the time of the invocation of op). $OpRec$ structures can be totally ordered using the $clock$ values and replica rids (line 2).

Each replica maintains two data structures: $operations$ and $visible$ (lines 9-10). They are used upon operation invocation to create the operation context, as required by the function $\mathcal{F}$ (line 6). The $operations$ set stores information about the operation invocations the replica is aware of. On the other hand, the $visible$ set is used by the replica to maintain information about the relative visibility of such events. A pair (o, o') belongs to $visible$, iff o' observes o.

Upon invocation of operation op, we calculate the return value $rval$ using $\mathcal{F}$ and the appropriately created operation context (line 12). Then the replica needs to update its state and notify other replicas about the execution of op. To this end, a new $OpRec$ o is created (line 13). The value of $o.clock$ is chosen so that it is larger than the $clock$ field of any other $OpRec$ in the $operations$ set. Next, we extend the $visible$ set so that o observes all the o' in the $operations$ set (line 14). Then we add o to the $operations$ set (line 15). Next we write both $operations$ and $visible$ to stable storage (line 16). Finally, both the $operations$ and $visible$ sets are broadcast to all replicas using best-effort broadcast in an UPDATE message (line 17).

Upon receipt of an UPDATE message (line 19), when necessary, the replica updates its $operations$ and $visible$ sets by merging them with the incoming ones, writes both $operations$ and $visible$ to stable storage and finally broadcasts a message with the new state. The last two steps are necessary to ensure FAEV.

Upon recovery (line 25) the replica initializes its $operations$ and $visible$ sets from the stable storage and broadcasts a RECOVERY message. A RECOVERY message from a replica R_i has a double purpose:

- R_i ensures that other replicas will also receive the operations R_i performed and saved to its stable storage (but perhaps failed to disseminate), and
- upon receipt of a RECOVERY message (line 28), other replicas R_j will resend to R_i all operations that R_i might be missing.

For each fair execution (corresponding to some history $H = (E, op, rval, rb, so, sp, crash)$) of a system that implements Algorithm 6, we need to show that there exists an abstract execution $A = (H, vis, ar)$, such that $A \models \text{FABEC}(\mathcal{F})$. It is because if there are no fatal failures, FABEC reduces to BEC in the TNP model and PABEC in the PNP model. If there are fatal failures, FABEC reduces to CABEC. Instead of considering all isomorphic histories, we consider only histories for which E contains elements of $OpRec$ type, which were constructed according to the pseudocode of Algorithm 6 in the actual execution.

For simplicity we assume, that whenever a message is BE-cast in the pseudocode, it is scheduled for sending, but it is actually sent only after the entire code block in which BE-cast occurs finishes execution.

First, let us introduce some auxiliary definitions. For an event $e \in E$, we denote by $pre(e)$ and $post(e)$ the volatile state of the replica, respectively, just before the execution of e, and just after the execution of e. Similarly, by $sspre(e)$ and $sspost(e)$ we denote the contents of stable storage of the replica executing e, before and after the execution of e. Note that $pre(e) = sspre(e)$, but not always $post(e) = sspost(e)$ (e.g., when the replica executing e crashes before completing the execution). Moreover, $sspre(e).operations \subseteq sspost(e).operations$ and $sspre(e).visible \subseteq sspost(e).visible$. Similarly, for any two events $e, e' \in E$ executed on the same replica in that order, $sspre(e').operations \subseteq sspost(e).operations$ and $sspre(e').visible \subseteq sspost(e).visible$.

To construct A, for any $a, b \in E$, we let $a \xrightarrow{ar} b \Leftrightarrow a < b$ and $a \xrightarrow{vis} b \Leftrightarrow a \in pre(b).operations$. Now we need to show that A satisfies FAEV, NCC and RVAL($\mathcal{F}$).

Let us make an observation. For an event e executed on some replica R_i, if $rval(e) \neq \nabla$ it means that $e \in sspost(e).operations$. Similarly, if $vis(e) \neq \emptyset$, and thus there exists $e' \in E$, such that $e \xrightarrow{vis} e'$, it means that e must have been broadcast and delivered by some other replica R_j, or e' is some subsequent event executed on R_i. In either case $e \in sspost(e).operations$.

When e is recorded in stable storage of some correct replica, eventually it will be recorded in stable storage of each correct replica in the same partition. A replica R_i saves e to stable storage in two cases: either it executed $op(e)$ locally or received e in some UPDATE or RECOVERY message. In either case, R_i will broadcast e as part of its $operations$ set. Since R_i is correct and it uses best-effort broadcast, every correct replica R_j, which belongs to the same network partition as R_i, will eventually deliver the message, and if R_j does not already have e in its $operations$ set (which is persisted on stable storage), R_j will add e to its $operations$ set and write it to stable storage.

For any partition $p \in E/\approx_{sp}$, any $e, e' \in E$, such that $e' \in p \wedge e \in pre(e').operations \wedge \neg crash(e')$, it holds that there is only a finite number of events $e'' \in p$, such that $e \notin pre(e'').operations$. Therefore, $|\{e'' \in p : e \xrightarrow{rb} e'' \wedge e \not\xrightarrow{vis} e''\}| < \infty$. On the other hand, for an event $e \in E$, for which there does not exist such an event $e' \in p$, naturally $\big(p \cap crash^{-1}(false) \cap (vis(e) \cup \{e\}) = \emptyset\big)$, since $\big(p \cap crash^{-1}(false) = \emptyset\big)$. Thus, $A \models$ FAEV.

Now let us focus on no-circular-causality (NCC). Observe that, for any two events $a, b \in E$ executed on the same replica in that order, $a \xrightarrow{rb} b$. Moreover, the same holds also for any two events $a, b \in E$, such that a was executed on R_i, b was executed on R_j, $R_i \neq R_j$, and $a \in pre(b).operations$. This is so, because for a to be included in the state of R_j, a must have been BE-cast by R_i in an appropriate message already after the execution of a has finished. Thus, $vis \subseteq rb$. Additionally, $so \subseteq rb$, by well-formedness of a history. Therefore, $hb = (so \cup vis)^+ \subseteq rb$, and since rb is a partial order, hb is acyclic, and $A \models$ NCC.

Finally, we turn our attention to RVAL($\mathcal{F}$). The return value for each not

	no-crash	crash-stop	crash-recovery	
			only transient failures	all failures
temporary network partitions	BEC	CABEC	BEC	CABEC
permanent network partitions	PABEC	FABEC	PABEC	FABEC

Figure 7.1: The possible to ensure correctness criteria in various failure models for a highly available system that implements an arbitrary (non-trivial) replicated data type $\mathcal{F}$.

		no-crash	crash-stop	crash-recovery	
				only transient failures	all failures
temporary network partitions	(all clients)		○ ◉ ≰		○ ◉ ≰
permanent network partitions	(all clients)	◐	○ ◉ ≰ ◐	◐	○ ◉ ≰ ◐
	(mobile clients)	◉ ≰		◉ ≰	

○ – phantom operations
◉ – no ERYW (eventually read your writes) for stateless clients
≰ – no EMR (eventually monotonic reads) for stateless clients
◐ – split brain syndrome for clients who communicate outside the system

Figure 7.2: Phenomena observable by the clients in various failure models for a highly available system that implements an arbitrary (non-trivial) replicated data type $\mathcal{F}$.

pending event $e \in E$ is computed using the function $\mathcal{F}$ itself. We need to show that the output $C' = (E', op', vis', ar')$ of the function $makeContext$ is isomorphic with $C'' = context(A, e) = (E'', op, vis, ar)$. Firstly, by definition $vis^{-1}(e) = pre(e).operations$, thus $E' = E''$. Secondly, for each $e' \in E'$, $opFun(e') = e'.op$, and $e'.op = op(e')$. Thus, for each $e' \in E'$, $op'(e) = op(e)$. Thirdly, for any three events $a, b, c \in E$, if $a \in pre(b).operations \wedge a, b \in pre(c).operations$, then $(a, b) \in pre(c).visible$, because the sets $operations$ and $visible$ are always modified, persisted and disseminated together atomically. Thus, for any two $a, b \in E'$, such that $a \xrightarrow{vis} b$, $(a, b) \in pre(e).visible$, which means that $vis' = vis|_{E'}$. Fourthly, $(a, b) \in ar' \Leftrightarrow a, b \in E' \wedge a < b$, and $(a, b) \in ar \Leftrightarrow a, b \in E \wedge a < b$. Thus, $ar' = ar|_{E'}$. Finally, since $E' = E''$, while op', vis' and ar' are restrictions of op, vis and ar to E', C' and C'' are isomorphic, and $A \models \text{RVAL}(\mathcal{F})$.

To conclude, since $A \models \text{FAEV}$, $A \models \text{NCC}$ and $A \models \text{RVAL}(\mathcal{F})$, $A \models \text{FABEC}(\mathcal{F})$.

■

Corollary 4. *For any non-trivial $\mathcal{F}$, in the CR-TNP model, if no fatal failures occur, it is possible to implement a highly available system that satisfies* $\text{BEC}(\mathcal{F})$.

The above corollary follows directly from Theorem 13.

7.4 Summary

In Figures 7.1 and 7.2 we summarize the consistency guarantees which are possible to achieve in highly available systems and the artifacts a client can observe in various combinations of replica and network failure models. In terms of the offered guarantees and the types of artifacts which the clients can encounter, the crash-recovery (CR) model with only transient failures is akin to the no-crash (NC) model. When we admit fatal failures in CR model, we achieve the same guarantees and types of artifacts as in the crash-stop (CS) model.

Note that the formal framework can be easily extended for a particular class of systems. E.g., one could use our approach to define a whole family of failure-aware consistency criteria, based on other baseline predicates, such as causal consistency.

8
Related work

In this chapter we present work relevant to our research. We start with various definitions of high availability. Then, we discuss eventual consistency and other consistency guarantees. Next, we focus on work concerning limitations and tradeoffs present in highly available systems. Finally, we survey the most influential highly available system designs.

We cover mixed-consistency models in detail separately in Section 4.1.5.

8.1 High availability

(High) availability was first defined as a formal guarantee for replicated systems by Brewer in his CAP conjecture [18], in which it stipulates that eventually, for every request, a response needs to be provided. However, high availability was already recognized as an important feature of eventually consistent systems earlier, e.g. in the design of Bayou [44] and similar early eventually consistent systems, or in *BASE (Basically available, soft state, eventually consistent* semantics [121].

As we discuss in Section 2.2.5, when the network paritions are permanent, high-availability as proposed by Brewer is akin to wait-freedom [77], the fact which was also noted by Gilbert and Lynch in their paper concerning CAP [110].

Bailis *et al.* [109] focus on high availability in the context of transaction processing systems, where they define a few variants of high availability, such as *sticky availability*, *replica availability* and *transactional availability*. As we discuss in Section 5.2, their notion of sticky availability is different than ours when considering sticky clients.

More recently (see, e.g., [49, 58, 60]) high availability was modelled as a design property in which system replicas are required to respond to client requests immediately without synchronous communication with peers. We follow this approach.

8.2 Eventual consistency

As follows from the CAP theorem, highly available systems can only guarantee some form of *eventual consistency*. While eventual consistency has been considered in the context of replicated databases much earlier [64, 122, 44], these approaches lacked clearly defined consistency requirements (they only describe eventual consistency in operational terms). On the other hand, early definitions of eventual consistency that followed [19, 43] are rather informal: they stipulate that in the absence of updates, eventually the read operations on the same object on all processes will return the same value. Such a definition (sometimes classified as *quiescent consistency* [49]) makes eventual consistency a pure liveness property, as it does not impose any restrictions on the possible responses when updates continue to be performed. In particular, according to this definition, a read of an object can return a value that was never written to it.

Shapiro *et al.* proposed *strong eventual consistency* (SEC) [26, 27] as a target correctness criterion for CRDTs. It requires any two replicas that receive the same set of messages to be in the same state. While this definition improves somewhat on the earlier approach, it still does not guarantee that the responses returned by the replicas are explainable by the semantics of the implemented data type (e.g., a replica of a register which always returns value 0 in every state is correct according to SEC; e.g., consider Algorithm 5 with line 10 replaced by `return 0`). In our approach, we avoid this problem by utilizing the replicated data type specifications [28] to bind the allowed responses with the history of previously executed operations. Moreover, unlike in case of SEC, which is defined solely on replica states, we consider external clients and the guarantees provided to them, which is a more robust approach. Although defining consistency models over internal replica states seems convenient and easy to follow, it is the externally observable behaviour of the system that really matters. Additionally, Shapiro *et al.* assume that every step a CRDT algorithm performs is synchronously logged to stable storage. This assumption is a very strong one and it does not reflect the way CRDTs are usually implemented (as lightweight, in-memory data structures).

The use of replicated data type specification also lays at the core of *basic eventual consistency* (BEC), which was proposed by Burckhardt *et al.* [48, 49]. BEC abstracts away from implementation details such as internal replica states or exchanged messages. We closely follow this work as we base fluctuating eventual consistency (FEC) and our family of failure-aware correctness criteria on BEC. In [49] Burckhardt formally specified a number of eventually consistent protocols. However, the correctness proofs for these protocols, as well as proofs for other formal results, only consider system runs in which no failures occur.

Eventually consistent linearizability [2] can be considered a precursor to the crash-aware basic eventual consistency defined in Section 7.2, as it admits artifacts such as phantom operations. However, it is not a failure-aware criterion: it admits such phenomena also in failure-free runs.

Eventual linearizability, first defined in [45] and later refined in [46] models intermittent inconsistencies (as in self-stabilizing systems) that may occur in a system only up to certain point in time (measured in time or number of events from the beginning of computation). As noted by the authors, providing eventual linearizability is as difficult as providing linearizability itself, because, from some moment on, all processes have to work in a fully coordinated fashion (no further inconsistencies are allowed). Thus, eventual linearizability is in fact very similar to strongly consistent correctness criteria and requires an asynchronous system model to be augmented with the same failure detector Ω, as in case of strong consistency, in order to be achievable [123].

Finally, Δ-*atomicity* [124] and Γ-*atomicity* [125] can also be used to describe the consistency model of eventually consistent data stores. However, both Δ-atomicity and Γ-atomicity, as well as any other recency guarantees cannot be enforced when failures occur.

8.3 Causal consistency and session guarantees

Causal consistency is a popular consistency model employed in many replicated systems and CRDT implementations. It can be considered to be a stronger form of eventual consistency. In its original formulation [126], inspired by *causal broadcast* [127], it was restricted only to read-write registers, and lacked built-in conflict resolution for updates which means it did not guarantee convergence. Convergent conflict handling was delegated to the application. A revised definition which explicitly guarantees convergence was provided by Burckhardt *et al.* in [47].

Several strengthenings of causal consistency have been proposed, most notably *causal+ consistency* [86], *natural causal consistency* [128], and *observable causal consistency* (OCC) [58]. *Parallel snapshot isolation* [95] proposed as an isolation property for replicated databases is a variant of causal consistency, which solves conflict resolution by aborting conflicting transactions.

Causal consistency is achievable in highly available systems assuming a classic asynchronous message-passing system. In fact OCC is claimed to be the strongest correctness criterion achievable in highly available systems [58, 60]. Since it is stronger than eventual consistency, but weaker than strong consistency, it is often advocated to be an optimal correctness criterion for highly available systems [86, 58]. However, causal consistency is known to be costly to achieve in practice [62], and is not always needed [63]. Moreover, as discussed in Section 5.2, it is *not achievable* in highly available systems when external clients are considered [108].

The four classic *session guarantees* [44], discussed in detail in Chapter 6, feature similar guarantees to causal consistency. In fact the four session guarantees combined with eventual consistency imply causal consistency [129, 49]. Early eventually consistent systems, which had clients collocated with replicas, could

provide these guarantees with ease. With external clients session guarantees can be trivially implemented by requiring clients to always communicate with a quorum of replicas, albeit this approach is not highly available. On the other hand, some session guarantees can be provided when the clients cache their writes and reads [108]. In turn, signifcant use of client-side resources is necessary, and thus this approach is typically avoided. The popular NoSQL systems often feature tunable consistency levels, which means they can be configured to operate either in the highly available mode, with no session guarantees, or utilizing a quorum of replicas to achieve stronger consistency at the cost of high availability.

In Section 6.1 we show that enforcing causal consistency or the classic session guarantees can be actually counterproductive for certain replicated data types. Instead we provide a novel substitute called *context preservation* and we define *eventual session guarantees*, which can be obtained with stateless clients in certain failure modes without compromising on high availability.

8.4 Limitations of highly available systems

In this section we revisit some results concerning limitations of highly available systems, i.e., work that is most relevant to ours.

Naturally, the most important result is the CAP theorem [18, 110], which established the impossibility of achieving strong consistency in highly available systems. It has been especially influential in driving the design choices of many highly available eventually consistent systems. Over the years several follow up articles appeared (e.g., [130, 131, 132]) discussing its relevance. We directly relate our work to the CAP theorem in Section 7.1.

Attiya *et al.* [58] [60] show that an eventually consistent data store implementing multi-valued registers (MVRs) [20] cannot provide stronger guarantees than already mentioned observable causal consistency (OCC). Thus, contrary to CAP, this result constitutes a tight upper bound, i.e., it defines not only which guarantees are not achievable (stronger than OCC), but also which are. However, as discussed above, causal consistency, and by extension OCC, are achievable only when no external clients are considered. The authors completely ignore replica crashes and assume the availability of fair-loss links, and thus consider only temporary network splits.

Mahajan *et al.* [128, 133] formulate the CAC theorem which shows the tradeoff between consistency, availability and convergence in distributed systems. The authors prove that *natural causal consistency* (a variant of causal consistency in which the causal order of operations needs to respect additional real-time constraints) is the strongest guarantee that can be achieved for a certain class of highly available data stores. The considered data stores are assumed to feature only (invisible) read and write operations and no particular conflict resolution policy (so they might expose concurrent writes to the clients as in MVRs

[20]). An additional requirement on the data store implementation states that a pair of processes must be able to converge to a single state in two steps of one-way communication (provided no intermediate communication with other processes). The authors consider *omission-failure* model, thus they assume permanent network splits, but no replica crashes and recoveries. The authors also consider a Byzantine failure model but do not provide a tight upper bound in this model. The introduced liveness property of *convergence* is state-based and relies on explicit message exchanges, and so is similar to SEC. Mahajan *et al.* also do not consider external clients.

Both Attiya *et al.* and Mahajan *et al.* consider only data stores implementing MVRs, and not arbitrary replicated data types, as in our case.

Bailis *et al.* [109] as already mentioned focus on high availability in the context of transaction processing systems. The authors analyze which database isolation properties are compatible with high availability, and which are not. In terms of failures, they consider network splits, however, they largely abstract away from server crashes. They do note that certain replication schemes may become unavailable due to crashes, e.g., when a transaction coordinator fails, or that certain fault-tolerance requirements are incompatible with high availability, but do not explore the topic further. On the other hand, the protocol for handling session guarantees they provide is inherently blocking and does not guarantee progress in spite of crashes, which seems ill-suited as a solution aimed at fault-tolerant highly available systems, which strive to gracefully tolerate failures. Additionally, in the sticky variant of high availability they consider clients which always connect to the same replica, which as we have discussed in Section 5.2, may lead to clients being blocked permanently in case the replica crashes.

8.5 Highly available system designs

Finally, we discuss some influential highly available system designs. As discussed earlier, relaxed consistency models were studied in the context of database management systems before the rapid evolution of the Internet, but eventual consistency came to prominence only later. Due to the demand for scalable and highly available services many experimental systems were devised, including the serminal Bayou system [44], which we discuss in detail in Section 2.2. Commercial solutions followed suite. In this regard the design of Amazon Dynamo [20] has been particularly influential, as it has popularized techniques, such as the sole reliance on gossip protocols for (asynchronous) inter-replica communication, consistent hashing for dataset partitioning, the use of version vectors to enable handling of concurrent writes to the same data items and the use of hinted-handoff, sloppy quorum and anti-entropy algorithms to recover from failures. In effect, Amazon Dynamo, and the plethora of systems influenced by it (see, e.g., Apache Cassandra [21], Scylla [22], Riak [23], Volde-

mort [24], Netflix's Dynomite [25]) are massively scalable, can gracefully tolerate machine and network failures and still provide low latency responses. The latter trait can be attributed to the fact that in these systems typically communication with only a single service replica (that stores a copy of a dataset pertaining to the client's request) is sufficient to complete the request. It means that a replica is able to respond to the request without performing synchronous communication with other replicas. In the context of the *PACELC* framework [130], these systems choose low-latency over consistency even when no network splits occur.

Among all the systems influenced by Dynamo, Apache Cassandra has received the most attention in the research community, including work devoted to analyse its correctness properties [56, 125, 134], as well as attempts to strenghten them [135, 136].

Some of the systems mentioned above always synchronously write each update to disk before responding to the client, while other ones operate in-memory, with only asynchronous writes to stable storage. Since we consider both crash-stop and crash-recovery failure models, and stable storage plays a role only when recovery is possible, our analysis encompasses both kinds of systems.

9
Conclusions

In this dissertation we tackled the problem of correctness of highly available eventually consistent systems from the theoretical point of view. We explored the consistency and progress guarantees achievable under various assumptions. In particular, we investigated mixed-consistency systems, and examined the behaviour of highly available systems in the presence of failures. Doing so, not only we have gained new insights on the correctness of highly available systems, but we also proposed new useful abstractions such as *acute cloud types*.

The main goal of this dissertation, as stated in Section 1.2, was to formally identify and reason about tradeoffs and limitations in highly available systems' correctness guarantees resulting from mixed-consistency operations and from machine and network failures. We believe that the goal has been accomplished. To support this claim, we summarize below the main contributions of this dissertation.

First, in Chapter 2 we defined acute cloud types (ACTs), as an abstraction for mixed-consistency systems which combine the best features of eventually consistent and strongly consistent systems. 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 utilize non-blocking consensus-based synchronization prior to execution. We proposed and analysed *acute non-negative counter* (ANNC), an example ACT. We also analyzed the seminal Bayou system and showed how it can be improved to become a general purpose ACT called AcuteBayou. Doing so we identified undesirable phenomena which can occur in Bayou. In particular, we found that *temporary operation reordering* is unavoidable in systems similar to Bayou which use two incompatbile mechanisms to order weak and strong operations.

Next, in Chapter 3 we provided a formal framework which enables reasoning about ACTs and other mixed-consistency systems as well. Within the framework we have formalized several correctness criteria, including *fluctuating eventual consistency* (FEC). FEC adequately captures the guarantees offered by sys-

tems, which similarly to AcuteBayou, exhibit temporary operation reordering. Using the framework we also formally proved the correctness guarantees of the ANNC and AcuteBayou.

Then, in Chapter 4 we generalized our findings about AcuteBayou and proposed a formal result to show the limitations of mixed-consistency systems. We proved that mixed-consistency systems which combine the best features of eventually consistent and strongly consistent systems, as ACTs do, and which feature arbitrarily complex semantics, cannot avoid temporary operation reordering. Thus such systems do not satisfy basic eventual consistency for weak operations. We explored the possible tradeoffs in terms of correctness guarantees and properties by analyzing other solutions which circumvent the impossibility result, by compromising fault-tolerance or sacrificing high availability.

After that, in Chapter 5 we showed how to adequately model real world highly available systems utilizing client-server architecture, in order to reason about their correctness in spite of failures. We outlined the possible failure scenarios and classified six failure models. We also provided a formal framework which explicitly considers failures, and thus enables formalizing *failure-aware correctness criteria*.

Next, in Chapter 6 we expressed session guarantees in our framework and discussed their relevance and properties. We showed that classic session guarantees can be counter-productive for certain data types and provided a novel substitute instead, called *context preservation*.

Finally, in Chapter 7 we analyzed the correctness guarantees which are possible to achieve under different failure models. In particular we showed that basic eventual consistency is not achievable if permanent network partitions or unrecoverable replica crashes occur. We defined eventual variants of two key sesion guarantees and a family of failure-aware correctness criteria which precisely capture the correctness guarantees achievable in the considered failure models. We also identified several undesirable artifacts observable by the clients when failures occur. We showed how some of them can be remedied.

In the future, we plan to devise novel acute cloud types that can be applied in practical settings, such as within a NoSQL data store. Moreover, we would like to investigate which groups of operations can be implemented in ACTs without temporary operation reordering occuring.

Bibliography

[1] M. Kokociński, T. Kobus, and P. T. Wojciechowski, "On mixing eventual and strong consistency: Acute cloud types," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, 2022.

[2] M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Brief announcement: Eventually consistent linearizability," in *Proceedings of PODC '15: the 34th ACM Symposium on Principles of Distributed Computing*, July 2015.

[3] M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Brief announcement: On mixing eventual and strong consistency: Bayou revisited," in *Proc. of PODC '19*, July 2019.

[4] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Introduction to transactional replication," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications* (R. Guerraoui and P. Romano, eds.), vol. 8913 of *Lecture Notes in Computer Science*, Springer, 2015.

[5] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "State-machine and deferred-update replication: Analysis and comparison," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, 2017.

[6] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Relaxing real-time order in opacity and linearizability," *Elsevier Journal on Parallel and Distributed Computing*, vol. 100, 2017.

[7] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Hybrid transactional replication: State-machine and deferred-update replication combined," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, July 2018.

[8] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "Model-driven comparison of state-machine-based and deferred-update replication schemes," in *Proceedings of SRDS '12: the 31st IEEE International Symposium on Reliable Distributed Systems*, Oct. 2012.

[9] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *Proceedings of ICDCS '13: the 33rd IEEE International Conference on Distributed Computing Systems*, July 2013.

[10] M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Make the leader work: Executive deferred update replication," in *Proceedings of SRDS '14: the 33rd IEEE International Symposium on Reliable Distributed Systems*, Oct. 2014.

[11] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "The correctness criterion for deferred update replication," in *Program of TRANSACT '15: the 10th ACM SIGPLAN Workshop on Transactional Computing*, June 2015.

[12] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Jiffy: A lock-free skip list with batch updates and snapshots," in *Proceedings of PPoPP '22: the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Apr. 2022.

[13] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "A fault-tolerant data processing computer system and method for implementing a distributed two-tier state machine," 2017. EPO patent no. EP 3193256 B1, July 7, 2017.

[14] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "A fault-tolerant data processing computer system and method for implementing a distributed two-tier state machine," 2018. USPTO patent no. US 10135929 B2, Nov. 20, 2018.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, July 1978.

[16] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, Dec. 1990.

[17] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, *The primary-backup approach*. ACM Press/Addison-Wesley Publishing Co., 1993.

[18] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proc. of PODC '00: the 19th Annual ACM Symposium on Principles of Distributed Computing*, July 2000.

[19] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, Jan. 2009.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Operating Systems Review*, vol. 41, Oct. 2007.

[21] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Operating Systems Review*, vol. 44, Apr. 2010.

[22] ScyllaDB documentation, "Scylla." https://www.scylladb.com/.

[23] Basho documentation, "Riak key-value store." http://basho.com/products/riak-overview/.

[24] Project Voldemort, "Voldemort key-value store." https://www.project-voldemort.com/voldemort/.

[25] Netflix, "Netflix dynomite distributed dynamo layer." https://github.com/Netflix/dynomite.

[26] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proc. of SSS '11*, May 2011.

[27] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," Tech. Rep. 7506, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[28] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, "Replicated data types: Specification, verification, optimality," in *Proc. of POPL '14: the 41st Symposium on Principles of Programming Languages (POPL)*, Jan. 2014.

[29] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski, "Specification and complexity of collaborative text editing," in *Proc. of PODC '16*, 2016.

[30] N. M. Preguiça, C. Baquero, and M. Shapiro, "Conflict-free replicated data types," *Computing Research Repository*, vol. abs/1805.06358, 2018.

[31] Amazon DynamoDB documentation, "Consistent reads in DynamoDB." https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html, 2019.

[32] Microsoft Azure documentation, "Consistency levels in Cosmos DB." https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels, 2019.

[33] Apache Cassandra documentation, "Light weight transactions in Cassandra." https://docs.datastax.com/en/cql/3.3/cql/cql_using/useInsertLWT.html, 2019.

[34] Basho documentation, "Consistency levels in Riak." https://docs.basho.com/riak/kv/2.2.3/developing/app-guide/strong-consistency, 2019.

[35] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proceedings of ODSI '12: the 10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012.

[36] C. Li, N. Preguiça, and R. Rodrigues, "Fine-grained consistency for geo-replicated systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, July 2018.

[37] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood, "Cloud types for eventual consistency," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, June 2012.

[38] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich, "Global sequence protocol: A robust abstraction for replicated shared state.," in *Proc. of ECOOP '15*, July 2015.

[39] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. of VLDB Endowment*, vol. 1, no. 2, 2008.

[40] 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. of SOSP '13*, Nov. 2013.

[41] M. Milano and A. C. Myers, "Mixt: a language for mixing consistency in geodistributed transactions," *ACM SIGPLAN Notices*, vol. 53, no. 4, 2018.

[42] V. Gavrielatos, A. Katsarakis, V. Nagarajan, B. Grot, and A. Joshi, "Kite: Efficient and available release consistency for the datacenter," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2020.

[43] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Queue*, vol. 11, Mar. 2013.

[44] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *Proceeding of SOSP '95: the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.

[45] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, and N. Suri, "Eventually linearizable shared objects," in *Proc. of PODC '10*, July 2010.

[46] R. Guerraoui and E. Ruppert, "A paradox of eventual linearizability in shared memory," in *Proc. of PODC '14*, July 2014.

[47] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv, "Eventually consistent transactions," in *Proceedings of the 21st European Conference on Programming Languages and Systems*, Mar. 2012.

[48] S. Burckhardt, A. Gotsman, and H. Yang, "Understanding eventual consistency," Tech. Rep. MSR-TR-2013-39, Microsoft Research, Mar. 2013.

[49] S. Burckhardt, "Principles of eventual consistency," *Foundations and Trends in Programming Languages*, vol. 1, Oct. 2014.

[50] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, Sept. 1979.

[51] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM*, vol. 26, no. 4, 1979.

[52] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, 1990.

[53] Apache Cassandra Issues (Jira), "Mixing LWT and non-LWT operations can result in an LWT operation being acknowledged but not applied." `https://jira.apache.org/jira/browse/CASSANDRA-11000`, 2016.

[54] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, Nov. 1992.

[55] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, 1998.

[56] S. Liu, M. Rahman, S. Skeirik, I. Gupta, and J. Meseguer, "Formal modeling and analysis of cassandra in maude," in *Formal Methods and Software Engineering* (S. Merz and J. Pang, eds.), vol. 8829 of *LNCS*, Springer International Publishing, 2014.

[57] A. Bouajjani, C. Enea, and J. Hamza, "Verifying eventual consistency of optimistic replication systems," in *Proc. of POPL '14*, Jan. 2014.

[58] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," in *Proc. of PODC '15: the 34th ACM Symposium on Principles of Distributed Computing*, July 2015.

[59] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro, "'Cause I'm strong enough: Reasoning about consistency choices in distributed systems," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2016.

[60] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, Jan. 2017.

[61] A. Girault, G. Gößler, R. Guerraoui, J. Hamza, and D. Seredinschi, "Monotonic prefix consistency in distributed systems," in *Proc. of FORTE '18*, June 2018.

[62] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *Proc. of SoCC '12*, Oct. 2012.

[63] F. Houshmand and M. Lesani, "Hamsaz: Replication coordination analysis and synthesis," in *Proc. of POPL '19*, vol. 3, Jan. 2019.

[64] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems*, vol. 4, June 1979.

[65] M. Shapiro, M. S. Ardekani, and G. Petri, "Consistency in 3d," in *Proceedings of CONCUR '16: The 27th International Conference on Concurrency Theory*, vol. 59, Aug. 2016.

[66] 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. of PDIS '94: the 3rd International Conference on Parallel and Distributed Information Systems*, 1994.

[67] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, Mar. 1977.

[68] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, Sept. 1987.

[69] R. Guerraoui and E. Ruppert, "Linearizability is not always a safety property," in *Proceedings of NETYS '14: the 4th International Conference on Networked Systems*, May 2014.

[70] A. Rudoff, "Persistent memory programming," *;login:*, vol. 42, no. 2, 2017.

[71] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

[72] F. Pedone, R. Guerraoui, and A. Schiper, "Exploiting atomic broadcast in replicated databases," in *Proceedings of Euro-Par '98: the 4th International Conference on Parallel Processing*, Sept. 1998.

[73] B. Charron-Bost, F. Pedone, and A. Schiper, eds., *Replication - Theory and Practice*, vol. 5959 of *Lecture Notes in Computer Science*. 2010.

[74] R. Hansdah and L. Patnaik, "Update serializability in locking," in *Proceedings of ICDT '86: the 1st International Conference on Database Theory*, Sept. 1986.

[75] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication.," in *Proc. of ICDCS '12*, June 2012.

[76] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman, "Eventually-serializable data services," in *Proc. of PODC '96*, May 1996.

[77] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, Jan. 1991.

[78] P. A., Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[79] G. V. Chockler and A. Gotsman, "Multi-shot distributed transaction commit," in *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, Oct. 2018.

[80] N. M. Preguiça, C. Baquero, P. S. Almeida, V. Fonte, and R. Gonçalves, "Dotted version vectors: Logical clocks for optimistic replication," *CoRR*, vol. abs/1011.5808, 2010.

[81] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, Apr. 1985.

[82] P. Verissimo and C. Almeida, "Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models.," *IEEE TCOS Bulletin*, vol. 7, Dec. 1995.

[83] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Journal of the ACM*, vol. 43, July 1996.

[84] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proc. of SIGMOD '13*, June 2013.

[85] N. Preguiça, M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. Shapiro, "SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine," in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, Oct. 2014.

[86] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Oct. 2011.

[87] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Closing the performance gap between causal consistency and eventual consistency," in *Proc. of PaPEC '14*, Apr. 2014.

[88] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Computing Surveys*, vol. 49, June 2016.

[89] T. L. Greenough, *Representation and Enumeration of Interval Orders*. Ph.D., Dartmouth College, 1976.

[90] D. Skeen, "Nonblocking commit protocols," in *Proc. of SIGMOD '81*, 1981.

[91] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, "Putting consistency back into eventual consistency," in *Proc. of EuroSys '15*, Apr. 2015.

[92] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis, "Zeno: Eventually consistent byzantine-fault tolerance," in *Proceedings of NSDI '09: the 6th USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2009.

[93] A.-M. Bosneag and M. Brockmeyer, "A formal model for eventual consistency semantics.," in *In Proceedings of International Conference on Parallel and Distributed Computing Systems, PDCS 2002*, Jan. 2002.

[94] A. Gotsman and S. Burckhardt, "Consistency models with global operation sequencing and their composition," in *Proc. of DISC '17*, Oct. 2017.

[95] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of SOSP '11: the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[96] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proc. of NSDI '13*, Apr. 2013.

[97] 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 *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, Mar. 2017.

[98] ISO/IEC 9075-1:2016, "SQL standard," 2016.

[99] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *SIGMOD Rec.*, vol. 24, May 1995.

[100] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, "Transaction chains: achieving serializability with low latency in geo-distributed storage systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, Nov. 2013.

[101] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "Salt: Combining ACID and BASE in a distributed database," in *Proc. of OSDI '14*, Oct. 2014.

[102] X. Zhao and P. Haller, "Replicated data types that unify eventual consistency and observable atomic consistency," *Journal of Logical and Algebraic Methods in Programming*, vol. 114, 2020.

[103] P. E. O'Neil, "The escrow transactional method," *ACM Transactions on Database Systems*, vol. 11, Dec. 1986.

[104] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça, "Extending eventually consistent cloud databases for enforcing numeric invariants," in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, Sept. 2015.

[105] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi, "Incremental consistency guarantees for replicated objects," in *Proc. of OSDI '16*, Nov. 2016.

[106] Apache ZooKeeper documentation, "Zookeeper." `https://zookeeper.apache.org`, 2019.

[107] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

[108] P. A. Bernstein and S. Das, "Rethinking eventual consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, June 2013.

[109] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. of VLDB Endowment*, vol. 7, Nov. 2013.

[110] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, June 2002.

[111] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997.

[112] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *Proceedings of the ACM SIGCOMM 2011 conference*, 2011.

[113] D. Turner, K. Levchenko, J. C. Mogul, S. Savage, and A. C. Snoeren, "On failure in managed enterprise networks," *HP Labs HPL-2012-101*, 2012.

[114] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu, "A large scale study of data center network reliability," in *Proc. of IMC '18*, 2018.

[115] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, Oct. 2018.

[116] M. Alfatafta, "An analysis of partial network partitioning failures in modern distributed systems," Master's thesis, University of Waterloo, 2020.

[117] J. Bonér, "Scalability, availability & stability patterns." `https://www.slideshare.net/jboner/scalability-availability-stability-patterns/`, 2010.

[118] P. S. Almeida, C. Baquero, R. Gonçalves, N. Preguiça, and V. Fonte, "Scalable and accurate causality tracking for eventually consistent stores," in *Proc. of DAIS '14*, June 2014.

[119] Google documentation, "Google cloud – storage options." `https://cloud.google.com/compute/docs/disks/`.

[120] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable NVM," in *Proc. of MEMSYS '19*, Sept. 2019.

[121] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-based scalable network services," in *Proc. of SOSP '97*, 1997.

[122] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, (New York, NY, USA), ACM, 1987.

[123] S. Dubois, R. Guerraoui, P. Kuznetsov, F. Petit, and P. Sens, "The weakest failure detector for eventual consistency," in *Proc. of PODC '15*, 2015.

[124] W. Golab, X. Li, and M. A. Shah, "Analyzing consistency properties for fun and profit," in *Proc. of PODC '11*, 2011.

[125] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and I. Gupta, "Client-centric benchmarking of eventual consistency for cloud storage systems," in *Proc. of PODC '14*, 2014.

[126] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, Mar. 1995.

[127] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 3, 1991.

[128] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, and convergence," Technical Report TR-11-22, University of Texas at Austin, USA, May 2011.

[129] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak, "From session causality to causal consistency," in *Proceeding of PDP '04: the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Feb 2004.

[130] D. Abadi, "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story," *Computer*, vol. 45, Feb. 2012.

[131] E. Brewer, "Cap twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, 2012.

[132] S. Gilbert and N. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, 2012.

[133] P. Mahajan, *Highly available storage with minimal trust*. Ph.D., University of Texas at Austin, USA, May 2012.

[134] H. Fan, A. Ramaraju, M. McKenzie, W. Golab, and B. Wong, "Understanding the causes of consistency anomalies in apache cassandra," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, 2015.

[135] P. Garefalakis, P. Papadopoulos, I. Manousakis, and K. Magoutis, "Strengthening consistency in the cassandra distributed key-value store," in *IFIP International Conference on Distributed Applications and Interoperable Systems*, 2013.

[136] G.-S. Gao, K. Konwar, J. Mantica, H. Pan, D. Russell Kish, L. Tseng, Z. Wang, and Y. Wu, "Practical experience report: Cassandra+: Trading-off consistency, latency, and fault-tolerance in cassandra," in *International Conference on Distributed Computing and Networking 2021*, 2021.

A
StateObject properties

Although in Algorithm 3 we present a referential implementation of StateObject, in general we treat the $state$ object as a black box with unknown implementation. The correctness of AcuteBayou depends on the properties of the $state$ object which we formalize below.

Take the list of requests that were executed on the $state$, and remove the requests which were rolled back; we call the resulting sequence α the current trace of the $state$.[1] Since the $state$ encapsulates the state of the system after locally executing and revoking requests, we require that the $state$'s responses are consistent with a deterministic serial execution of α as specified by the type specification $\mathcal{F}$ when taking into account the relative visibility between requests encoded in the ctx field of the Req record. In case of any strong operation op (in a request r), we assume that all requests $r' \in \alpha$ prior to r are visible to r (regardless of ctx). This is because op is executed only once r is on the $committed$ list and thus its position relative to all other operations is fixed and corresponds to the TOB order.

More precisely, for any given trace α, the $state$ object deterministically holds the state S_α, and for any operation $op \in ops(\mathcal{F})$, the response of the $state.\mathrm{execute}$ function invoked on the $state$ object in state S_α equals $\mathcal{F}(op, C_\alpha)$, where $C_\alpha = (E_\alpha, op_\alpha, vis_\alpha, ar_\alpha)$ is a context such that:

- E_α consists of all the requests in α,
- $op_\alpha(r) = r.op$, for any request $r \in E_\alpha$,
- vis_α is the visibility relation based on the ctx fields of the Req record for the weak operations and on the order in α for strong operations, i.e. for any $r, r' \in E_\alpha$ such that $r \xrightarrow{vis_\alpha} r'$:
  - if $r'.strongOp = false$, then $r.id \in r'.ctx$;
  - if $r'.strongOp = true$, then $r \xrightarrow{ar_\alpha} r'$;

[1] We omit weak RO operations executed in Algorithm 4 line 4, which are not associated with any Req record.

- ar_α is the enumeration of requests in E_α according to their position in α.

In AcuteBayou, $\alpha = executed \cdot \mathrm{reverse}(toBeRolledBack)$, because:

- requests are executed only if $toBeRolledBack$ is empty,[2]
- whenever a request is executed it is added to the $executed$ list, thus it is appended to the end of α,
- in the $\mathrm{adjustExecution}$ function, some requests move from the $executed$ list to the end of the $toBeRolledBack$ list, thus not changing their position in α,
- whenever a request is rolled back, it is removed from the head of the $toBeRolledBack$ list, and thus removed from the end of α, consistently with the definition of a trace.

[2]Weak requests are also executed in the invoke block, independently of the $toBeExecuted$ and $toBeRolledBack$ lists, but they are immediately afterwards rolled back, so they do not influence the trace.

B

Correctness proofs for ANNC and AcuteBayou

In this section we provide the formal proofs of correctness for ANNC and AcuteBayou anticipated in Section 3.8. We start with an overview of proofs' structures.

In order to prove correctness of either protocol, we take a single arbitrary execution of the protocol, and without making any specific assumptions about it, we show how the visibility and arbitration relations can be defined so that the appropriate correctness guarantees can be proven. Below we briefly outline our approach.

In both ANNC and AcuteBayou, strong operations are disseminated solely by TOB, and weak updating operations are sent using both RB and TOB. On the other hand weak RO operations are executed completely locally and do not involve any network communication (strong RO operations are present only in AcuteBayou and are treated as regular strong operations). Thus, in the proofs, for the purpose of constructing the arbitration relation (ar), we order all updating (strong or weak) operations based on the order of the delivery of their respective messages broadcast using TOB. In the case of updating operations whose messages were not $\mathrm{TOB\text{-}delivered}$ (which can happen in the asynchronous runs), we order them in ar after all the operations whose messages were $\mathrm{TOB\text{-}delivered}$. Their relative order can be arbitrary in ANNC, and in AcuteBayou it has to conform to the order imposed by the Req records. Finally, for completeness, ar needs to include also weak RO operations. We carefully interleave them with updating operations in such a way to guarantee no circular causality as well as equivalence between visibility and arbitration for strong operations.

We construct the visibility relation (vis) by choosing for any two events e, e' whether one should be observed by the other. We include an edge $e \xrightarrow{vis} e'$ under two, broad conditions: the edge is *essential*, i.e., e could have influenced the return value of e', or the edge is *non-essential*, i.e., e could not have influenced the return value of e' (because, e.g., e is an RO operation), but e occurs before e' in real-time or arbitration. Non-essential edges are important to guarantee eventual visibility for all events.

Now let us make some observations regarding network properties during

synchronous and asynchronous runs. Since we consider infinite fair executions, in both types of runs each message RB-cast is guaranteed to be RB-delivered by each replica. On the other hand, the same delivery guarantee, but for messages TOB-cast, holds only in the stable runs, and in the asynchronous runs, some messages can be TOB-delivered while others may remain pending. However, asynchronous runs still obey other guarantees, which means that, crucially, no messages TOB-cast will be TOB-delivered by any replica out of order. Moreover, if some message was TOB-delivered by one replica, then it will be TOB-delivered by all replicas. Also, if one replica manages to TOB-cast infinitely many messages which are then TOB-delivered, then each replica can successfully TOB-cast and TOB-deliver its messages. Thus, in the asynchronous runs, we expect a finite number of TOB-cast messages to be TOB-delivered, while all other to remain pending.

For each event e let us denote by $msg_{TOB}(e)$ and $msg_{RB}(e)$, respectively, the message TOB-cast in the event e and the message RB-cast in the event e (both $msg_{TOB}(e)$ and $msg_{RB}(e)$ can be undefined for a given event e, denoted $msg_{TOB}(e) = \bot$ or $msg_{RB}(e) = \bot$). For any two events e, e', such that $msg_{TOB}(e) = m$, $msg_{TOB}(e') = m'$ and $tobNo(m) < tobNo(m')$ we introduce the following notation: $e \xrightarrow{tobNo} e'$, which defines the *tobNo order* (based on the $tobNo$ function). Additionally, for any two events e, e', such that $msg_{TOB}(e) = m$ (or respectively $msg_{RB}(e) = m$), we write $e \xrightarrow{TOBdel} e'$ ($e \xrightarrow{RBdel} e'$), if e' executes on a replica that has TOB-delivered (RB-delivered) m prior to its execution.

Finally, let us observe that we model replicas as deterministic state machines (as discussed in Section 2.4.1), whose specification we give through pseudocode. The variables declared in the algorithms of ANNC and AcuteBayou represent the state of the replicas, while the code blocks represent atomic steps that transition the replicas from one state to another. It means that each such block executes completely before any of its effects become visible. This allows us to infer the following rule (in both ANNC and AcuteBayou) for weak operations which execute in one atomic transition in some event e, which is either in the $TOBdel$ or $RBdel$ relation with any other event e': $lvl(e) = weak \wedge (e \xrightarrow{TOBdel} e' \vee e \xrightarrow{RBdel} e') \Rightarrow e \xrightarrow{rb} e'$ (e *returns before* e').

B.1 ANNC correctness proofs

Let us proceed with the proof of the guarantees offered by ANNC in the stable runs.

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

Proof. For any given arbitrary stable run of ANNC represented by a history $H = (E, op, rval, rb, ss, lvl)$ we have to find suitable vis, ar and par, such that $A = (H, vis, ar, par)$ is such that $A \models \text{BEC}(weak, \mathcal{F}_{NNC}) \wedge \text{LIN}(strong, \mathcal{F}_{NNC})$.

Additional observations. Note that each $subtract$ operation executed in some event e finishes when the replica TOB-delivers the message $m = msg_{TOB}(e)$. It means that for every operation executed in event e', such that $e \xrightarrow{rb} e'$, if $msg_{TOB}(e') = m'$ ($m' \neq \perp$), then $tobNo(m) < tobNo(m')$.

Arbitration. We construct the total order relation ar by sorting all updating events (additions and subtractions) based on the order in which their respective TOB-cast messages are TOB-delivered, i.e., respecting the $tobNo$ order.

Next, we interleave the updating events with RO events (gets) in the following way: each such an RO event e occurs in ar after the last subtract event e' such that $e \xrightarrow{rb} e'$. Thus, for each subtract event e' the following holds $e \xrightarrow{ar} e' \Rightarrow e \xrightarrow{rb} e'$. The relative order of RO operations is irrelevant.

As ANNC does not feature operation reordering, for each event e we simply let $par(e) = ar$.

Visibility. For any two events $e, e' \in E$, we include an edge $e \xrightarrow{vis} e'$ in our construction of vis, if:

1. $op(e) = add(v)$ or $op(e) = subtract(v)$, $op(e') = subtract(v')$ and $e \xrightarrow{tobNo} e'$,
2. $op(e) = subtract(v)$, $op(e') = get$ and $e \xrightarrow{TOBdel} e'$,
3. $op(e) = add(v)$, $op(e') = get$, and $e \xrightarrow{TOBdel} e'$,
4. $op(e) = add(v)$, $op(e') = get$, and $e \xrightarrow{RBdel} e'$,
5. $op(e) = get$, $op(e') = get$ and $e \xrightarrow{rb} e'$,
6. $op(e) = get$, $op(e') = subtract(v')$ and $e \xrightarrow{ar} e'$,
7. $op(e') = add(v')$ and $e \xrightarrow{rb} e'$,

(for some $v, v' \in \mathbb{N}$).

The edges 1-4 are essential, while the edges 5-7 are non-essential. The updates that are visible to a $subtract$ operation depends solely on the $tobNo$ order, while in case of a get operation, the $TOBdel$ and $RBdel$ relations play a role. It does not matter which updates are visible to an add operation because it always responds with a simple ok acknowledgment, hence the edge 7 is non-essential.

Note that in case of edges 3-4, $e \xrightarrow{rb} e'$ is implied (see the general observations in Section B), and in case of the edge 6, $e \xrightarrow{rb} e'$ follows directly from the construction of ar. Thus, for all edges 3-7, $e \xrightarrow{rb} e'$.

Having defined A (through vis, ar and par), it now remains to show that $A \models \text{BEC}(weak, \mathcal{F}_{NNC}) \wedge \text{LIN}(strong, \mathcal{F}_{NNC})$, or more specifically $A \models \text{EV}(weak) \wedge \text{EV}(strong) \wedge \text{NCC}(weak) \wedge \text{NCC}(strong) \wedge \text{RVAL}(weak) \wedge \text{RVAL}(strong) \wedge \text{SINORD}(strong) \wedge \text{RT}(strong)$.

Eventual visibility. We prove now that eventual visibility is satisfied for all events:

- each add or $subtract$ event e is visible to all subsequent $subtract$ events from some point, because there is only a finite number of updating events e' such that $e \not\xrightarrow{tobNo} e'$ (1),

- each add or $subtract$ event e is visible to all subsequent get events from some point, because both $msg_{RB}(e)$ and $msg_{TOB}(e)$ are eventually delivered on all replicas, (2, 3 and 4),
- each get event e is visible to all subsequent get events from some point (5),
- each get event e is visible to all subsequent $subtract$ events from some point, because by construction of ar there is only a finite number of events e' such that $e \xrightarrow{ar} e'$ (6),
- each event is visible to all subsequent add events from some point (7).

No circular causality. We need to show that $acyclic(hb \cap (W \times W))$ and $acyclic(hb \cap (S \times S))$, where $W \subseteq E, S \subseteq E$, are, respectively, the sets of all weak, and strong events. We elect to prove a more general case of $acyclic(hb)$.

Recall that $hb = (vis \cup so)^+$. If $acyclic(vis \cup so)$, then $acyclic(hb)$, because transitive edges cannot introduce cycles. Thus, we have eight types of edges to consider: edges 1-7 from vis and the eight edge $e \xrightarrow{so} e'$. We divide them into two groups: the first one consists of edges 1-2, while the second one consists of edges 3-8. Note that for the second group $e \xrightarrow{rb} e'$ always holds.

There can be no cycles when we restrict the edges only to the ones from the first group, as edge 1 is constrained by the $tobNo$ order, and edge 2 leads to a get event which cannot be followed using only edges from the first group.

Also, there can be no cycles when we restrict the edges only to the ones from the second group, as all the edges are constrained by the rb relation, which is naturally acyclic.

Thus, a potential cycle could only form when we mix edges from both groups. Let us assume that the cycle contains the following chain of edges: $a \xrightarrow{hb} b \xrightarrow{hb} ... \xrightarrow{hb} c \xrightarrow{hb} ... \xrightarrow{hb} d$, where $a, b, c, d \in E$, all the edges between b and c belong to the second group, while the other ones belong to the first group. Notice that $b \xrightarrow{rb} c$, and that $op(a), op(c) \in \{add(v) : v \in \mathbb{N}\} \cup \{subtract(v) : v \in \mathbb{N}\}$ while $op(b), op(d) \in \{subtract(v) : v \in \mathbb{N}\} \cup \{get\}$. Thus, the chain consists of a series of edges from the first group and a series of edges from the second group. The whole cycle can be combined from multiple such chains, but for simplicity, let us assume that it contains only one such chain and that $d = a$ (the same reasoning as below can be applied iteratively for multiple interleavings of edges from the two groups).

If $op(b) = subtract(v)$, for some $v \in \mathbb{N}$, then $a \xrightarrow{tobNo} b$ (edge 1), and since $b \xrightarrow{rb} c$, also $b \xrightarrow{tobNo} c$ (see the additional observations in the beginning of the proof). A contradiction: $a \xrightarrow{tobNo} b \xrightarrow{tobNo} c \xrightarrow{tobNo} a$.

If $op(b) = get$, then $op(a) = subtract(v)$, for some $v \in \mathbb{N}$, and $a \xrightarrow{TOBdel} b$ (edge 2). Either $a \xrightarrow{tobNo} c$, or $c \xrightarrow{tobNo} a$. In the former case we end up with a similar contradiction as above: $a \xrightarrow{tobNo} c \xrightarrow{tobNo} a$. In the latter case, since $c \xrightarrow{tobNo} a$, also $c \xrightarrow{TOBdel} b$ (the message $msg_{TOB}(c)$ is TOB-delivered before the message $msg_{TOB}(a)$). However, $b \xrightarrow{rb} c$, which means that the message $msg_{TOB}(c)$ was not even TOB-cast yet when b executed. A contradiction.

Return value consistency. We need to show that for each event $e \in E$: $rval(e) = \mathcal{F}_{NNC}(op(e), context(A, e))$. We base our reasoning below on essential vis edges and ar order.

Trivially, the condition is satisfied for all add events, which always return ok. For all $subtract$ and get events, we can exclude from $context(A, e)$ all get events which by the definition of an RO operation are irrelevant for the computation of $\mathcal{F}_{NNC}$.

In case of a $subtract(v)$ operation, for some $v \in \mathbb{N}$, executed in some event e, $context(A, e)$ includes all the add and $subtract$ events that precede e in the $tobNo$ order. When applying the $foldl$ function from the definition of $\mathcal{F}_{NNC}$, these add and $subtract$ operations are processed one by one, in the order of their TOB-delivery (by construction of ar). Each $add(v)$ operation increases the accumulator by v, and each $subtract(v)$ operation decreases the accumulator by v, but only if it is greater or equal v. This matches the pseudocode (lines 24 and 27-28) with the accumulator corresponding to the difference between $strongAdd$ and $strongSub$ variables. Thus, the computed value of the $foldl$ function corresponds to the difference between $strongAdd$ and $strongSub$ variables at the time the response to e is computed in line 26. If that value is greater or equal v then $true$ is returned, which matches the pseudocode's behaviour.

In case of a get operation executed in some event e, $context(A, e)$ includes all the add and $subtract$ events that were TOB-delivered before the execution of e, as well as, (possibly) some add events which were not TOB-delivered, but only RB-delivered before the execution of e. Note that all the latter add events are ordered according to ar, after all the former add and $subtract$ events (have they had been ordered earlier due to lower $tobNo$ value of their respective TOB-cast message, they would also be TOB-delivered). When processing the $foldl$ function up to the last TOB-delivered event, the value of the accumulator corresponds, similarly as in case of $subtract$ events above, to the difference between $strongAdd$ and $strongSub$ variables. Then, when processing the remaining add events the final computed value of the $foldl$ function grows by an amount V, which is equal to the sum of all these add operations' arguments. Due to the fact that each TOB-delivered message is first RB-delivered or is processed as if it were RB-delivered (lines 22-23), the value of $weakAdd$ is always greater or equal $strongAdd$. The difference between $weakAdd$ and $strongAdd$ variables corresponds exactly to V, because it includes events which were RB-delivered, but not TOB-delivered. Thus, the computed value of $\mathcal{F}_{NNC}(get, context(A, e))$ equals $strongAdd - strongSub + V = strongAdd - strongSub + weakAdd - strongAdd = weakAdd - strongSub$ at the time of executing e, which matches $rval(e)$.

Single order. Since there are no pending $subtract$ operations (because eventually every message is TOB-delivered and the operations finish), we have to simply prove that $vis \cap (E \times S) = ar \cap (E \times S)$, where $S = \{e : lvl(e) = strong\}$. In other words, for any two events $e \in E, e' \in S$: $e \xrightarrow{vis} e' \Leftrightarrow e \xrightarrow{ar} e'$.

Let us begin with $e \xrightarrow{vis} e' \Rightarrow e \xrightarrow{ar} e'$. Either $e \xrightarrow{tobNo} e'$ (edge 1), or $op(e) = get$ (edge 6). In both cases $e \xrightarrow{ar} e'$.

Now let us consider $e \xrightarrow{ar} e' \Rightarrow e \xrightarrow{vis} e'$. Either $op(e) \in \{add(v) : v \in$

$\mathbb{N}\} \cup \{subtract(v) : v \in \mathbb{N}\}$, or $op(e) = get$. In the former case, $e \xrightarrow{tobNo} e'$, and thus $e \xrightarrow{vis} e'$ (edge 1). In the latter case, $e \xrightarrow{rb} e'$ (by construction of ar), and thus $e \xrightarrow{vis} e'$ (edge 6).

Real-time order. We need to show that arbitration order respects the real-time order of strong operations, i.e., $rb \cap (S \times S) \subseteq ar$. In other words, for any two $e, e' \in S$: $e \xrightarrow{rb} e' \Rightarrow e \xrightarrow{ar} e'$.

Clearly, if $e \xrightarrow{rb} e'$, then $e \xrightarrow{tobNo} e'$ (see the additional observations in the beginning of the proof). Thus, $e \xrightarrow{ar} e'$ (by construction of ar). ■

Now, let us continue with the proof of the guarantees offered by ANNC in the asynchronous runs.

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

Proof. To show the inability of ANNC to satisfy LIN$(strong, \mathcal{F}_{NNC})$ in asynchronous runs, it is sufficient to observe that due to some of the TOB-cast messages not being TOB-delivered, some of the $subtract$ operations remain pending. A pending operation's return value equals ∇ which is irreconcilable with the requirements of the predicate RVAL$(\mathcal{F}_{NNC})$.

The proof regarding the guarantees of the weak operations is similar to the one for the stable runs, thus we rely on it and focus only on differences between stable and asynchronous runs that need to be addressed. Now for any given arbitrary asynchronous run of ANNC represented by a history $H = (E, op, rval, rb, ss, lvl)$ we have to find suitable vis, ar and par, such that $A = (H, vis, ar, par)$ is such that $A \models$ BEC$(weak, \mathcal{F}_{NNC})$.

Arbitration. We construct the total order relation ar by sorting all updating events (additions and subtractions) based on the order in which their respective TOB-cast messages are TOB-delivered, i.e., respecting the $tobNo$ order. Updating events whose messages are not TOB-delivered are ordered after those whose messages are TOB-delivered.

Next, we interleave the updating events with RO events (gets) in the following way: each such an RO event e occurs in ar after the last *non-pending* subtract event e' such that $e \not\xrightarrow{rb} e'$. Thus, for each non-pending subtract event e' the following holds $e \xrightarrow{ar} e' \Rightarrow e \xrightarrow{rb} e'$. The relative order of RO operations is irrelevant.

As ANNC does not feature operation reordering, for each event e we simply let $par(e) = ar$.

Visibility. We construct the visibility relation in the same way as in the stable runs case. However, we remove edges to and from pending $subtract$ events. Since pending operations do not provide a return value, no edge to a pending event is essential. Also, as we guarantee only eventual visibility for weak events, edges to $subtract$ events are not necessary to satisfy EV$(weak)$. Moreover, edges *from* pending events are not needed either, because by definition a pending event is never followed in rb by any other event (which is a requirement to fail the test for EV). Again, for all edges 3-7, $e \xrightarrow{rb} e'$.

Having defined A (through vis, ar and par), it now remains to show that $A \models \text{BEC}(weak, \mathcal{F}_{NNC})$, or more specifically $A \models \text{EV}(weak) \wedge \text{NCC}(weak) \wedge \text{RVAL}(weak)$.

Eventual visibility. We prove now that eventual visibility is satisfied for all *weak* events:

- each add or *non-pending* $subtract$ event e is visible to all subsequent get events from some point, because $msg_{RB}(e)$ or $msg_{TOB}(e)$ are eventually delivered on all replicas (2, 3 and 4),
- each get event e is visible to all subsequent get events from some point (5),
- each *non-pending* event is visible to all subsequent add events from some point (7).

No circular causality. We use exactly the same reasoning as in the stable runs case to show that $acyclic(hb)$ holds true.

Return value consistency. Again, we use exactly the same reasoning as in the stable runs case to show that for each *weak* event $e \in E$: $rval(e) = \mathcal{F}_{NNC}(op(e), context(A, e))$. Although this time we only need to prove return value consistency for add and get operations, it can be shown that it also holds for *non-pending* subtract events.

■

B.2 AcuteBayou correctness proofs

The proofs for AcuteBayou are analogous to those for ANNC, but are slightly more complex due to operation reordering and the more general nature of AcuteBayou with unconstrained operations' semantics (in contrast ANNC features weak updating operations that always return ok). Because we strive in this section for self-contained proofs we do not refer to the proofs for ANNC even when doing so would allow us to omit some repetitions.

We begin with the proof of guarantees offered by AcuteBayou in the stable runs.

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

Proof. For any given arbitrary stable run of AcuteBayou represented by a history $H = (E, op, rval, rb, ss, lvl)$ we have to find suitable vis, ar and par, such that $A = (H, vis, ar, par)$ is such that $A \models \text{FEC}(weak, \mathcal{F}) \wedge \text{LIN}(strong, \mathcal{F})$.

Additional observations. All events besides weak RO ones, have an associated unique Req record which is disseminated using RB-cast and TOB-cast; let us denote by $req(e)$ the Req record of the event e.[1] Since, the handling of weak RO

[1] Thus a trace of the $state$ object, which consists of such records, can be translated into a sequence of events.

events, which are *local* to a replica, differ significantly from other events, which are *shared*, we divide the set of all events E into two subsets: $\Psi \subseteq E$, consisting of weak updating, strong updating and strong RO events; and $\Omega \subseteq E$, consisting of weak RO events. We also further divide Ψ into subsets Ψ_w and Ψ_s, consisting of, respectively, weak and strong events.

Upon TOB-delivery of a COMMIT message, the received request r is *committed* (Algorithm 4 line 21), i.e., it is appended at the end of the $committed$ list, and removed from the $tentative$ list (if present there). Note that the position of r established on the $committed$ list never changes as the list is only appended to. Once the request is committed, the operation associated with the request is eventually executed (unless the request was already executed in the order consistent with the commit order) and then the request is never rolled back. This is so, because:

- the $committed$ list is included in the $newOrder$ list as a prefix in the commit procedure (Algorithm 2 line 33),
- until the request r executes it has to feature on the list $toBeExecuted$ (Algorithm 2 line 47) and there can be only a finite number of items preceding it on that list,
- the $toBeRolledBack$ list cannot grow indefinitely without executing some of the requests from the $toBeExecuted$ list, which means that r is eventually executed (Algorithm 2 line 55),
- and finally a request which is included in both the $committed$ and $executed$ lists is never part of the $outOfOrder$ list (Algorithm 2 line 45), which means it will not be scheduled for rollback.

Weak operations execute atomically in the invoke code block where the response is always returned immediately to the client.[2] For a given weak event e the response is computed on the $state$ object in some state S_α, where α is the current trace of the $state$ object at the time of the operation's invocation. We let $trace(e)$ denote the trace α.

On the other hand, strong operations follow a more complicated route. For a strong event e: firstly the COMMIT message is TOB-cast, then upon its TOB-delivery the request $r = req(e)$ is committed. Since r is not disseminated using RB-cast, it is never included in the $tentative$ list, and so it executes for the first time after its commit. Thus, each strong operation is executed on each replica exactly once, on a $state$ object in some state S_α, where α is the current trace of the $state$ object at the time of the execution. Note that the trace α is exactly the same on each replica and it consists exactly of all the requests preceding $req(e)$ in the $committed$ list (which due to the properties of TOB-delivery has the same value on each replica upon r's commit). Again, as in case of weak events, we let $trace(e)$ denote the trace α.

Note that each strong operation executed in some event e finishes only after

[2] If due to operation reexecutions multiple responses are returned to the client we discard the additional ones.

the replica TOB-delivers the message $m = msg_{TOB}(e)$. It means that for every operation executed in event e', such that $e \xrightarrow{rb} e'$, if $msg_{TOB}(e') = m'$ ($m' \neq \perp$), then $tobNo(m) < tobNo(m')$.

Arbitration. We construct the total order relation ar by sorting all shared events based on the order in which their respective TOB-cast messages are TOB-delivered, i.e., respecting the $tobNo$ order.

Next, we interleave the shared events with local events in the following way: each local event e occurs in ar after the last shared event e' such that $e \not\xrightarrow{rb} e'$. Thus, for each shared event e' the following holds $e \xrightarrow{ar} e' \Rightarrow e \xrightarrow{rb} e'$. The relative order of local operations is irrelevant.

We construct the perceived arbitration order $par(e)$, for each event e, using the trace $\alpha = trace(e)$. More precisely, we add all the events whose requests appear in α in the order of occurrence, next we add all the remaining shared events according to their order in ar. Finally, we interleave the constructed sequence with local events in a similar way as in case of ar, i.e., for each local event f and each shared event g, the following holds $f \xrightarrow{par(e)} g \Rightarrow f \xrightarrow{rb} g$.

Note that for a strong event e, $par(e) = ar$. This is because e executes once $req(e)$ is on the *committed* list, and its position on the list is determined by the $tobNo$ order, which means that the trace α contains exactly all the shared events preceding e in ar.

Visibility. For any two events $e, e' \in E$, such that $trace(e') = \alpha$, we include an edge $e \xrightarrow{vis} e'$ in our construction of vis, if:

1. $e \in \Psi$, $e' \in \Psi_s$, and $req(e) \in \alpha$,
2. $e \in \Psi_s$, $e' \in \Psi_w$, and $req(e) \in \alpha$,
3. $e \in \Psi_s$, $e' \in \Omega$, and $req(e) \in \alpha$,
4. $e \in \Psi_w$, $e' \in \Psi_w \cup \Omega$, and $req(e) \in \alpha$,
5. $e, e' \in \Omega$, and $e \xrightarrow{rb} e'$,
6. $e \in \Omega$, $e' \in \Psi$, and $e \xrightarrow{ar} e'$.

The edges 1-4 are essential, while the edges 5-6 are non-essential.

Note that in case of edge 4, either $e \xrightarrow{TOBdel} e'$, or $e \xrightarrow{RBdel} e'$, and thus $e \xrightarrow{rb} e'$ is implied (see the general observations in Section B). Thus, for all edges 4-6, $e \xrightarrow{rb} e'$.

Additionally, observe that in case of edge 1, $e \xrightarrow{TOBdel} e'$, because α contains only requests on the *committed* list (see the additional observations in the beginning of the proof), and thus $e \xrightarrow{tobNo} e'$. Similarly, in case of edges 2 and 3, $e \xrightarrow{TOBdel} e'$, because $msg_{RB}(e) = \perp$ and thus $req(e)$ can appear in α only if it was TOB-delivered by the replica executing e'. Also in case of edge 2, $e \xrightarrow{tobNo} e'$.

Having defined A (through vis, ar and par), it now remains to show that $A \models \text{FEC}(weak, \mathcal{F}) \wedge \text{LIN}(strong, \mathcal{F})$, or more specifically $A \models \text{EV}(weak) \wedge \text{EV}(strong) \wedge \text{NCC}(weak) \wedge \text{NCC}(strong) \wedge \text{FRVAL}(weak) \wedge \text{RVAL}(strong) \wedge \text{CPAR}(weak) \wedge \text{SINORD}(strong) \wedge \text{RT}(strong)$.

Eventual visibility. We prove now that eventual visibility is satisfied for all events:

- each shared event e is visible to all subsequent events from some point, because $msg_{TOB}(e)$ is eventually TOB-delivered and $r = req(e)$ is placed on the $committed$ list on each replica, thus r is eventually executed and never rolled back, and is included in the trace of the $state$ object from some point (1, 2, 3 and 4),
- each local event e is visible to all subsequent local events from some point (5),
- each local event e is visible to all subsequent shared events from some point, because by construction of ar there is only a finite number of events e' such that $e \xrightarrow{ar} e'$ (6).

No circular causality. We need to show that $acyclic(hb \cap (W \times W))$ and $acyclic(hb \cap (S \times S))$, where $W \subseteq E, S \subseteq E$, are, respectively, the sets of all weak, and strong events. We elect to prove a more general case of $acyclic(hb)$.

Recall that $hb = (vis \cup so)^+$. If $acyclic(vis \cup so)$, then $acyclic(hb)$, because transitive edges cannot introduce cycles. Thus, we have six types of edges to consider: edges 1-6 from vis and the seventh edge $e \xrightarrow{so} e'$. We divide them into two groups: the first one consists of edges 1-3, while the second one consists of edges 4-7. Note that for the second group $e \xrightarrow{rb} e'$ always holds.

There can be no cycles when we restrict the edges only to the ones from the first group, as the edges 1 and 2 are constrained by the $tobNo$ order, and edge 3 leads to a local event which cannot be followed using only edges from the first group.

Also, there can be no cycles when we restrict the edges only to the ones from the second group, as all the edges are constrained by the rb relation, which is naturally acyclic.

Thus, a potential cycle could only form when we mix edges from both groups. Let us assume that the cycle contains the following chain of edges: $a \xrightarrow{hb} b \xrightarrow{hb} \ldots \xrightarrow{hb} c \xrightarrow{hb} \ldots \xrightarrow{hb} d$, where $a, b, c, d \in E$, all the edges between b and c belong to the second group, while the other ones belong to the first group. Notice that $b \xrightarrow{rb} c$, and that $a, c \in \Psi$. Thus, the chain consists of a series of edges from the first group and a series of edges from the second group. The whole cycle can be combined from multiple such chains, but for simplicity, let us assume that it contains only one such chain and that $d = a$ (the same reasoning as below can be applied iteratively for multiple interleavings of edges from the two groups).

If $b \in \Psi$, then $a \xrightarrow{tobNo} b$ (edges 1 and 2), and since $b \xrightarrow{rb} c$, also $b \xrightarrow{tobNo} c$ (see the additional observations in the beginning of the proof). A contradiction: $a \xrightarrow{tobNo} b \xrightarrow{tobNo} c \xrightarrow{tobNo} a$.

If $b \in \Omega$, then $a \in \Psi_s$, and $a \xrightarrow{TOBdel} b$ (edge 3). Either $a \xrightarrow{tobNo} c$, or $c \xrightarrow{tobNo} a$. In the former case we end up with a similar contradiction as above: $a \xrightarrow{tobNo} c \xrightarrow{tobNo} a$. In the latter case, since $c \xrightarrow{tobNo} a$, also $c \xrightarrow{TOBdel} b$ (the message $msg_{TOB}(c)$ is TOB-delivered before the message $msg_{TOB}(a)$). However, $b \xrightarrow{rb} c$,

which means that the message $msg_{TOB}(c)$ was not even TOB-cast yet when b executed. A contradiction.

Single order. Since there are no pending strong operations (because eventually every message is TOB-delivered and the operations finish), we have to simply prove that $vis \cap (E \times S) = ar \cap (E \times S)$, where $S = \{e : lvl(e) = strong\}$. In other words, for any two events $e \in E, e' \in S$: $e \xrightarrow{vis} e' \Leftrightarrow e \xrightarrow{ar} e'$.

Let us begin with $e \xrightarrow{vis} e' \Rightarrow e \xrightarrow{ar} e'$. Either $e \in \Psi$, and thus $e \xrightarrow{tobNo} e'$ (edge 1), or $e \in \Omega$ (edge 6). In both cases $e \xrightarrow{ar} e'$.

Now let us consider $e \xrightarrow{ar} e' \Rightarrow e \xrightarrow{vis} e'$. Either $e \in \Psi$, or $e \in \Omega$. In the former case, $e \xrightarrow{tobNo} e'$, and thus e must be included in $trace(e')$, which means that $e \xrightarrow{vis} e'$ (edge 1). In the latter case, $e \xrightarrow{rb} e'$ (by construction of ar), and thus also $e \xrightarrow{vis} e'$ (edge 6).

Return value consistency. Since for a strong event e, $par(e) = ar$ and $fcontext(A, e) = context(A, e)$. Thus, for each event $e \in E$, we need to show that: $rval(e) = \mathcal{F}(op(e), fcontext(A, e))$. We base our reasoning below on essential vis edges and $par(e)$ order.

Firstly, observe that we can exclude from $fcontext(A, e)$ all local events which by the definition of an RO operation are irrelevant for the computation of $\mathcal{F}$. Thus, let $C = (E_C, op, vis, par(e))$, where $E_C = \{e' \in \Psi : e' \xrightarrow{vis} e\}$.

Then, recall that $rval(e)$ is obtained by calling $state$.execute on the $state$ object in state S_α, where $\alpha = trace(e)$, and that $rval(e) = \mathcal{F}(op, C_\alpha)$, where $C_\alpha = (E_\alpha, op_\alpha, vis_\alpha, ar_\alpha)$ is a context constructed from α as defined in Section A. It suffices to show that the context C is isomorphic with C_α, which we do below.

Clearly, by construction of vis, if $e' \xrightarrow{vis} e$ and $e' \in E_C$, then $req(e') \in \alpha$. Thus, E_α consists of the Req records of the events in E_C. By the way how Req records are constructed (Algorithm 4 line 9), for any given event $e \in E_C$, $op_\alpha(req(e))$ equals $op(e)$. Also, for any two events $f, g \in E_C$, $f \xrightarrow{par(e)} g \Leftrightarrow req(f) \xrightarrow{ar_\alpha} req(g)$, which follows trivially from the construction of $par(e)$. It remains to show that for any two events $f, g \in E_C$, $f \xrightarrow{vis} g \Leftrightarrow req(f) \xrightarrow{vis_\alpha} req(g)$.

If $g \in \Psi_w$ and $f \xrightarrow{vis} g$, then $req(f) \in trace(g)$, and thus $req(f).id \in req(g).ctx$, which implies $req(f) \xrightarrow{vis_\alpha} req(g)$.

If $g \in \Psi_w$ and $req(f) \xrightarrow{vis_\alpha} req(g)$, then $req(f).id \in req(g).ctx$, and thus $req(f) \in trace(g)$, which implies $f \xrightarrow{vis} g$.

If $g \in \Psi_s$ and $f \xrightarrow{vis} g$, then $f \xrightarrow{ar} g$ (by Single Order), and thus $f \xrightarrow{tobNo} g$. Since $req(g)$ is committed at the time of e's execution ($req(g) \in \alpha$ and $lvl(g) = strong$), so is $req(f)$ but its position on the $committed$ list is earlier ($f \xrightarrow{tobNo} g$). Because the order of requests in the trace is based on the $executed$ list, whose order is consistent with the order of the $committed$ list, $req(f)$ precedes $req(g)$ in α, which implies $req(f) \xrightarrow{ar_\alpha} req(g)$. Then, by construction of C_α, $req(f) \xrightarrow{vis_\alpha} req(g)$.

If $g \in \Psi_s$ and $req(f) \xrightarrow{vis_\alpha} req(g)$, then $req(f) \xrightarrow{ar_\alpha} req(g)$, and thus $req(f)$ precedes $req(g)$ in α. Since $req(g)$ is committed at the time of e's execution, both $req(f)$ and $req(g)$ belong to the $committed$ list during the e's execution, which

implies that $f \xrightarrow{tobNo} g$. Thus, $f \xrightarrow{ar} g$, and by Single Order, $f \xrightarrow{vis} g$.

Thus, C is isomorphic with C_α.

Convergent perceived arbitration. We now show, that for each event $e \in E$ there exist only a finite number of weak events e', such that the prefixes of $par(e')$ and ar up to the event e differ, which is a sufficient condition to prove CPAR($weak$).

If $e \in \Psi$, then eventually on each replica $msg_{TOB}(e)$ is TOB-delivered, and $req(e)$ is committed and executed. Thus, from some point, the trace of each subsequent event e' contains $req(e)$, preceded by requests of events e'' committed earlier, such that $e'' \xrightarrow{tobNo} e$. Both ar and $par(e')$ are constructed by first ordering shared events and then interleaving them with local events using the same procedure. In both ar and $par(e')$, e is preceded by the same shared events e'', such that $e'' \xrightarrow{tobNo} e$. Then, it is also preceded by the same local events, which means the prefixes of $par(e')$ and ar up to e are equal.

If $e \in \Omega$, then eventually the requests of all shared events e'', such that $e'' \xrightarrow{ar} e$, are committed and executed on each replica. Then, from some point, the trace of each subsequent event e' contains the requests of events e'', ordered by $tobNo$. Thus, e is preceded in both ar and $par(e')$ by the same shared events e''. Because both ar and $par(e')$ are interleaved with local events using the same procedure, e is also preceded in both ar and $par(e')$ by the same local events, which means the prefixes of $par(e')$ and ar up to e are equal.

Real-time order. We need to show that arbitration order respects real-time order of strong operations, i.e., $rb \cap (S \times S) \subseteq ar$, where $S = \{e : lvl(e) = strong\}$. In other words, for any two $e, e' \in S$: $e \xrightarrow{rb} e' \Rightarrow e \xrightarrow{ar} e'$.

Clearly, if $e \xrightarrow{rb} e'$, then $e \xrightarrow{tobNo} e'$ (see the additional observations in the beginning of the proof). Thus, $e \xrightarrow{ar} e'$ (by construction of ar). ■

Now, let us continue with the proof of the guarantees offered by AcuteBayou in the asynchronous runs.

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* ($\mathcal{F}, lvlmap$).

Proof. To show the inability of AcuteBayou to satisfy LIN($strong, \mathcal{F}$) in asynchronous runs, it is sufficient to observe that due to some of the TOB-cast messages not being TOB-delivered, some of the strong operations remain pending. A pending operation's return value equals ∇ which is irreconcilable with the requirements of the predicate RVAL($\mathcal{F}$).

The proof regarding the guarantees of the weak operations is similar to the one for the stable runs, thus we rely on it and focus only on differences between stable and asynchronous runs that need to be addressed. Now for any given arbitrary asynchronous run of AcuteBayou represented by a history $H = (E, op, rval, rb, ss, lvl)$ we have to find suitable vis, ar and par, such that $A = (H, vis, ar, par)$ is such that $A \models$ FEC($weak, \mathcal{F}$).

Additional observations. The same observations apply as in case of stable runs, with the only distinction that some strong events e remain pending due to the

lack of TOB-delivery of $msg_{TOB}(e)$. In such cases $trace(e)$ is undefined.

Now let us make one more observation: the request of a weak updating event e whose $msg_{TOB}(e)$ is never TOB-delivered, even though it never commits, eventually *settles*, i.e. it is eventually executed and is never rolled back after that execution. It is so, because after $r = req(e)$ is RB-delivered by each replica and placed on the $tentative$ list, only a finite number of other requests can commit (due to the properties of TOB in asynchronous runs), and also only a finite number of other requests can have a lesser Req record (as defined by the operator $<$ in Algorithm 2) and thus precede r in the $tentative$ list (due to monotonically increasing clocks on each replica). Thus, once r is placed on the $toBeExecuted$ list, it eventually executes, and when executed r can be rolled back at most a finite number of times, due to a commit of other request, or a lesser Req being inserted into the $tentative$ list.

Arbitration. We construct the total order relation ar by sorting all shared events based on the order in which their respective TOB-cast messages are TOB-delivered, i.e., respecting the $tobNo$ order. Shared events whose messages are not TOB-delivered are ordered after those whose messages are TOB-delivered, with weak updating events appearing first, ordered relatively based on their Req records, followed by pending strong events.

Next, we interleave the shared events with local events in the following way: each local event e occurs in ar after the last *non-pending* shared event e' such that $e \overset{rb}{\not\longrightarrow} e'$. Thus, for each non-pending shared event e' the following holds $e \overset{ar}{\longrightarrow} e' \Rightarrow e \overset{rb}{\longrightarrow} e'$. The relative order of local events is irrelevant.

We construct the perceived arbitration order $par(e)$ for each event e, in the same way as in case of stable runs, i.e. using $trace(e)$, the remaining shared events from ar, and finally interleaving the constructed sequence with local events as in case of ar (so that for each local event e' and each non-pending shared event e'', the following holds $e' \overset{par(e)}{\longrightarrow} e'' \Rightarrow e' \overset{rb}{\longrightarrow} e''$.

For a pending strong event e, which was not executed at all, we let $par(e) = ar$.

Note that for a non-pending strong event e, $par(e) = ar$. This is because e executes once $req(e)$ is on the $committed$ list, and its position on the list is determined by the $tobNo$ order, which means that its trace will contain exactly all the shared events preceding e in ar.

Visibility. We construct the visibility relation in the same way as in the stable runs case. However, we remove edges to and from pending strong events. Since pending operations do not provide a return value, no edge to a pending event is essential. Also, as we guarantee only eventual visibility for weak events, edges to strong events are not necessary to satisfy EV($weak$). Moreover, edges *from* pending events are not needed either, because by definition a pending event is never followed in rb by any other event (which is a requirement to fail the test for EV). Again, for all edges 4-6, $e \overset{rb}{\longrightarrow} e'$.

Having defined A (through vis, ar and par), it now remains to show that $A \models \text{FEC}(weak, \mathcal{F}_{NNC})$, or more specifically $A \models \text{EV}(weak) \wedge \text{NCC}(weak) \wedge \text{FRVAL}(weak) \wedge \text{CPAR}(weak)$.

Eventual visibility. We prove now that eventual visibility is satisfied for all *weak* events:

- each non-pending shared event e, such that $msg_{TOB}(e)$ is eventually TOB-delivered on each replica, is visible to all subsequent non-pending events from some point, because $r = req(e)$ is placed on the $committed$ list on each replica, thus r is eventually executed and never rolled back, and is included in the trace of the $state$ object from some point (2, 3 and 4),
- each weak updating event e, such that $msg_{TOB}(e)$ is *not* eventually TOB-delivered on each replica, is visible to all subsequent non-pending events from some point, because it settles (see the additional observations in the beginning of the proof) and is included in the trace of the $state$ object on each replica from some point (4),
- each local event e is visible to all subsequent local events from some point (5),
- each local event e is visible to all subsequent non-pending shared events from some point, because by construction of ar there is only a finite number of events e' such that $e \xrightarrow{ar} e'$ (6).

No circular causality. We use exactly the same reasoning as in the stable runs case to show that $acyclic(hb)$ holds true.

Return value consistency. Again, we use exactly the same reasoning as in the stable runs case to show that for each *weak* event $e \in E$: $rval(e) = \mathcal{F}_{NNC}(op(e), fcontext(A, e))$. Although this time we only need to prove return value consistency for weak operations, it can be shown that it also holds for *non-pending* strong events.

Convergent perceived arbitration. We now show, that for each non-pending[3] event $e \in E$ there exist only a finite number of weak events e', such that the prefixes of $par(e')$ and ar up to the event e differ, which is a sufficient condition to prove CPAR($weak$).

If $e \in \Psi$ and $msg_{TOB}(e)$ is eventually TOB-delivered, then the same logic can be applied as in case of stable runs to show that from some point for each subsequent event e' the prefixes of $par(e')$ and ar up to e are equal.

If $e \in \Psi_w$ and $msg_{TOB}(e)$ is never TOB-delivered, then it eventually settles (see the additional observations in the beginning of the proof) and thus also the same logic can be applied as in case of stable runs, with the distinction that e is preceded in ar and $par(e')$ not only by events e'' whose requests are committed, but also by events e'', such that $req(e'') < req(e)$.

If $e \in \Omega$, then eventually the requests of all shared events e'', such that $e'' \xrightarrow{ar} e$ (none of which are pending by the construction of ar), are either committed, or settled, and executed on each replica. Then, from some point, the trace of each subsequent event e' contains the requests of events e'', ordered by both $tobNo$, and based on their Req records. Thus, e is preceded in both ar and $par(e')$ by the

[3]We can exclude pending events, because according to the construction of vis they are not visible to any other event, and thus automatically satisfy the requirements of the CPAR predicate.

same shared events e''. Because both ar and $par(e')$ are interleaved with local events using the same procedure, e is also preceded in both ar and $par(e')$ by the same local events, which means the prefixes of $par(e')$ and ar up to e are equal. ■

Streszczenie

Globalne usługi, które stanowią serce dzisiejszego internetu, takie jak komunikatory, sieci społecznościowe, handel elektroniczny, bankowość, rachunki maklerskie, czy gry online, są realizowane przez skomplikowane systemy rozproszone. Aby sprostać rosnącemu obciążeniu generowanemu przez miliony użytkowników, te systemy muszą być skalowalne horyzontalnie, co oznacza, że można zwiększyć ich wydajność poprzez dodanie dodatkowych węzłów obliczeniowych. Spełnienie owych ogromnych wymagań dotyczących skalowalności jest dodatkowo utrudnione przez fakt, że systemy te muszą pozostawać sprawne przez cały czas. Jako, że żaden sprzęt komputerowy, czy sieciowy, nie są całkowicie odporne na usterki techniczne, same usługi muszą być zaimplementowane w taki sposób, który umożliwi im z łatwością tolerować awarie. W ten sposób systemy realizujące te usługi mogą stać się *wysoko dostępne*, tzn. mogą przetwarzać żądania klientów nawet gdy występują (częściowe) awarie systemu.

Powszechną techniką stosowaną w celu zwiększenia dostępności systemu jest *replikacja*, która polega na utrzymywaniu wielu kopii danych i kodu usługi, zwanych *replikami*, na fizycznie niezależnych węzłach, często rozproszonych geograficznie. Poza zapewnieniem odporności na awarie, replikacja poprawia skalowalność i obniża czasy odpowiedzi gdy repliki znajdują się geograficznie blisko klientów. Tradycyjne schematy replikacji, takie jak *zreplikowana maszyna stanowa* [15, 16] czy *replika główna – replika zapasowa* (ang. *primary-backup*) [17], utrzymują silną spójność pomiędzy replikami, tzn. repliki koordynują zmiany swoich stanów w taki sposób, aby system jako całość dla klientów sprawiał wrażenie pojedynczego scentralizowanego serwera. Jednakże, utrzymywanie replik w stanie synchronizacji jest kosztowne, ponieważ zazwyczaj wymaga rozwiązania problemu rozproszonego konsensusu. Z tego powodu przed wysłaniem odpowiedzi do klienta repliki muszą wymienić pomiędzy sobą wiele wiadomości, co znacząco zwiększa czas odpowiedzi. Ponadto, utrzymanie spójności replik jest niemożliwe gdy występują podziały sieci, a usługa powinna pozostać dostępna, jak stanowi słynne twierdzenie CAP [18]. Dlatego tradycyjne

silnie spójne schematy replikacji gwarantują jedynie dostępność w przypadku (ograniczonej liczby) awarii replik, ale nie w przypadku awarii sieci powodujących brak łączności pomiędzy grupami replik.

W celu pokonania powyższych ograniczeń, wymagania co do spójności mogą być osłabione. Kiedy stosowana jest jakaś odmiana *spójności ostatecznej* [19] repliki mogą synchronizować swoje stany jedynie ostatecznie. Znaczy to, że repliki mogą przetwarzać żądania klientów niezależnie i rozsyłać zmiany stanów asynchronicznie. Z tego powodu, nawet gdy występują podziały sieci, wysoka dostępność może być utrzymana. By osiągnąć ten cel, ostatecznie spójne systemy cechują się zdecentralizowaną architekturą i polegają na (asynchronicznej) komunikacji peer to peer (p2p). Jest to model wprowadzony pierwszy raz przez Amazon w ich wpływowym magazynie danych Dynamo [20] i który został powielony w wielu popularnych magazynach danych NoSQL (przykładowo w Apache Cassandra [21], Scylla [22], Riak [23], Voldemort [24] oraz Netflix Dynomite [25]).

Jednakże, osłabione modele spójności oferują słabsze gwarancje i z zasady dopuszczają pewną ilość niespójności. Jeżeli nie są obsłużone prawidłowo, mogą prowadzić to niepożądanych anomalii, w tym utraty danych. Z tego powodu programiści muszą ostrożnie projektować kod zreplikowanych usług tak by obsłużyć wszystkie przypadki graniczne i uwzględnić możliwość występowania anomalii. Aby zredukować obciążenie programistów stosowane są specjalne struktury danych, zwane *wolnymi od konfliktów, replikowanymi typami danych* (ang. *conflict-free replicated data types, CRDTs*) [26, 27, 28]. CRDT mogą być zaimplementowane w sposób całkowicie asynchroniczny i z zasady zapewniają ostateczną zbieżność stanu replik. Do popularnych CRDT zaliczają się *wielowartościowe rejestry* (ang. *multi-value registers, MVRs*), *rejestry typu ostatni-zapis-wygrywa* (ang. emphlast-write-wins registers, LWW-registers), *pozytywno-negatywne liczniki* (ang. *positive-negative-counters, PN-counters*), *zbiory zaobserwowane-usuń* (ang. *observed-remove sets, OR-sets*) [27], jak i struktury danych do współedytowania tekstu online [29].

Niestety semantyka CRDT jest bardzo ograniczona. Aby zapewnić wysoką dostępność, niskie czasy odpowiedzi oraz ostateczną zbieżność stanów replik, struktury te wymagają aby wszystkie operacje były naprzemienne, albo by istniały naprzemienne, asocjacyjne, idempotentne procedury scalania stanów replik. Dlatego struktury te nie nadają się do wszystkich zastosowań. Dla przykładu rozważmy pojedynczy nieujemny licznik całkowitoliczbowy. Operacja dodania może być trywialnie zaimplementowana w sposób wolny od konfliktów, ponieważ operacje dodawania są naprzemienne. Jednakże implementacja operacji odejmowania wymaga globalnego uzgodnienia żeby zapewnić, że wartość licznika nigdy nie spadnie poniżej zera. Podobnie w systemie aukcyjnym współbieżne oferty mogą być uznane za operacje niezależne, więc ich wykonanie nie wymaga synchronizacji. Jednakże operacja, która zamyka aukcję wymaga rozwiązania rozproszonego konsensusu by wybrać jedną zwycięską ofertę [30].

Z powodu ograniczeń CRDT, i spójności ostatecznej w ogólności, w ostatnim

czasie podejmowanych było wiele prób, zarówno w przemyśle [31, 32, 33, 34], jak i w nauce [35, 36, 37, 38, 39, 40, 41, 42], żeby wzmocnić semantykę systemów ostatecznie spójnych poprzez dopuszczenie do wykonywania części operacji z silniejszymi gwarancjami spójności, lub poprzez dodanie funkcji pseudotransakcyjnych. W ten sposób wyłoniła się nowa klasa systemów wysoko dostępnych, zwana *systemami o mieszanej spójności*, w której to jedynie część operacji musi być wysoko dostępna. Operacje wykonywane ze słabszymi gwarancjami spójności, zwane *słabymi*, pozostają wysoko dostępne nawet w obliczu występowania awarii, podczas gdy operacje wykonywane w trybie silnie spójnym, zwane *silnymi*, mogą się blokować, np z powodu awarii sieci. Analiza i formalizacja własności poprawności systemów o mieszanej spójności stanowią główny temat tej pracy.

Motywacje

Tak jak to zostało powyżej omówione, z powodu rosnącego znaczenia wysokiej dostępności w kontekście współczesnych globalnych usług w internecie, systemy wysoko dostępne, włączając w to rozwiązania warstwy pośredniczącej (ang. middleware) takie jak magazyny danych NoSQL, dynamicznie się rozpowszechniają. Z powodu naszego rosnącego uzależnienia od tych systemów, badanie i weryfikacja ich poprawności stanowi problem najwyższego znaczenia. Mimo, że istnieje obecnie znacząca liczba badań w tym obszarze, wciąż wiele pozostaje do zrobienia.

Przez długi czas spójność ostateczna unikała klarownego ujęcia i sformalizowania. Wiele definicji zostało zaproponowanych (zobacz np. [19, 43, 44, 45, 46, 2, 26, 37, 47, 48, 49]), które różniły się znacząco zarówno pod względem użytych technik formalizacji, jak i praktycznie oferowanych gwarancji. Z drugiej strony silna spójność, która jest stosowana od wielu dekad (zobacz np. [50, 51, 52]), jest dużo lepiej zrozumiana. Jest tak ponieważ silna spójność opiera się na bardzo prostej zasadzie: system silnie spójny wykonujący żądania współbieżnie powinien być nierozróżnialny od systemu wykonującego żądania sekwencyjnie. W porównaniu do spójności silnej, spójność ostateczna zapewnia gwarancje, które są nie tylko znacznie słabsze, ale również trudne do zrozumienia z powodu ich skomplikowania albo nieprecyzyjności. Dla przykładu definicja podana przez Vogelsa [19] mówi, że kiedy zapisy ustaną, ostatecznie wszystkie odczyty zwrócą tą samą wartość, ale definicja ta nie nakłada żadnych ograniczeń na zwrócone wartości gdy zapisy nigdy nie ustają. Znowuż *typy chmurowe* (ang. *cloud types*) [37] wymagają od użytkownika myślenia w kategoriach rewizji, które mogą się dzielić i łączyć jak w systemie kontroli kodu źródłowego. Jest to tzw model *spójności rewizji* (ang. *revision consistency*) [47], który został ostatecznie porzucony ze względu na nadmierne skomplikowanie [38]. Z tego powodu udowadnianie poprawności konkretnego systemu ostatecznie spójnego, jak również wnioskowanie na temat takich systemów w ogólności, jest bardziej wymagające. Co więcej, gdy ostatecznie spójne (słabe) operacje są mieszane z operacjami silnie spójnymi (silnymi) w jednym systemie o spójności mieszanej,

klarowność oferowanych gwarancji jest jeszcze mocniej obniżona. Istotnie, nie ma obecnie konsensusu co do oczekiwanej semantyki tego typu systemów.

Systemom o mieszanej spójności wykorzystywanym w przemyśle brakuje klarownie zdefiniowanej semantyki, albo mają tę semantykę znacząco ograniczoną. Dla przykładu wykonywanie *lekkich transakcji* w Apache Cassandra [21] na danych, które są w tym samym czasie modyfikowane przez zwykłe operacje ostatecznie spójne, prowadzi do niezdefiniowanego stanu systemu (ang. undefined behaviour) [53]. Z drugiej strony w Riaku [23] elementy, do których dostęp jest realizowany przez operacje słabe i elementy, do których dostęp jest realizowany przez operacje silne, muszą się mieścić w oddzielnych, niezależnych przestrzeniach, zwanych *kubełkami*, [34], przez co system tak naprawdę nie posiada semantyki spójności mieszanej. Inne systemy [39, 31, 32] umożliwiają dobór poziomu spójności jedynie dla operacji odczytu. Klienci mogą wybrać odczyty świeże albo potencjalnie przestarzałe (ang. stale). Z kolei zapisy w tym podejściu są zawsze wykonywane jako operacje silne.

Wszystkie znane podejścia, które faktycznie spełniają semantykę mieszanej spójności posiadają jakieś ograniczenia w kontekście ich działania w obliczu awarii. Dla przykładu w *typach chmurowych* [37], jak i w *globalnym protokole sekwencyjnym* [38], wszystkie operacje zapisu (zarówno słabe jak i silne) muszą być przekazane do scentralizowanego podsystemu, zwanego *chmurą*, którego zadaniem jest rozgłaszanie do wszystkich węzłów komunikatów o zmianach stanów w uporządkowanym strumieniu. Kiedy chmura jest niedostępna, np z powodu awarii większości serwerów działających w chmurze, albo z powodu podziału sieci, operacja zapisu wciąż może zostać wykonana i zaaplikowana lokalnie na którymś z węzłów, ale nie będzie widoczna dla innych węzłów. Jako kolejny przykład rozważmy *replikację leniwą* [54], *spójność niebiesko-czerwoną* [35], oraz *częściowe ograniczenia porządku* [36], w których to wszystkie repliki muszą pozostawać sprawne, aby możliwe było wykonanie operacji silnej na dowolnej z replik. W związku z tym awaria pojedynczej repliki może zablokować zdolność systemu do wykonywania operacji silnych, aż do naprawienia usterki. Typowe silnie spójne systemy zreplikowane wykorzystujące *nie-blokujące* protokoły uzgadniania, takie jak Paxos [55], mogą tolerować awarie aż do połowy wszystkich replik i wciąż przetwarzać operacje. Tak więc niemożność tolerowania nawet pojedynczej awarii w systemie wysoko dostępnym, który powinien z łatwością tolerować awarie, wydaje się głęboko niezadowalająca, nawet jeżeli owa niemożność dotyczy jedynie operacji silnych.

Rozwiązania omówione powyżej w obliczu awarii idą na kompromis osłabiając postęp, albo operacji słabych (nie propagując wytworzonych przez nie zmian), albo operacji silnych (blokując ich wykonanie). Takie kompromisy wynikające z mieszania operacji słabych i silnych są warte zbadania. W szczególności interesującym pytaniem, na które próbujemy znaleźć odpowiedź w tej pracy, jest to czy istnieje system o spójności mieszanej, który obsługuje operacje silne w sposób nie-blokujący (tzn toleruje przynajmniej jakąś liczbę awarii replik), jednocześnie nie ograniczając postępu operacji słabych. Taki system posiadałby najlepsze cechy systemów ostatecznie spójnych i silnie spójnych. Mianowicie,

oferowałby wysoką dostępność i niskie czasy dostępu w przypadku operacji słabych, oraz silne gwarancje i (ograniczoną) tolerancję awarii w przypadku operacji silnych. Następnym nasuwającym się pytaniem jest to jakie gwarancje poprawności taki system może oferować.

Niezależnie od tego czy dany system wysoko dostępny posiada dodatkowe operacje silne czy nie, zapewnienie jego poprawnego działania w obliczu awarii jest krytycznie istotne. Systemy te są celowo projektowane pod kątem scenariuszy, w których w każdej chwili mogą wystąpić awarie. Może być zatem zaskakującym fakt, że większość prac dotyczących poprawności systemów wysoko dostępnych, które można znaleźć w literaturze, całkowicie abstrahuje od problemu awarii replik lub sieci (zobacz np. [45, 35, 46, 56, 57, 58, 59, 60, 36, 61, 48, 49]). Analizy wykonane w ten sposób mogą być uważane za niekompletne: protokół, który działa poprawnie tylko gdy awarie nie występują, niekoniecznie działa poprawnie gdy awarie *występują*. Z tego względu przeprowadzenie szerokiej, wyczerpującej analizy poprawności systemów wysoko dostępnych z jawnym uwzględnieniem różnorodnych modeli awarii, może dostarczyć nowych wglądów i wiedzy na temat występujących kompromisów, które dotychczas pozostawały niezauważone w środowisku naukowym jak i w przemyśle.

Cele i wkład pracy

Biorąc pod uwagę powyższe motywacje, w następujący sposób formułujemy główną tezę dysertacji:

> *Ograniczenia i kompromisy występujące w osiągalnych gwarancjach poprawności systemów wysoko dostępnych wynikające z operacji o mieszanej spójności, oraz wynikające z występowania awarii serwerów i sieci, mogą zostać formalnie oznaczone i można na ich temat wnioskować.*

Potwierdzamy prawdziwość tezy w dwóch częściach. Po pierwsze oznaczamy i analizujemy kompromisy dotyczące poprawności wynikające z operacji o mieszanej spójności. Po drugie wykorzystując podejście holistyczne do analizy poprawności, którego częścią jest stworzenie wiernego modelu rzeczywistych systemów opartych o architekturę klient-serwer oraz uwzględnienie w sposób jawny występowania awarii, precyzyjnie określamy ograniczenia w gwarancjach poprawności tego typu systemów wynikające z występowania awarii.

Poniżej podsumowujemy osiągnięty wkład naukowy naszej pracy.

Po pierwsze w rozdziale 2 definiujemy *przenikliwe typy chmurowe* (ang. *acute cloud types*, ACT), jako abstrakcję dla systemów o spójności mieszanej, które łączą najlepsze cechy systemów spójnych ostatecznie oraz systemów silnie spójnych. ACT posiadają dwa rodzaje operacji: *operacje słabe* nacelowowane na nieograniczoną skalowalność i niskie czasy odpowiedzi (jak operacje w CRDT), oraz *operacje silne* używane gdy gwarancje spójności ostatecznej są niewystar-

czające. Operacje silne wykorzystują nie-blokującą synchronizację na bazie rozproszonego konsensusu. Zaproponowaliśmy i przeanalizowaliśmy przykładowy ACT o nazwie *przenikliwy licznik nieujemny* (ang. *acute non-negative counter*, ANNC). Przeanalizowaliśmy również wpływowy system Bayou i pokazaliśmy jak można go usprawnić aby stał się ACT ogólnego przeznaczenia o nazwie AcuteBayou. Dzięki temu zidentyfikowaliśmy niepożądane anomalie, które mogą się pojawić w Bayou. W szczególności odkryliśmy anomalię, którą nazwaliśmy *tymczasową zamianą kolejności operacji*, która okazała się nieodzowną, nieusuwalną cechą systemu Bayou i innych, które w podobny sposób co Bayou, używają dwóch niekompatybilnych ze sobą sposobów szeregowania operacji silnych i słabych.

Następnie w rozdziale 3 wyprowadziliśmy system formalny (ang. formal framework), który umożliwia wnioskowanie na temat ACT i innych systemów o spójności mieszanej. W ramach tego systemu sformalizowaliśmy kilka kryteriów poprawności, w tym *oscylującą spójność ostateczną* (ang. *fluctuating eventual consistency*, FEC), która adekwatnie oddaje gwarancje poprawności oferowane przez systemy dla których tymczasowa zamiana kolejności operacji jest nieodzowna. Korzystając z naszego systemu formalnego udowodniliśmy również poprawność ANNC i AcuteBayou.

Potem w rozdziale 4 uogólniliśmy nasze spostrzeżenia dotyczące AcuteBayou i zaproponowaliśmy rezultat formalny ukazujący ograniczenia w możliwych do osiągnięcia gwarancjach poprawności. Udowodniliśmy, że systemy o mieszanej spójności, które łączą najlepsze cechy systemów spójnych ostatecznie oraz systemów silnie spójnych, tak jak ACT, i które cechują się dowolnie skomplikowaną semantyką operacji, nie mogą uniknąć tymczasowej zamiany kolejności operacji. Wobec tego, systemy te nie spełniają własności podstawowej spójności ostatecznej (ang. basic eventual consistency) dla operacji słabych. Zbadaliśmy potencjalnie występujące kompromisy w gwarancjach poprawności oraz cechach systemów o mieszanej spójności poprzez poddanie analizie innych rozwiązań o mieszanej spójności, które nie posiadają wszystkich pożądanych cech ACT.

Dalej w rozdziale 5 pokazaliśmy jak adekwatnie modelować rzeczywiste systemy wysoko dostępne o architekturze klient-serwer w celu wnioskowania na temat ich poprawności w obliczu występowania awarii. Nakreśliliśmy możliwe scenariusze awarii i sklasyfikowaliśmy je w sześciu modelach awarii (ang. failure models). Podaliśmy również model formalny (ang. formal framework), w którym można jawnie specyfikować awarie. Model ten umożliwia formułowanie *świadomych awarii kryteriów poprawności*.

Kolejno w rozdziale 6 wyraziliśmy w naszym modelu gwarancje sesji i przeanalizowaliśmy ich własności i znaczenie. Pokazaliśmy, że klasyczne gwarancje sesji mogą być kontrproduktywne w przypadku niektórych typów danych i podaliśmy nowy substytut w ich miejsce, o nazwie *zachowanie kontekstu*.

Na koniec w rozdziale 7 przeanalizowaliśmy gwarancje poprawności, które są możliwe do osiągnięcia w sześciu różnych modelach awarii. W szczególności pokazaliśmy, że podstawowa spójność ostateczna nie jest osiągalna gdy

występują permanentne podziały sieci lub awarie replik, po których niemożliwe jest odtworzenie ich stanu. Zdefiniowaliśmy ostateczne warianty dwóch kluczowych gwarancji sesji i rodzinę świadomych awarii kryteriów poprawności, które precyzyjnie oddają gwarancje poprawności osiągalne w rozważanych modelach awarii. Zidentyfikowaliśmy również kilka niepożądanych anomalii, które mogą być zaobserwowane przez klientów gdy występują awarie. Pokazaliśmy jak można przeciwdziałać występowaniu niektórych z nich.

W przyszłości planujemy zaprojektować nowe przenikliwe typy chmurowe. W szczególności takie, które mogą być zastosowane praktycznie, np. w magazynie danych NoSQL. Ponadto, chcielibyśmy zbadać, które grupy operacji mogą być zaimplementowane w ACT bez tymczasowej zamiany kolejności operacji.