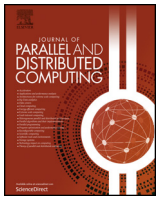
Journal of Parallel and Distributed Computing 180 (2023) 104707

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



On the correctness of highly available systems in the presence of failures ☆

Maciej Kokociński *, Tadeusz Kobus, Paweł T. Wojciechowski

Institute of Computing Science, Poznan University of Technology, Piotrowo 2, 90-965, Poland

ARTICLE INFO

Article history:
Received 12 March 2022
Received in revised form 4 April 2023
Accepted 21 April 2023
Available online 10 May 2023

Keywords:
Fault-tolerance
CAP
Eventual consistency

ABSTRACT

In this paper we formally study the guarantees provided by highly available, eventually consistent replicated systems in an environment in which machine failures and network splits are possible. Our analysis accounts for possible replica recovery after a crash, and clients that are (1) stateless or stateful, (2) sticky (always connect to a concrete set of replicas) or mobile, and (3) which can timeout before receiving a response to the sent request. We show why the approaches to prove protocol correctness prevalent in the literature, which do not take into account replica or network crashes, may lead to incorrect conclusions regarding the guarantees offered by the protocol. We adapt the existing formal correctness criteria, such as *basic eventual consistency*, to the considered environment by defining the family of *failure-aware* consistency guarantees. We formally identify a set of undesired phenomena (in particular *phantom operations*) observed by the clients, which, as we prove, are unavoidable in highly available systems in which unrecoverable replica crashes are possible. We also introduce *context preservation*, a new client-side requirement for eventually consistent systems that expose concurrency to the client, i.e., allow clients to use, e.g., multi-value registers or observed-remove sets. Context preservation is incomparable with classic session guarantees.

1. Introduction

In order to cope with the increasing traffic generated by millions of users, the distributed systems that power today's Internet must stay operational at all times. To this end, cloud service providers utilize redundant hardware and reliable networking infrastructure. Since hardware failures, such as transient or permanent machine crashes and network splits, cannot be avoided altogether, the services themselves must be implemented in a way that gracefully tolerates failures. More precisely, we consider *highly available* replicated systems which serve each request promptly even when (partial) failures occur. To this end, such systems feature a decentralized architecture and rely on peer to peer asynchronous communication protocols. It is a design brought into the mainstream in the Amazon's Dynamo storage system [28], and followed in a plethora of popular NoSQL data stores. Many of these systems employ various Conflict-free Replicated Data Types (CRDTs) [62,63] to offer rich semantics that includes, among others, highly available Multi-Value Registers, PN-Counters, Observed-Remove Sets, and structures for collaborative text editing [15]. However, high availability is at odds with strong consistency guarantees typical for, e.g., classic replicated DBMS [55] or state machine replication [44,73], as formalized by the CAP theorem [22,29].

Reasoning about correctness guarantees of a distributed system requires taking into account both the *safety* and *liveness* aspects of its behaviour, as shown by Lamport [42] and then further formalized by Alpern and Schneider [12]. Informally, proving safety involves showing that *nothing bad* ever happens in any execution of the system. On the other hand, satisfying a liveness guarantee ensures that eventually *something good* happens in an execution. These two properties require different proof techniques. Proving safety is often deemed easier, since it involves reasoning only about finite sequences of execution events: if nothing bad happens in all finite executions of a system, then by induction also nothing bad can happen in all infinite executions. On the other hand, proving liveness requires reasoning directly about infinite executions. It is because while in a finite execution a crashed process or a process that is part of a different network partition is indistinguishable from a process that executes very slowly, this is not

☆ This work was supported by the Foundation for Polish Science, within the TEAM programme co-financed by the European Union under the European Regional Development Fund (grant No. POIR.04.04.00-00-5C5B/17-00).

* Corresponding author.
E-mail addresses: Maciej.Kokocinski@cs.put.edu.pl (M. Kokociński), Tadeusz.Kobus@cs.put.edu.pl (T. Kobus), Pawel.T.Wojciechowski@cs.put.edu.pl (P.T. Wojciechowski).

https://doi.org/10.1016/j.jpdc.2023.04.008
0743-7315/

true in infinite executions. In turn, replica crashes and subsequent recoveries or possible network splits are much harder to be accounted for.

Interestingly, the majority of existing 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., [14,16,21,24,26,31,33,35,46–48,61,71]). It is easy to show that analysis conducted that way may lead to incorrect conclusions. For example, as we discuss in detail in Section 2, a simple implementation of a distributed register relying on the last-write-wins policy [66] and best-effort broadcast [27] can be deemed correct only when we consider no machine or network failures. However, once we take failures into account, the implementation does not satisfy even the simplest definition of eventual consistency, e.g., as defined by Vogels [70]: when updates cease eventually all read operations return the same value. On the other hand, the works that do concern failures, such as [32], fail to account for liveness properties which leads to similar results as above.

Another interesting example involves our observation regarding the CAP theorem and an existing, widely accepted and seemingly precise definition of eventual consistency, namely Burckhardt's *basic eventual consistency* (BEC) [26]. As shown in Section 6.1, by using the original set of assumptions, we can prove a stronger result: not only is linearizability incompatible with partition tolerance and high availability, but so is BEC. The reason for this counter-intuitive result lies in the fact that machine and network failures are not properly accounted for in the formal framework proposed by Burckhardt.

In this article we holistically approach the problem of correctness of highly available systems. To this end, we introduce a novel formal framework that explicitly considers hardware failures, such as transient or permanent machine crashes and network splits. We define system semantics through replicated data type specification, similarly to [26], thus allowing us to rigorously reason about both safety and liveness guarantees of a replicated system. We define a family of *fault-aware* consistency guarantees for highly available systems that specifically account for hardware failures and fit well with the intuition behind CAP.

To make our analysis accurately reflect how the contemporary replicated systems function, we model a client-server architecture with external clients that are *not* colocated with the service replicas. In contrast, in the classic model (e.g. [16]), 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. This choice has important consequences in terms of potentially achievable correctness guarantees. For example, it is known that causal consistency is achievable in highly available systems in the classic model, whereas it is not achievable with external clients [19].

Moreover, in our approach we do not consider *sticky clients* in the sense found in other works (e.g., in [18]), i.e., clients that issue all their requests always to exactly the same replica. The traditional notion of a sticky client is enticing, since it allows the system to achieve certain session guarantees, such as *read-your-writes* (RYW) [64], essentially for free. However, if a client is locked to a single replica and the replica suffers from an unrecoverable failure, then the client is unable to successfully complete any of its requests, which is against the very basic idea behind the notion of high availability. Unfortunately, the definitions of high availability found in the literature (e.g., [22,24,29]) do not preclude such a scenario, as they only concern replicas and not clients. According to these definitions, the system remains available as long as non-failed replicas eventually respond to every request. Thus, to avoid such pitfalls, in our approach we consider only *mobile* clients, which can connect to any replica, and *sticky* clients, which are limited to a group of replicas (e.g., replicas residing in the same geographic region), but not to a single replica.[1]

Once we exclude the possibility of achieving session guarantees by *stickiness* it is important to consider achieving them through caching and retaining state on the client side. However, retaining a large amount of state information on the client is not always possible or practical, and thus we consider *stateful* and *stateless* clients separately.

To sum up, we consider clients that can be either mobile or sticky, stateful or stateless. We also consider six different failure models, and for each we identify which consistency guarantees cannot be provided for a given type of clients. Additionally we specify new correctness conditions that can be achieved. We also show a set of phenomena observable by the clients that result from the highly-available nature of the environment. In particular, we identify a phenomenon called a *phantom operation*, which describes a successfully processed update, which however can be observed only temporarily by some clients. The other phenomena that we discuss include a split brain syndrome for clients who communicate outside the system as well as the lack of various client-session guarantees, such as eventually consistent variant of monotonic reads and read-your-writes that we formally define using our framework. On top of that, we define a novel client-session guarantee called *context preservation* that is incompatible with the classic session guarantees and is suited for highly-available systems utilizing CRDTs.

1.1. Contributions

We summarize our core contributions below:

- We define a framework that explicitly considers hardware (replica and network) failures, and thus enables formal reasoning about both the safety and liveness guarantees of highly available, eventually consistent systems in failure-prone environments. In particular, we admit replica recovery after crash considering various types of clients (stateless, stateful, sticky, and mobile).
- We define a family of *failure-aware* consistency criteria, e.g., based on the well-known *basic eventual consistency* [26], to adequately capture the behaviour of eventually consistent systems in the considered environment. We use our novel consistency criteria to systematize in the formal way the existing knowledge and intuitions regarding the correctness of highly available systems under failure conditions.
- We show specific liveness guarantees that cannot be provided when certain failures occur, such as *eventual visibility* of all events (see Section 4.6 for the definition). 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 the 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 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 *context preservation* (CP), a new session guarantee for systems that expose the concurrency to the client, e.g., through multi-value registers [62], observed-removed sets [63], etc. The CP guarantee is incomparable with the classic session guarantees [65].

[1] This approach is also consistent with how in practice clients' requests are routed to the replicas through stateless load balancers, thus giving no pairing guarantees between a client and a replica.

1.2. Article structure

The remainder of the paper is structured as follows. In Section 2, we illustrate our motivations with an example of a simple CRDT. Next, in Section 3, we outline the system model and then in Section 4 we specify our formal framework. In Section 5, we discuss the problem of maintaining state by clients and discuss session guarantees. In Section 6, we analyze the liveness guarantees that can be achieved under various combinations of possible replica and network failures. In this section, we also specify our new correctness criteria. In Section 7 we present the related work. Finally, we conclude in Section 8.

2. Motivation

In order to illustrate problems with formalizing guarantees of highly available systems, below we discuss example executions of a simple (state-based) CRDT,[2] in which each replica runs an implementation of a last-write-wins register (LWW-register, also called *epidemic register*) [24].

Shapiro et al. proposed a formal framework for proving correctness of CRDTs [62] and a target consistency guarantee called *strong eventual consistency* (SEC), which requires any two replicas that receive the same set of messages to be in the same state. Proving the correctness of a (state-based) CRDT involves showing that the CRDT's merge operation (also known as the *join* or *supremum*) satisfies the properties of a semi-lattice. However, doing so involves reasoning about internal replica states and message exchanges, which can be considered an implementation specific aspect of a CRDT. This is in contrast with well established and declarative approaches to formalizing consistency through conditions, such as sequential consistency [45], linearizability [37] or serializability for database systems [55]. These consistency conditions are defined over abstract system traces that represent the externally observable behaviour.

The formal framework proposed by Burckhardt [24] completely abstracts away from implementation-specific aspects of a replicated system as it solely relies on functions that represent data types, events that correspond to invocations of certain operations exported by the data type, and visibility and arbitration relations between the events. This way of reasoning about correctness is well suited to model infinite executions, which, as we argued earlier, must be considered when proving liveness guarantees. However, Burckhardt approach fails to properly account for replica or network failures as we demonstrate below.

Consider a simple system, in which each replica runs an implementation of a last-write-wins register (LWW-register, also called *epidemic register* [24]; see the pseudocode in Algorithm 1). 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 [66]. 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 [44] and R_i's identifier (line 12). Then R_i uses best-effort broadcast [27] 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) [41] 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 Vogels [70]: when updates cease all read() operations eventually return the same value. Formally one could show that the implementation ensures *basic eventual consistency* (BEC) [26] (for the formal definition of BEC see Section 4). However, as clearly follows from the executions in Fig. 1, when replica crashes (left execution) or network splits are possible (right example), the implementation no longer satisfies eventual consistency (even according to the simple Vogels' definition and 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* [27] 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 [12,42] 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 safety (the reads return some correct values written earlier), but in case when failures occur only the latter one satisfies liveness (convergence of the returned values).

Note that our example is based on a very simple CRDT. The implementations of more complex CRDTs, such as *replicated growable array* (RGA) [58] (which are notoriously difficult to reason about on their own [15,32]), are even more prone to the kinds of subtle bugs that we showcased. Moreover, in our example we discussed 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/sticky or stateless/stateful, are accounted for. Popular eventually consistent systems, e.g., [1–5], feature various anti-entropy mechanisms used to detect and repair discrepancies or inconsistencies in data, which may occur due to network splits or system failures. These mechanisms prevent anomalies depicted in Fig. 1. However, we are not aware of a general formal framework which could be used to reason about the correctness of systems which exhibit such subtle behaviour.[3] Therefore we wanted to fill this gap. In particular, the formal framework that we introduce in this article can be used to study the correctness of highly available systems and allows us to detect defects, such as the ones in Algorithm 1.

3. System model and failure models

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

[2] The other type of CRDTs are *operation-based*, where replicas exchange commutative updating operations and apply them locally in any order. In this work, for simplicity, we consider only state-based CRDTs.

[3] Interestingly, the sophisticated frameworks, such as [24,32,71], fail to correctly address the anomalies shown in Fig. 1, and the original definition of SEC by Shapiro et al. [62] does not admit them at all. However, the formulation by Shapiro et al. lacks in other areas as we discuss in detail in Section 7.

Algorithm 1 Naïve Replicated Register Protocol for replica R_i identified by rid.

```
1: struct RegRec(clk : int, rid : int, val : Val)
2: operator <(o : RegRec, o′ : RegRec)
3:     return (o.clk < o′.clk)
           ∨(o.clk = o′.clk ∧ o.rid < o′.rid)
4: var myReg : RegRec
5: operation write(val : Val)                    // write
6:     myReg = RegRec(myReg.clk + 1, rid, val)
7:     BE-cast myReg
8:     return ok
9: operation read()                              // read
10:    return myReg.val
11: upon BE-deliver(update : RegRec)
12:    if myReg < update then
13:        myReg = update
14:        // BE-cast update                      // protocol fix
```

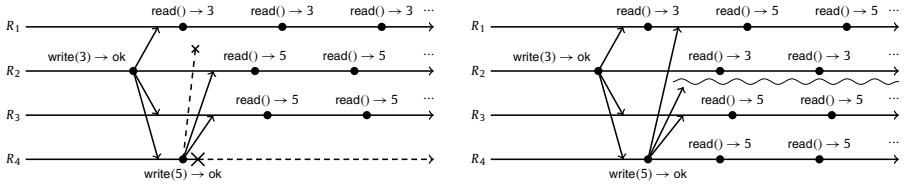


Fig. 1. Example executions of a system implementing Algorithm 1. Solid arrows originating in events (dots) represent sent messages. Left execution: eventual transmission of a message is guaranteed only for the three correct (never crashed) processes. A crash of R_4 leads to inconsistent states of replicas. Right 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 $\mathcal{R}$.

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.

3.1. Replicas

Replicas form a set $\mathcal{R} = \{R_1, R_2, ..., R_n\}$,[4] which can be divided into disjoint subsets $G_1, G_2, ...$, called *ensembles*. The ensembles represent sets of replicas located physically close to each other, e.g. in the same region, datacenter, or within the same availability zone (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.[5] 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, where the latter writes block the code execution on the replica until the written data is guaranteed to be persisted. Both kinds of writes can be interrupted by a crash. For simplicity we assume full replication of application data (each replica holds all data necessary to serve any particular request). We briefly discuss an extension to a partial replication setting in Appendix A.

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 repeated 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 ensemble is correct. Once a client request is received by a replica, the replica starts to execute it, and unless the replica subsequently crashes, the replica returns a response without waiting for any external events, such as messages from other replicas.

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.

3.2. 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.[6] 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 dependencies between requests issued by the same user. 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 lose state when the user's device is restarted without saving the state to stable storage, or when the user switches between devices during a session.

As discussed in Section 1, we elect to represent clients as external entities to better reflect the client-server architecture used in practice, and because of important consequences from the correctness point of view. Recall that causal consistency is achievable in highly available systems in a classic model, whereas it is not achievable with external clients [19]. It is so even when the clients are stateful and cache all their requests and responses. To achieve causal consistency clients would need to continuously 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 clients or even replicas. Thus, we believe it is essential to include the clients in our analysis of highly available systems as external entities.

[4] For simplicity we assume that the set is fixed, however reconfiguration could be easily supported.

[5] 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) [60].

[6] A client may also be used by another service. Then, the service is the user of the system.

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 ensemble 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. [18]), where a sticky client issues all requests always to exactly the same replica. We discuss this difference below.

3.3. Interactions between clients and replicas

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, as discussed in Section 1, in case of a fatal crash of a replica, the client would remain blocked, which violates high availability requirements.[7] This is one of the reasons why we exclude from our 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 each request independently and assigning the requests to replicas in round-robin fashion), or stateful (maintaining the information about client connections and routing the 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.[8] 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 [41], 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. We do not control on the client side which replica will serve the issued request.[9]

However, we do discern between mobile clients, which are completely unrestricted in terms of replicas they can connect to, 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.

3.4. 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 [27], 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 theorem [22,29] 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. Both approaches are 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 [13,30,52,67] (see also [18] 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. For example, *partial partitionings* [10,13] 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. Thus, temporary splits simply represent the regular mode of operation in an asynchronous network. On the other hand, permanent splits cannot be justified by asynchrony itself and represent actual failures of network hardware, such as link or switch failures, or even misconfiguration events. In practice, temporary splits may be caused by a sudden congestion spike causing hardware overload, or by a minor hardware failure which can be fixed automatically, e.g. by an automatic failover procedure. On the other hand, permanent network splits are major failure events, lasting in time, and requiring manual intervention of the system operators.

Moreover, temporary and permanent splits' impact is perceived differently by end users. Note that even though a highly available system is still required to respond to each request during a network split, these responses may not reflect previous operations, or otherwise violate business logic of the application. However, in case of temporary splits, these anomalies may be even unnoticed by end users, or simply cause the system to return correct responses with a small delay, e.g. prompting the end user to refresh a webpage a couple of times. On the other hand, permanent splits may render the system completely useless to the end user, and force the user to abandon the current activity (users' experience is compromised and as a result users decide to finish their sessions early).

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 [27], while the latter is similar to the model assumed in the CAP theorem, 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, $P_1, P_2, \ldots, P_k$, 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.

When considering sticky clients which always connect to a specific ensemble 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 ensemble boundaries. Thus, if we define ensembles 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

[7] Recall, however, that to satisfy high availability as defined in [22,24,29], only correct replicas need to eventually respond, and clients connected to a crashed replica may remain blocked infinitely.

[8] Note that if the load balancers use a replicated state machine [43] 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.

[9] 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.

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

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

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

clients).[10] Also, we assume that within each ensemble G_i there is at least a single correct replica reachable by external clients.

3.5. 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.

4. Formal framework

Below we provide the formal framework that allows us to reason about execution histories and correctness criteria. We extend the framework by Burckhardt et al. [24,26] (also used in several other works, e.g., [14,16,21,40,69]).

4.1. Preliminaries

A binary relation rel over set A is a subset $\text{rel} \subseteq A \times A$. By $\overline{\text{rel}}$, we denote the complement of the relation rel, i.e. $(A \times A) \setminus \text{rel}$. For $a, b \in A$, we use the notation $a \xrightarrow{\text{rel}} b$ to denote $(a, b) \in \text{rel}$, and the notation $\text{rel}(a)$ to denote $\{b \in A | a \xrightarrow{\text{rel}} b\}$. Thus, $\overline{\text{rel}}(a) = \{b \in A | a \not\xrightarrow{\text{rel}} b\}$. We use the notation rel^{-1} to denote the inverse relation, i.e. $(a \xrightarrow{\text{rel}^{-1}} b) \Leftrightarrow (b \xrightarrow{\text{rel}} a)$. Thus, $\text{rel}^{-1}(b) = \{a \in A | a \xrightarrow{\text{rel}} b\}$. Given two binary relations rel, rel′ over A, we define the composition $\text{rel}; \text{rel}' = \{(a, c) | \exists b \in A : a \xrightarrow{\text{rel}} b \xrightarrow{\text{rel}'} c\}$. We let id_A be the identity relation over A, i.e., $(a \xrightarrow{\text{id}_A} b) \Leftrightarrow (a \in A) \wedge (a = b)$. For $n \in \mathbb{N}_0$, we let rel^n be the n-ary composition rel; rel…; rel, with $\text{rel}^0 = \text{id}_A$. We let $\text{rel}^+ = \bigcup_{n \geq 1} \text{rel}^n$ and $\text{rel}^* = \bigcup_{n \geq 0} \text{rel}^n$. For some subset $A' \subseteq A$, we define the restricted relation $\text{rel}|_{A'} \stackrel{\text{def}}{=} \text{rel} \cap (A' \times A')$. In Fig. 2 we summarize various properties of relations.

If rel is an equivalence relation, we can use the notation $a \approx_{\text{rel}} b \stackrel{\text{def}}{\Longleftrightarrow} a \xrightarrow{\text{rel}} b$. An equivalence relation rel on A partitions A into equivalence classes $[x]_{\text{rel}} = \{y \in A | y \approx_{\text{rel}} x\}$. The classes are pairwise disjoint and cover A. We write $A/\approx_{\text{rel}}$ to denote the set of equivalence classes wrt. relation rel. For example, the parity relation *par* over the set of natural numbers $\mathcal{N}$, produces: $\mathcal{N}/\approx_{par} = \{[0]_{par}, [1]_{par}\}$, with $[0]_{par} = [2]_{par} = \ldots$, $7 \in [1]_{par}$, and $13 \approx_{par} 101$.

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

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

4.2. Histories

We represent a high-level view of a system execution as a *history*. We omit implementation details, such as message exchanges or internal execution steps. We include only the observable behaviour of the system (as perceived by the clients through received responses), and the information about failures. While the latter is not directly observable by clients,[11] we rely on this information to formalize the expected behaviour of the system in the presence of failures. Formally, a *history* is an event graph $H = (E, \text{op}, \text{rval}, \text{rb}, \text{so}, \text{sp}, \text{crash})$, where:

- $\text{op} : E \to \text{Operations}$, specifies the operation invoked in a particular event, e.g., $\text{op}(e) = \text{write}(3)$,
- $\text{rval} : E \to \text{Values} \cup \{\nabla\}$, specifies the value returned by the operation, e.g., $\text{rval}(e) = 3$, or $\text{rval}(e') = \nabla$, 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,
- $\text{crash} : E \to \{\text{true}, \text{false}\}$, specifies if a particular event was executed on a replica that subsequently crashed (true), or not (false).

Note that, for some event $e \in E$, $\text{crash}(e) = \text{true}$ 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 $a, b \in E$, $a \not\approx_{\text{sp}} b$ does not mean that replicas, which executed a and b, could not communicate with each other at the time these

[10] If network splits can occur, e.g., between datacenters and inside of a datacenter, but not within a rack of servers, ensembles 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, not sticky.

[11] For example, a timeout on a client's operation does not necessarily result from a replica failure.

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. $\mathsf{rval}(e) \neq \nabla$ means the response was generated by the replica, but this 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, $\mathsf{rval}(e) = \nabla$ 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 $\forall e \in E : (\neg\mathsf{crash}(e) \Rightarrow \mathsf{rval}(e) \neq \nabla)$ (in infinite executions replicas which do not crash eventually respond),
- $\forall a, b \in E : (a \xrightarrow{\mathsf{rb}} b \Rightarrow \mathsf{rval}(a) \neq \nabla)$ (a pending operation does not return),
- $\forall a, b, c, d \in E : (a \xrightarrow{\mathsf{rb}} b \wedge c \xrightarrow{\mathsf{rb}} d) \Rightarrow (a \xrightarrow{\mathsf{rb}} d \vee c \xrightarrow{\mathsf{rb}} b)$ (rb is an *interval order*, i.e. it is consistent with a timeline interpretation where operations correspond to segments [24,34]),
- $\mathsf{so} \subseteq \mathsf{rb}$ (session order respects the returns-before order),
- for all $e \in E$, the set $\mathsf{so}^{-1}(e)$ is well ordered by the relation so (so is a union of trees),
- $\forall a, b, c \in E : (a \xrightarrow{\mathsf{so}} b \wedge a \xrightarrow{\mathsf{so}} c) \Rightarrow (\mathsf{op}(b) = \mathsf{op}(c) \wedge (\mathsf{so}(b) = \emptyset \vee \mathsf{so}(c) = \emptyset))$ (there is only limited branching in so due to timeouts),
- $\forall a, b, c \in E : (a \xrightarrow{\mathsf{so}} b \wedge a \xrightarrow{\mathsf{so}} c \wedge \mathsf{so}(b) = \emptyset \wedge \mathsf{so}(c) \neq \emptyset) \Rightarrow (c \xrightarrow{\mathsf{rb}} b)$ (a client issues a request again only if it has abandoned the previous attempt),
- $|/\approx_{\mathsf{sp}}| < \infty$ (there is only a finite number of permanent network partitions).

4.3. Abstract executions

In order to *explain* the history, i.e., the observed return values, and reason about the system properties, we need to extend the history with information about the abstract relationships between events. For strongly consistent systems typically we do so by finding a *serialization* [45] (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 $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{so}, \mathsf{sp}, \mathsf{crash}, \mathsf{vis}, \mathsf{ar})$, such that $(E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{so}, \mathsf{sp}, \mathsf{crash})$ 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 $A = (H, \mathsf{vis}, \mathsf{ar})$ 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. Note that they are abstract notions and do not relate directly to the history's underlying execution.

Visibility describes the relative influence of operation executions in a history on each others' return values: if a is visible to b (denoted $a \xrightarrow{\mathsf{vis}} b$), then the effect of a is visible to the replica performing b (and thus reflected in the 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 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 $a \xrightarrow{\mathsf{ar}} b$), 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 the conflict resolution scheme used in the system, e.g., physical time-based timestamps, or logical clocks.

4.4. Correctness predicates

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

4.5. Replicated data type

We model the whole system as a single replicated object (as in case of Algorithm 1). 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*. We define the semantics of the replicated object through *replicated data types* [25]. Unlike classic sequential data types [37], replicated data types (defined formally below) can be used when system exposes concurrency to the client (see, e.g., multi-value register, MVR [62], or observed-remove set, OR-Set [63]).

In this approach, the state on which an operation $op \in \mathsf{Operations}$ 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 $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{so}, \mathsf{sp}, \mathsf{crash}, \mathsf{vis}, \mathsf{ar})$, the operation context of e in A is the event graph $\mathsf{context}(A, e) \overset{\text{def}}{=} (\mathsf{vis}^{-1}(e), \mathsf{op}, \mathsf{vis}, \mathsf{ar})$. Note that the context lacks return values, the returns-before relation, the session order, and the information about failures. The set of previously invoked operations and their relative visibility and arbitration unambiguously defines the output of each operation.

A *replicated data type* $\mathcal{F}$ is a function that, for each operation $op \in \mathsf{ops}(\mathcal{F})$ (where $\mathsf{ops}(\mathcal{F}) \subseteq \mathsf{Operations}$) and operation context C, defines the expected return value $v = \mathcal{F}(op, C) \in \mathsf{Values}$, 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'. Fig. 3 shows example replicated data types. We say that $op \in \mathsf{ops}(\mathcal{F})$ is a *read-only* operation (denoted $op \in \mathsf{readonlyops}(\mathcal{F})$), if and only if, for any operation op', context $C = (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})$ and event $e \in E$, such that $\mathsf{op}(e) = op$, $\mathcal{F}(op', C) = \mathcal{F}(op', C')$, where $C' = (E \setminus \{e\}, \mathsf{op}, \mathsf{vis}, \mathsf{ar})$. In other words, read-only operations can be excluded from any context C, producing C', and the result of

$$\mathcal{F}_{reg}(\mathsf{write}(v), (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = \mathsf{ok}$$
$$\mathcal{F}_{reg}(\mathsf{read}(), (E, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = \mathsf{if}\ \exists e \in E : (\mathsf{op}(e) = \mathsf{write}(v) \wedge \nexists e' \in E : (\mathsf{op}(e') = \mathsf{write}(v') \wedge e \xrightarrow{\mathsf{ar}} e'))\ \mathsf{then}\ v\ \mathsf{else}\ 0$$

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

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

Fig. 3. The replicated data type specifications for LWW-register, MVR, and OR-set.

any operation op' will not change. On the other hand, if an operation op is not read-only, we say that it is an *update* (denoted $op \in \mathsf{updateops}(\mathcal{F})$).

4.6. Basic eventual consistency

Now we introduce a baseline correctness criterion: *basic eventual consistency* (BEC) [26]. It consists of three simple requirements, which lie at the basis of modern eventually consistent systems, including the popular NoSQL data stores and CRDTs. Formally:

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

The first requirement is the *eventual visibility* (EV) of events. EV requires that for any not pending operation executed in an event $e \in E$, there is only a finite number of events in E that do not observe e. Additionally, a pending operation executed in an event e may become visible to other events, but then the same rules apply to it as if it was not pending. Formally: $\mathrm{EV} \stackrel{\mathrm{def}}{=} \forall e \in E : \big((\mathsf{rval}(e) \neq \nabla \vee \mathsf{vis}(e) \neq \emptyset) \Rightarrow |\overline{\mathsf{vis}}(e)| < \infty\big)$. Intuitively, EV implies progress in the system as replicas must synchronize and exchange knowledge about operations executed in the system.

The second requirement concerns avoiding circular causality. To formalize it, we introduce an auxiliary definition: the *happens-before* relation $\mathsf{hb} \stackrel{\mathrm{def}}{=} (\mathsf{so} \cup \mathsf{vis})^+$ (the transitive closure of session order and visibility). It allows us to express the causal dependency between events. Intuitively, if $e \xrightarrow{\mathsf{hb}} e'$, then e' potentially depends on e. We simply require *no circular causality*, $\mathrm{NCC} \stackrel{\mathrm{def}}{=} \mathsf{acyclic}(\mathsf{hb})$.

Finally, we specify *return value consistency* (RVAL), which requires that the return value of each non-pending operation can be explained using the specification of the replicated data type $\mathcal{F}$ and the operation's context: $\mathrm{RVAL}(\mathcal{F}) \stackrel{\mathrm{def}}{=} \forall e \in E : \big(\mathsf{rval}(e) \neq \nabla \Rightarrow \mathsf{rval}(e) = \mathcal{F}(\mathsf{op}(e), \mathsf{context}(A, e))\big)$.

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 Section 6, it cannot be satisfied when failures occur. Thus, it adequately captures the guarantees provided by eventually consistent systems only in the best case, i.e., in failure-free scenarios.

5. Client-side guarantees

Before we proceed with our analysis of highly available systems in face of failures, let us first discuss additional client-side guarantees called *session guarantees*. The four classic session guarantees [65] facilitate an intuitive and pragmatic programming model that builds on top of basic eventual consistency. System architects typically optimize their systems' designs for the *read your writes* (RYW) and *monotonic reads* (MR) session guarantees, as these are mostly anticipated by the users [20]. 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. They can be expressed in our model as: $\mathrm{RYW} \stackrel{\mathrm{def}}{=} \mathsf{so} \subseteq \mathsf{vis}$ and $\mathrm{MR} \stackrel{\mathrm{def}}{=} (\mathsf{vis}; \mathsf{so}) \subseteq \mathsf{vis}$.

Unfortunately, as we have discussed in Section 3 (see also Section 7), 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. For example, MR is usually described as important in the context of a webmail client: whenever a user has seen an email in her 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, then it *eventually* becomes visible, which is exactly the guarantee that BEC provides. In Section 6.1 we provide variants of RYW and MR, called *eventual session guarantees*, which are ensured only eventually.

5.1. 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). The specification of an OR-set ($\mathcal{F}_{orset}$) is in Fig. 3. An OR-set functions like a regular set but binds the remove() operations with the read() operations, in a way that restricts a client's ability to remove elements from the set only to the elements which the client has observed. This way the client cannot accidentally remove elements that were concurrently added by other clients. Assume that a client received x in 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 OR-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 (add(x) becomes visible to remove(x)).

This problem is even more evident in case of a system that implements a multi-value register (MVR), so systems, such as Dynamo [28] and Riak [3]). The specification of an MVR ($\mathcal{F}_{MVR}$) is in Fig. 3. 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 by executing the 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 that an MVR can be used as intended. Such a session guarantee requires some metadata to be kept in the client state.[12]

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 $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{so}, \mathsf{sp}, \mathsf{crash}, \mathsf{vis}, \mathsf{ar})$ in which two clients concurrently issue operations regarding registers x and y. The first client issues a following chain of operations: $\mathsf{read}(x) \rightarrow \{u\}$, $\mathsf{read}(y) \rightarrow \{v\}$, $\mathsf{write}(x, u')$, in events e_1, e_2 and e_3, respectively (which means, e.g., that $\mathsf{op}(e_1) = \mathsf{read}(x)$ and $\mathsf{rval}(e_1) = \{u\}$). The second client issues: $\mathsf{read}(x) \rightarrow \{u\}$, $\mathsf{write}(x, u'')$, 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 $e_5 \xrightarrow{\mathsf{vis}} e_2$. 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, $e_5 \xrightarrow{\mathsf{vis}} e_3$ 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 erroneously overwritten, and if insufficient writes are visible, unnecessary siblings will be created. Assuming some abstract execution $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{so}, \mathsf{sp}, \mathsf{crash}, \mathsf{vis}, \mathsf{ar})$, we can express this requirement through a new predicate that we call *context preservation* (CP):

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

$\mathrm{CP}(\mathcal{F}_{MVR})$ 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 LWW-register (see the specification of $\mathcal{F}_{reg}$ in Fig. 3, and the implementation in Algorithm 1) all operations are independent and concurrency is never exposed to the client. Thus, an LWW-register can work correctly with stateless clients.

[12] Such metadata can be efficiently maintained by using, e.g., dotted version vectors [11,57], as in Riak.

6. Correctness in the face of failures

In this section we present a formal analysis of the behaviour of highly available replicated systems in the presence of failures. In our analysis we consider NC-PNP, CS-TNP, CS-PNP, CR-TNP, and CR-PNP models. We omit the NC-TNP model, because, as we discuss later, for highly available, eventually consistent systems it is equivalent to a failure-free model. In Section 4.6 we have shown 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 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 $A = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{so}, \mathsf{sp}, \mathsf{crash}, \mathsf{vis}, \mathsf{ar})$: $\exists op_r \in \mathsf{readonlyops}(\mathcal{F}), op_u \in \mathsf{updateops}(\mathcal{F}), e \in E : (\mathsf{op}(e) = op_u \wedge \mathcal{F}(op_r, (\emptyset, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) \neq \mathcal{F}(op_r, (\{e\}, \mathsf{op}, \mathsf{vis}, \mathsf{ar})))$. All practical replicated data types (including the ones in Fig. 3) are non-trivial.

6.1. Network splits and state convergence

We begin with the simple case of permanent network splits and assume that every replica is correct and no crashes occur (the NC-PNP model). Recall the example from Section 2. Clearly, the implementation of a register provided in Algorithm 1 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 1 does not satisfy EV and, in consequence, also $\mathrm{BEC}(\mathcal{F}_{reg})$. 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 1. *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* $\mathrm{BEC}(\mathcal{F})$.

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 $H = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{so}, \mathsf{sp}, \mathsf{crash})$. If the system satisfies $\mathrm{BEC}(\mathcal{F})$, then there exists an abstract execution $A = (H, \mathsf{vis}, \mathsf{ar})$, that satisfies $\mathrm{BEC}(\mathcal{F})$.

There is a single event $e_0 \in E$, such that $\mathsf{op}(e_0) = op_u$ and infinitely many events $e_i \in E$, with $i \geq 1$, such that $\mathsf{op}(e_i) = op_r$. For all $e_i, e_j \in E$, $i < j \Leftrightarrow e_i \xrightarrow{\mathsf{rb}} e_j$. 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 $i \geq k$, $e_0 \xrightarrow{\mathsf{vis}} e_i$. Then, for each $e_i \in (E \setminus \mathsf{vis}(e_0))$, $\mathsf{rval}(e_i) = \mathcal{F}(op_r, \mathsf{context}(A, e_i)) = \mathcal{F}(op_r, (\emptyset, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = v$, and for each $e_j \in \mathsf{vis}(e_0)$, $\mathsf{rval}(e_j) =$

$\mathcal{F}(op_r, \mathsf{context}(A, e_j)) = \mathcal{F}(op_r, (\{e_0\}, \mathsf{op}, \mathsf{vis}, \mathsf{ar})) = v'$, 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 $H' = (E', \mathsf{op}', \mathsf{rval}', \mathsf{rb}', \mathsf{so}', \mathsf{sp}', \mathsf{crash}')$, 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 $e'' \in E''$, $\mathsf{rval}'(e'') = \mathsf{rval}(e'')$. Such a history H' must exist, because the replicas in H were separated by a network split, and the events on R_2 occurred 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 $A' = (H', \mathsf{vis}', \mathsf{ar}')$, that satisfies BEC($\mathcal{F}$). By RVAL($\mathcal{F}$) and properties of read-only operations, for each $e' \in E'$, $\mathsf{rval}(e') = \mathcal{F}(op_r, \mathsf{context}(A', e')) = \mathcal{F}(op_r, (\emptyset, \mathsf{op}', \mathsf{vis}', \mathsf{ar}')) = v$. But clearly, for some $e'' \in E'' \subset E'$, $\mathsf{rval}(e'') = v'$, such that $v' \neq v$. A contradiction. □

The above result is clearly related to the CAP theorem [22,29], 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 [29] uses linearizability [37] as the consistency criterion. On the other hand, our result concerns the impossibility of satisfying Burckhardt's BEC, a much weaker correctness criterion, and thus can be viewed as a strengthening of the CAP theorem. The reason for this counter-intuitive result lies in the fact that machine and network failures are not reflected by the definition of BEC itself. Additionally, 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 [24] implicitly assuming only temporary network splits (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 [45]. However, he shows that for other data types, under these assumptions, it is impossible to satisfy sequential consistency. The above highlights just how important it is to clearly state all the assumptions. Since temporary network splits do not constitute substantial obstacles for highly available, eventually consistent systems (as Burckhardt shows for $\mathcal{F}_{reg}$, and as we later show in general with our Theorem 8), 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 1 is the impossibility to satisfy EV. Hence, we cannot expect the replica states to ever converge, as shown in Fig. 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 1 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 splits, but which directly prohibits the unnecessary phenomena present in Algorithm 1 without the fix. We first formalize the weakened version of EV: *partition-aware* EV (PAEV):

$$\mathrm{PAEV} \stackrel{\mathrm{def}}{=} \forall e \in E : ((\mathsf{rval}(e) \neq \nabla \Rightarrow |[e]_{\mathsf{sp}} \cap \overline{\mathsf{vis}}(e)| < \infty) \wedge \forall p \in E/\approx_{\mathsf{sp}} : (p \cap \mathsf{vis}(e) \neq \emptyset \Rightarrow |p \cap \overline{\mathsf{vis}}(e)| < \infty))$$

PAEV states that for any event e, (1) if e is not pending, 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:

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

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 (e.g., Algorithm 1 with and without the fix).

Note that network splits lead to a phenomenon called *split brain syndrome*, in which mobile clients that switch between replicas from different 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 discuss the eventual variants of key session guarantees: *eventually read your writes* (ERYW) and *eventually monotonic reads* (EMR). Below we state them formally:

$$\mathrm{ERYW} \stackrel{\mathrm{def}}{=} \forall e \in E : |\mathsf{so}(e) \cap \overline{\mathsf{vis}}^{-1}(e)| < \infty$$
$$\mathrm{EMR} \stackrel{\mathrm{def}}{=} \forall e \in E\ \forall e' \in \mathsf{vis}^{-1}(e) : |\mathsf{so}(e) \cap \overline{\mathsf{vis}}(e')| < \infty$$

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 ($e'' \in \mathsf{so}(e)$) 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 2. *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* PABEC($\mathcal{F}$) ∧ ERYW *or* PABEC($\mathcal{F}$) ∧ EMR *for stateless mobile clients.*

The proofs involving session guarantees can be found in Appendix B.

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 (see Sections 7 and 3). Note also, that context preservation CP (see Section 5.1) can also be achieved in the PNP model, but using less resources than in case of RYW or MR.

6.2. 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:

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

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:

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

Some acknowledged updates that were not propagated due to crashes are benign. For example, 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. Still, in principle we cannot avoid phantom operations if crashes occur in the CS model, as we state formally below:

Theorem 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* $\mathrm{XBEC}(\mathcal{F})$.

Proof. The proof is similar to the one for Theorem 1. Consider a history H as outlined in the proof of Theorem 1, 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 1, 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. □

Since BEC and PABEC clearly imply XBEC, it follows from the Theorem 3 that they also cannot be satisfied by a highly available system in the face of failures. Hence 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 CAEV and FAEV predicates, respectively:

$$\mathrm{CAEV} \stackrel{\text{def}}{=} \forall e \in E : (|\overline{\mathsf{vis}}(e)| < \infty \vee (\mathit{crash}(e) \wedge (\mathit{crash}^{-1}(\mathsf{false}) \cap \mathsf{vis}(e) = \emptyset)))$$
$$\mathrm{FAEV} \stackrel{\text{def}}{=} \forall e \in E\ \forall p \in E/\approx_{\mathsf{sp}} : (|p \cap \overline{\mathsf{vis}}(e)| < \infty \vee (p \cap \mathit{crash}^{-1}(\mathsf{false}) \cap (\mathsf{vis}(e) \cup \{e\}) = \emptyset))$$

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 p, either e is eventually visible in that partition, or no event $e' \in p$ that occurred in that partition on a replica 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:

$$\mathrm{CABEC}(\mathcal{F}) \stackrel{\text{def}}{=} \mathrm{CAEV} \wedge \mathrm{NCC} \wedge \mathrm{RVAL}(\mathcal{F})$$
$$\mathrm{FABEC}(\mathcal{F}) \stackrel{\text{def}}{=} \mathrm{FAEV} \wedge \mathrm{NCC} \wedge \mathrm{RVAL}(\mathcal{F})$$

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 4. *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* $\mathrm{CABEC}(\mathcal{F}) \wedge \mathrm{ERYW}$ *or* $\mathrm{CABEC}(\mathcal{F}) \wedge \mathrm{EMR}$ *for stateless clients.*

6.3. Replica recovery and stable storage

Theorem 3 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. 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 (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 5. *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* $\mathrm{XBEC}(\mathcal{F})$.

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: (1) a crash after which the replica does not recover, and (2) infinitely many crashes and recoveries of

the same replica. In the former case, the same reasoning applies as in Theorem 3 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 6. *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.*

Theorem 7. *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 3 (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 3 to reach a contradiction. □

Note that Theorem 7 *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., [6]). 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 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) [60] which promises performance that is almost on par with RAM [56], or upcoming storage class memory devices which are based on Compute Express Link (CXL) [7,8] .

We make one final observation for a system that uses stable storage to avoid some phantoms (the proof is available in Appendix B):

Theorem 8. *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.*

6.4. Summary

In Fig. 4 we summarize the consistency guarantees which are possible to achieve in highly available systems and the artefacts a client can observe in various combinations of replica and network failure models. In terms of the offered guarantees and the types of artefacts 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 artefacts as in the crash-stop (CS) model.

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

7. Related work

7.1. Formalization of high availability and eventual consistency

(High) availability was first defined as a formal guarantee for replicated systems by Brewer's CAP conjecture [22]. It stipulates that eventually, for every request, a response needs to be provided. Later, Gilbert and Lynch conducted a formal proof of the conjecture, rendering CAP a theorem [29]. In the latter work, high availability was equated to *wait-freedom* [36]. More recently (see, e.g., [14,16,24]) 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 and formally relate our work to the CAP theorem in Section 6.1.

As we mentioned earlier, the majority of the existing work on the correctness of highly available systems either abstract away from machine failures altogether [14,16,21,31,33,35,46,47,61,71], admit machine failures or network splits but in the correctness proofs consider only system runs in which no failures occur [24,26], or fail to consider infinite executions and liveness properties [32]. Below we discuss the most relevant bodies of work we are aware of.

As follows from the CAP theorem, highly available systems can only guarantee some form of *eventual consistency*. Early definitions of eventual consistency [17,70] 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 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, as in case of the target correctness condition for CRDTs, which we discuss below.

As we already briefly discussed in Section 2, highly available systems often feature *Conflict-free Replicated Data Types* (CRDTs) for enhanced semantics. Proposed by Shapiro et al. [62,63], CRDTs are specialized data structures, which can be implemented solely in an asynchronous manner and by design ensure eventual convergence of replica states. The authors assume that every step a CRDT algorithm performs is synchronously logged to stable storage. This is a substantial simplification, which does not reflect the way CRDTs are usually implemented (as lightweight, in-memory data structures). Our approach is more comprehensive, since e.g., we study when the writes to stable storage are necessary. Shapiro

			no-crash	crash-stop	crash-recovery	
					only transient failures	all failures
consistency guarantees	temporary network splits		BEC	CABEC	BEC	CABEC
	permanent network splits		PABEC	FABEC	PABEC	FABEC
artefacts that can be encountered by the clients	temporary network splits	(all clients)		○ ◉ ≴		○ ◉ ≴
	permanent network splits	(all clients)	◐ ◉ ≴	○ ◉ ≴ ◐	◐ ◉ ≴	○ ◉ ≴ ◐
		(mobile clients)				

○ – phantom operations, ◉ – no ERYW (eventual read my writes) for stateless clients
≴ – no EMR (eventual monotonic reads) for stateless clients, ◐ – split brain syndrome for clients who communicate outside the system

Fig. 4. The consistency guarantees and phenomena observable by the clients in various failure models for a highly available system that implements an arbitrary, non-trivial replicated data type $\mathcal{F}$.

et al. proposed *strong eventual consistency* (SEC) [62,63] 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 1 with line 10 replaced by `return 0`). In our approach, we avoid this problem by utilizing the replicated data type specifications [25] to bind the allowed responses with the history of previously executed operations. Moreover, unlike in the case of SEC, which is defined solely on replica states, we consider external clients and the guarantees provided to them, which is a more sound 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.

The use of replicated data type specification also lays at the core of *basic eventual consistency* (BEC), which was proposed by Burckhardt et al. [24,26]. BEC abstracts away from implementation details such as internal replica states or exchanged messages. We closely follow this work as we base our family of failure-aware correctness criteria on BEC. In [24] 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.

Wang et al. [71] try to improve upon BEC with a framework that replaces replicated data type specifications with (abstract-state-based) sequential specifications. This comes, however, at a cost of label rewriting in the history for types that cannot be normally explained by sequential semantics. This rewriting, which needs to be done manually, amounts to exposing implementation details concerning the metadata needed to implement the replicated data type given, e.g., an OR-set which we discuss in Section 5.1.

Early eventually consistent systems, which had clients colocated with replicas, featured additional client side guarantees called *session guarantees* [64], such as *read-your-writes* (RYW) or *monotonic reads* (MR). When clients are external to replicas, 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 (but not all of) session guarantees can be provided when the clients cache their writes and reads [19]. In turn, significant 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 our work we show that enforcing classic session guarantees can be actually counterproductive for certain replicated data types. Hence, we provide a novel substitute called *context preservation*. We also define *eventual session guarantees*, which can be obtained with stateless clients in certain failure modes without compromising on high availability. The four classic session guarantees combined with eventual consistency form causal consistency [23,24].

Mahajan et al. [50,51] discuss the properties of highly available systems that satisfy a form of causal consistency called *natural causal consistency*, in which the causal order of operations needs to respect additional real-time constraints. In their work they employ the crash-stop and Byzantine failure models, however, machine recovery after crash is not considered. They introduced liveness property of *convergence*, which is state-based and relies on explicit message exchanges, and so is similar to SEC. Mahajan et al. also do not consider external clients.

Bailis et al. [18] focus on high availability in the context of transaction processing systems. In terms of system 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 that strive to gracefully tolerate failures.

7.2. Mechanized proof techniques

Gomes et al. [32] apply mechanized proofs based on Isabelle [72] to study the correctness of CRDTs. As the authors note, mechanized proofs can fail (and did so in the past), when the assumptions about the environment are wrong. We concur with them, and in our work we put strong emphasis on modelling the networked environment of CRDTs realistically and correctly. Compared to Gomes et al., our work is more fundamental: instead of proving correctness of individual CRDTs, we are more interested in general possibility/impossibility results and axiomatic definition of correctness guarantees. Moreover, Gomes et al. consider only finite histories, and thus their analysis do not include liveness properties. As a result, their correctness proofs cannot detect anomalies discussed in Section 2, similarly to the proofs by Burckhardt which do not account for failures at all. Additionally, Gomes et al. do not consider recovery after crash, and assume causal broadcast as the weakest group communication primitive, which is too strong of a requirement for many systems.

Liu et al. [49] target semi-automatic verification (using SMT solving) of CRDTs. They utilize Liquid Haskell [59,68], and focus on proving strong convergence of CRDTs, which is a safety property. They ignore liveness properties such as eventual delivery, and abstract away from crashes and network failures altogether. However, they improve upon the work of Gomes et al. in some regard by dropping the unnecessary assumption of causal broadcast as the base communication primitive.

Imine et al. [38] propose using SPIKE, an automated theorem prover, to assist development of transformation functions for *Op-*

erational Transformation (OT) solutions (centralized precursors to CRDTs). They do not consider machine and network failures. Notably, the mechanized proofs by Imine et al. were later proved to be incorrect by Oster et al. [54]. The proofs failed due to false assumptions about the execution environments as noted by Gomes et al. in [32].

7.3. Highly available system implementations

Relaxed consistency models in database management systems emerged with the rapid evolution of the Internet and, in consequence, the demand for scalable and highly available services. In this regard the design of Amazon Dynamo [28] 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 [41], Scylla [4], Riak [3], Voldemort [5], Netflix's Dynomite [53]) are massively scalable, can gracefully tolerate machine and network failures and still provide low latency responses. The latter trait stems from 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 without synchronous communication with other replicas. In the context of the *PACELC* framework [9], these systems choose low-latency over consistency even when no network splits occur. We base our analysis on this very assumption.

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.

8. Conclusions

In this paper we discussed various aspects of correctness of real-life highly available, eventually consistent replicated systems, that work in an environment in which machine failures and network splits are likely to occur. As we have shown, not taking failures into account may lead to misconceptions regarding even the basic requirements of highly available systems, such as replica state convergence (within each network partition). Moreover, to prove correctness of systems that feature more complex semantics (e.g., systems that utilize MVRs or OR-sets), formal reasoning about the state maintained by external clients is necessary. We believe that our work remedies some long overlooked aspects of the theory of eventually consistent systems and thus may be of value especially to researchers interested in reasoning about correctness of eventually consistent systems.

CRediT authorship contribution statement

M. Kokociński: Conception and design of study, Drafting the manuscript, Revising the manuscript critically for important intellectual content, Approval of the version of the manuscript to be published. **T. Kobus:** Conception and design of study, Drafting the manuscript, Revising the manuscript critically for important intellectual content, Approval of the version of the manuscript to be published. **P.T. Wojciechowski:** Conception and design of study, Revising the manuscript critically for important intellectual content, Approval of the version of the manuscript to be published.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

The authors would like to thank the reviewers for their valuable comments which helped to improve the manuscript.

This work was supported by the Foundation for Polish Science, within the TEAM programme co-financed by the European Union under the European Regional Development Fund (grant No. POIR.04.04.00-00-5C5B/17-00).

Appendix A. Extension to partial replication

Let us discuss a possible extension of the system model from Section 3 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* [28,39]. However, care needs to be taken with partial replication, because in each ensemble 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 number of faulty replicas allowed.

Appendix B. Additional proofs

Theorem 2. *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* PABEC($\mathcal{F}$) $\wedge$ ERYW *or* PABEC($\mathcal{F}$) $\wedge$ EMR *for stateless mobile clients.*

Proof. The proof is similar to the one for Theorem 1. Consider a history H as outlined in the proof of Theorem 1, 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 $e_i, e_j \in E$, $i < j \Leftrightarrow e_i \xrightarrow{\text{so}} e_j$. Note that $\text{so}(e_0) = E \setminus \{e_0\}$. If the system satisfies PABEC($\mathcal{F}$) with either of the two session guarantees, then there exists an abstract execution $A = (H, \text{vis}, \text{ar})$, that satisfies PABEC($\mathcal{F}$) with the respective session guarantee.

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

If $A \models$ ERYW, then there exists some $k \geq 1$, such that for each $i \geq k$, $e_0 \xrightarrow{\text{vis}} e_i$ (e_0 has to be eventually visible, as the set $\text{so}(e_0) \cap \overline{\text{vis}}^{-1}(e_0)$ has to be finite, and all operations are issued within the same session).

If $A \models$ EMR, then there exists some $k \geq k'$, such that for each $i \geq k$, $e_0 \xrightarrow{\text{vis}} e_i$ (since e_0 is observed by some $e_{i'}$, where $i' \geq k' \wedge i' \equiv$

0 (mod 2), e_0 can be not observed only by a finite number of events $e_i \in \mathsf{so}(e_{i'})$).

Now, we can conclude (similarly to the way we did in the proof of Theorem 1) that there is infinitely many events e_j, such that $e_0 \xrightarrow{\text{vis}} e_j$, and that their return value v' is different than v (the return value for e_i such that $e_0 \not\xrightarrow{\text{vis}} e_i$).

When we consider an alternative history H', where op_u has not been invoked, we can see that it is indistinguishable from H for R_2: the stateless client issues the same operations, passing exactly the same information, and no messages from 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. □

Theorem 4. *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($\mathcal{F}$) ∧ ERYW *or* CABEC($\mathcal{F}$) ∧ EMR *for stateless clients.*

Proof. The proof is based on the same principles as the proofs for Theorems 1, 2 and 3, but is more complex. We consider a history H as in the proof of Theorem 1, but with $c_1 = c_2$, and R_1 crashing after serving the response for op_u. However, contrary to the proof of Theorem 3, 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 operations op_r 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 $H' = (E', \mathsf{op}', \mathsf{rval}', \mathsf{rb}', \mathsf{so}', \mathsf{sp}', \mathsf{crash}')$, 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 1. Note that R_1 does not crash in H', so e_0 cannot be a phantom operation. Since $crash(e_i) = \mathsf{false}$ 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, $H' \models$ BEC($\mathcal{F}$). Then, following the logic from the proof of Theorem 1, 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 a (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 2, 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 1, 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. □

Theorem 6. *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.*

Algorithm 2 Protocol implementing replicated data type $\mathcal{F}$, for replica R_i identified by rid.

```
1: struct OpRec(clk : integer, rid : integer, op : ops(F))
2: operator <(o : OpRec, o' : OpRec)
3:     return (o.clk < o'.clk) ∨ (o.clk = o'.clk ∧ 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: operation Perform(op : ops(F))
12:     var rval = F(op, makeContext(operations, visible))
13:     var o = OpRec(max(operations).clk + 1, rid, 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
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
```

Proof. Just as in Theorem 5 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 4 for the CS model. □

Theorem 8. *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 2, which shows a pseudocode of a generic protocol for an arbitrary $\mathcal{F}$. Note that Algorithm 2 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 a quanta of information concerning the 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 clk (the value of a logical clock maintained by the replica at the time of the invocation of op). OpRec structures can be totally ordered using the clk 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.clk is chosen so that it is larger than the clk 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 $H = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{so}, \mathsf{sp}, \mathsf{crash})$) of a system that implements Algorithm 2, we need to show that there exists an abstract execution $A = (H, \mathsf{vis}, \mathsf{ar})$, such that $A \models \mathrm{FABEC}(\mathcal{F})$. It is because if there are no fatal failures, FABEC reduces to BEC in the TNP model, and to 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 2 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 $\mathsf{pre}(e)$ and $\mathsf{post}(e)$ the volatile state of the replica, respectively, just before the execution of e, and just after the execution of e. Similarly, by $\mathsf{sspre}(e)$ and $\mathsf{sspost}(e)$ we denote the contents of stable storage of the replica executing e, before and after the execution of e. Note that $\mathsf{pre}(e) = \mathsf{sspre}(e)$, but not always $\mathsf{post}(e) = \mathsf{sspost}(e)$ (e.g., when the replica executing e crashes before completing the execution). Moreover, $\mathsf{sspre}(e).\mathsf{operations} \subseteq \mathsf{sspost}(e).\mathsf{operations}$ and $\mathsf{sspre}(e).\mathsf{visible} \subseteq \mathsf{sspost}(e).\mathsf{visible}$. Similarly, for any two events $e, e' \in E$ executed on the same replica in that order, $\mathsf{sspre}(e').\mathsf{operations} \subseteq \mathsf{sspost}(e).\mathsf{operations}$ and $\mathsf{sspre}(e').\mathsf{visible} \subseteq \mathsf{sspost}(e).\mathsf{visible}$.

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

Let us make an observation. For an event e executed on some replica R_i, if $\mathsf{rval}(e) \neq \nabla$ it means that $e \in \mathsf{sspost}(e).\mathsf{operations}$. Similarly, if $\mathsf{vis}(e) \neq \emptyset$, and thus there exists $e' \in E$, such that $e \xrightarrow{\mathsf{vis}} e'$, 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 $e \in \mathsf{sspost}(e).\mathsf{operations}$.

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

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, $a \xrightarrow{\mathsf{rb}} b$. 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 $a \in \mathsf{pre}(b).\mathsf{operations}$. 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, $\mathsf{vis} \subseteq \mathsf{rb}$. Additionally, $\mathsf{so} \subseteq \mathsf{rb}$, by well-formedness of a history. Therefore, $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^+ \subseteq \mathsf{rb}$, and since rb is a partial order, hb is acyclic, and $A \models \mathrm{NCC}$.

Finally, we turn our attention to $\mathrm{RVAL}(\mathcal{F})$. The return value for each not pending event $e \in E$ is computed using the function $\mathcal{F}$ itself. We need to show that the output $C' = (E', \mathsf{op}', \mathsf{vis}', \mathsf{ar}')$ of the makeContext function is isomorphic with $C'' = \mathsf{context}(A, e) = (E'', \mathsf{op}, \mathsf{vis}, \mathsf{ar})$. Firstly, by definition $\mathsf{vis}^{-1}(e) = \mathsf{pre}(e).\mathsf{operations}$, thus $E' = E''$. Secondly, for each $e' \in E'$, $\mathsf{opFun}(e') = e'.\mathit{op}$, and $e'.\mathit{op} = \mathsf{op}(e')$. Thus, for each $e' \in E'$, $\mathsf{op}'(e) = \mathsf{op}(e)$. Thirdly, for any three events $a, b, c \in E$, if $a \in \mathsf{pre}(b).\mathsf{operations} \wedge a, b \in \mathsf{pre}(c).\mathsf{operations}$, then $(a, b) \in \mathsf{pre}(c).\mathsf{visible}$, because the sets operations and visible are always modified, persisted and disseminated together atomically. Thus, for any two $a, b \in E'$, such that $a \xrightarrow{\mathsf{vis}} b$, $(a, b) \in \mathsf{pre}(e).\mathsf{visible}$, which means that $\mathsf{vis}' = \mathsf{vis}|_{E'}$. Fourthly, $(a, b) \in \mathsf{ar}' \Leftrightarrow a, b \in E' \wedge a < b$, and $(a, b) \in \mathsf{ar} \Leftrightarrow a, b \in E \wedge a < b$. Thus, $\mathsf{ar}' = \mathsf{ar}|_{E'}$. Finally, since $E' = E''$, while op′, vis′ and ar′ are restrictions of op, vis and ar to E', we have that C' and C'' are isomorphic, and so $A \models \mathrm{RVAL}(\mathcal{F})$.

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

References

[1] Apache Cassandra documentation, Apache cassandra, https://cassandra.apache.org/.
[2] Redis documentation, Redis, https://redis.io/.
[3] Basho documentation, Riak key-value store, http://basho.com/products/riak-overview/.
[4] ScyllaDB documentation, Scylla, https://www.scylladb.com/.
[5] Project Voldemort, Voldemort key-value store, https://www.project-voldemort.com/.
[6] Google documentation, Google cloud – storage options, https://cloud.google.com/compute/docs/disks/.
[7] tom's Hardware, Intel Kills Optane Memory Business, Pays $559 Million Inventory Write-Off, https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good, 2022.
[8] Business Wire, Kioxia Launches Second Generation of High-Performance, Cost-Effective XL-FLASH Storage Class Memory Solution, https://www.businesswire.com/news/home/20220801005862/en/Kioxia-Launches-Second-Generation-of-High-Performance-Cost-Effective-XL-FLASH%E2%84%A2-Storage-Class-Memory-Solution/, 2022.
[9] D. Abadi, Consistency tradeoffs in modern distributed database system design: CAP is only part of the story, Computer 45 (2) (Feb. 2012).
[10] M. Alfatafta, An analysis of partial network partitioning failures in modern distributed systems, Master's thesis, University of Waterloo, 2020.

[11] P.S. Almeida, C. Baquero, R. Gonçalves, N. Preguiça, V. Fonte, Scalable and accurate causality tracking for eventually consistent stores, in: Proc. of DAIS '14, 2014.
[12] B. Alpern, F.B. Schneider, Recognizing safety and liveness, Distrib. Comput. 2 (3) (1987) 117–126.
[13] A. Alquraan, H. Takruri, M. Alfatafta, S. Al-Kiswany, An analysis of network-partitioning failures in cloud systems, in: Proc. of OSDI '18, 2018.
[14] H. Attiya, F. Ellen, A. Morrison, Limitations of highly-available eventually-consistent data stores, in: Proc. of PODC '15, 2015.
[15] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, M. Zawirski, Specification and complexity of collaborative text editing, in: Proc. of PODC '16, 2016, pp. 259–268.
[16] H. Attiya, F. Ellen, A. Morrison, Limitations of highly-available eventually-consistent data stores, IEEE Trans. Parallel Distrib. Syst. 28 (1) (2017) 141–155.
[17] P. Bailis, A. Ghodsi, Eventual consistency today: limitations, extensions, and beyond, Queue 11 (3) (2013) 20–32.
[18] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J.M. Hellerstein, I. Stoica, Highly available transactions: virtues and limitations, Proc. VLDB Endow. 7 (3) (2013) 181–192.
[19] P.A. Bernstein, S. Das, Rethinking eventual consistency, in: Proc. of SIGMOD '13, 2013.
[20] J. Bonér, Scalability, availability & stability patterns, https://www.slideshare.net/jboner/scalability-availability-stability-patterns/, 2010.
[21] A. Bouajjani, C. Enea, J. Hamza, Verifying eventual consistency of optimistic replication systems, in: Proc. of POPL '14, 2014.
[22] E.A. Brewer, Towards robust distributed systems (abstract), in: Proc. of PODC '00, 2000.
[23] J. Brzezinski, C. Sobaniec, D. Wawrzyniak, From session causality to causal consistency, in: Proc. of PDP '04: the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004, pp. 152–158.
[24] S. Burckhardt, Principles of eventual consistency, Found. Trends Program. Lang. 1 (1–2) (2014) 1–150.
[25] S. Burckhardt, A. Gotsman, H. Yang, M. Zawirski, Replicated data types: specification, verification, optimality, in: Proc. of POPL '14, 2014.
[26] S. Burckhardt, A. Gotsman, H. Yang, Understanding eventual consistency, Tech. Rep. MSR-TR-2013-39, Microsoft Research, Mar. 2013.
[27] C. Cachin, R. Guerraoui, L. Rodrigues, Introduction to Reliable and Secure Distributed Programming, Springer, 2011.
[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, Oper. Syst. Rev. 41 (6) (2007) 205–220.
[29] S. Gilbert, N. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, SIGACT News 33 (2) (2002) 51–59.
[30] P. Gill, N. Jain, N. Nagappan, Understanding network failures in data centers: measurement, analysis, and implications, in: Proc. of the SIGCOMM '11, 2011, pp. 350–361.
[31] A. Girault, G. Gößler, R. Guerraoui, J. Hamza, D. Seredinschi, Monotonic prefix consistency in distributed systems, in: Proc. of FORTE '18, 2018.
[32] V.B. Gomes, M. Kleppmann, D.P. Mulligan, A.R. Beresford, Verifying strong eventual consistency in distributed systems, in: Proc. of OOPSLA '17 1 (2017) 1–28.
[33] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, M. Shapiro, 'Cause I'm strong enough: reasoning about consistency choices in distributed systems, in: Proc. of POPL '16, 2016.
[34] T.L. Greenough, Representation and enumeration of interval orders, Ph.D., Dartmouth College, 1976.
[35] R. Guerraoui, E. Ruppert, A paradox of eventual linearizability in shared memory, in: Proc. of PODC '14, 2014.
[36] M. Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. 13 (1) (1991) 124–149.
[37] M.P. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.
[38] A. Imine, P. Molli, G. Oster, M. Rusinowitch, Proving correctness of transformation functions in real-time groupware, in: Proc. of ECSCW '03, Springer, 2003, pp. 277–293.
[39] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin, Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web, in: Proc. of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, 1997, pp. 654–663.
[40] M. Kokociński, T. Kobus, P.T. Wojciechowski, On mixing eventual and strong consistency: acute cloud types, IEEE Trans. Parallel Distrib. Syst. 33 (6) (2022) 1338–1356.
[41] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, Oper. Syst. Rev. 44 (2) (2010) 35–40.
[42] L. Lamport, Proving the correctness of multiprocess programs, IEEE Trans. Softw. Eng. SE-3 (2) (1977) 125–143.
[43] L. Lamport, The part-time parliament, ACM Trans. Comput. Syst. 16 (2) (1998).
[44] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (Jul. 1978).
[45] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Comput. C-28 (9) (Sep. 1979).
[46] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, R. Rodrigues, Making geo-replicated systems fast as possible, consistent when necessary, in: Proc. of OSDI '12, 2012.
[47] C. Li, N. Preguiça, R. Rodrigues, Fine-Grained Consistency for Geo-Replicated Systems, Proc. of USENIX ATC, vol. '18, 2018.
[48] S. Liu, M. Rahman, S. Skeirik, I. Gupta, J. Meseguer, Formal modeling and analysis of Cassandra in Maude, in: Formal Methods and Software Engineering, in: LNCS, vol. 8829, 2014, pp. 332–347.
[49] Y. Liu, J. Parker, P. Redmond, L. Kuper, M. Hicks, N. Vazou, Verifying replicated data types with typeclass refinements in liquid Haskell, in: Proc. of OOPSLA '20 4 (2020) 1–30.
[50] P. Mahajan, Highly available storage with minimal trust, Ph.D., University of Texas at Austin, USA, May 2012.
[51] P. Mahajan, L. Alvisi, M. Dahlin, Consistency, availability, and convergence, Technical Report TR-11-22, University of Texas at Austin, USA, May 2011.
[52] J. Meza, T. Xu, K. Veeraraghavan, O. Mutlu, A large scale study of data center network reliability, in: Proc. of IMC '18, 2018.
[53] Netflix, Netflix dynomite distributed dynamo layer, https://github.com/Netflix/dynomite.
[54] G. Oster, P. Urso, P. Molli, A. Imine, Proving correctness of transformation functions in collaborative editing systems, Tech. rep., INRIA, 2005.
[55] C.H. Papadimitriou, The serializability of concurrent database updates, J. ACM 26 (4) (1979).
[56] I.B. Peng, M.B. Gokhale, E.W. Green, System evaluation of the Intel Optane byte-addressable NVM, in: Proc. of MEMSYS '19, 2019, pp. 304–315.
[57] N.M. Preguiça, C. Baquero, P.S. Almeida, V. Fonte, R. Gonçalves, Dotted version vectors: logical clocks for optimistic replication, CoRR, arXiv:1011.5808, 2010.
[58] H.-G. Roh, M. Jeon, J.-S. Kim, J. Lee, Replicated abstract data types: building blocks for collaborative applications, J. Parallel Distrib. Comput. 71 (3) (2011) 354–368.
[59] P.M. Rondon, M. Kawaguci, R. Jhala, Liquid types, in: Proc. of PLDI '08, 2008, pp. 159–169.
[60] A. Rudoff, Persistent memory programming, Login 42 (2) (2017).
[61] M. Serafini, D. Dobre, M. Majuntke, P. Bokor, N. Suri, Eventually linearizable shared objects, in: Proc. of PODC '10, 2010.
[62] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: Proc. of SSS '11, 2011.
[63] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, A comprehensive study of convergent and commutative replicated data types, Tech. Rep. 7506, Inria–Centre Paris-Rocquencourt; INRIA, 2011.
[64] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, C. Hauser, Managing update conflicts in Bayou, a weakly connected replicated storage system, in: Proc. of SOSP '95, 1995.
[65] D.B. Terry, A.J. Demers, K. Petersen, M. Spreitzer, M. Theimer, B.W. Welch, Session guarantees for weakly consistent replicated data, in: Proc. of PDIS '94, 1994.
[66] R.H. Thomas, A majority consensus approach to concurrency control for multiple copy databases, ACM Trans. Database Syst. 4 (2) (1979) 180–209.
[67] D. Turner, K. Levchenko, J.C. Mogul, S. Savage, A.C. Snoeren, On failure in managed enterprise networks, HP Labs HPL-2012-101, 2012.
[68] N. Vazou, E.L. Seidel, R. Jhala, D. Vytiniotis, S. Peyton-Jones, Refinement types for Haskell, in: Proc. of ICFP '14, 2014, pp. 269–282.
[69] P. Viotti, M. Vukolić, Consistency in non-transactional distributed storage systems, ACM Comput. Surv. 49 (1) (2016) 19.
[70] W. Vogels, Eventually consistent, Commun. ACM 52 (1) (Jan. 2009).
[71] C. Wang, C. Enea, S.O. Mutluergil, G. Petri, Replication-aware linearizability, in: Proc. of PLDI '19, 2019.
[72] M. Wenzel, L.C. Paulson, T. Nipkow, The Isabelle framework, in: Proc. of TPHOLs '08, Springer, 2008, pp. 33–38.
[73] P.T. Wojciechowski, T. Kobus, M. Kokociński, State-machine and deferred-update replication: analysis and comparison, IEEE Trans. Parallel Distrib. Syst. 28 (3) (2017) 891–904.

Maciej Kokociński received a Ph.D. degree in computer science from Poznan University of Technology, Poland. His research interests include fault-tolerant distributed systems, concurrent data structures and persistent memory. Currently, he works as a principal engineer with Huawei Warsaw Research Center. He is also an assistant professor at the Institute of Computing Science at Poznan University of Technology.

Tadeusz Kobus received a Ph.D. degree in computer science from Poznan University of Technology. He conducts research in the areas of fault-tolerant distributed systems, heterogeneous memory systems, and concurrent data structures. Currently, he works as a principal engineer with Huawei Warsaw Research Center. He is also an assistant professor at the Institute of Computing Science at Poznan University of Technology.

Paweł T. Wojciechowski received the Habilitation degree from Poznan University of Technology, Poland, in 2008, and the Ph.D. degree in computer science from the University of Cambridge, in 2000. He was a postdoctoral researcher with the School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, from 2001 to 2005, and with the University of Cambridge, in 2001. He is currently an Associate Professor with the Institute of Computing Science, Poznan University of Technology. His research interests span topics in concurrency, distributed computing, and programming languages.

Appendix A. Extension to partial replication

Let us discuss a possible extension of the system model from Section 3 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* [73, 1]. However, care needs to be taken with partial replication, because in each ensemble 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 number of faulty replicas allowed.

Appendix B. Additional proofs

Theorem 2. *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* $\mathrm{PABEC}(\mathcal{F}) \wedge \mathrm{ERYW}$ *or* $\mathrm{PABEC}(\mathcal{F}) \wedge \mathrm{EMR}$ *for stateless mobile clients.*

Proof. The proof is similar to the one for Theorem 1. Consider a history H as outlined in the proof of Theorem 1, 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 $e_i, e_j \in E$, $i < j \Leftrightarrow e_i \xrightarrow{\mathsf{so}} e_j$. Note that $\mathsf{so}(e_0) = E \setminus \{e_0\}$. If the system satisfies $\mathrm{PABEC}(\mathcal{F})$ with either of the two session guarantees, then there exists an abstract execution $A = (H, \mathsf{vis}, \mathsf{ar})$, that satisfies $\mathrm{PABEC}(\mathcal{F})$ with the respective session guarantee.

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

If $A \models$ ERYW, then there exists some $k \geq 1$, such that for each $i \geq k$, $e_0 \xrightarrow{\mathsf{vis}} e_i$ (e_0 has to be eventually visible, as the set $\mathsf{so}(e_0) \cap \overline{\mathsf{vis}}^{-1}(e_0)$ has to be finite, and all operations are issued within the same session).

If $A \models$ EMR, then there exists some $k \geq k'$, such that for each $i \geq k$, $e_0 \xrightarrow{\mathsf{vis}} e_i$ (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 $e_i \in \mathsf{so}(e_{i'})$).

Now, we can conclude (similarly to the way we did in the proof of Theorem 1) that there is infinitely many events e_j, such that $e_0 \xrightarrow{\mathsf{vis}} e_j$, and that their return value v' is different than v (the return value for e_i such that $e_0 \not\xrightarrow{\mathsf{vis}} e_i$).

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 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. ■

Theorem 4. *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 1, Theorem 2 and Theorem 3, but is more complex. We consider a history H as in the proof of Theorem 1, but with $c_1 = c_2$, and R_1 crashing after serving the response for op_u. However, contrary to the proof of Theorem 3, 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 $H' = (E', \mathsf{op}', \mathsf{rval}', \mathsf{rb}', \mathsf{so}', \mathsf{sp}', \mathsf{crash}')$, 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 1. Note that R_1 does not crash in H', so e_0 cannot be a phantom operation. Since $crash(e_i) = \mathsf{false}$ 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 1, 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 2, 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 1, 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. ■

Theorem 6. *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* $\text{CABEC}(\mathcal{F}) \wedge \text{ERYW}$ *or* $\text{CABEC}(\mathcal{F}) \wedge \text{EMR}$ *for stateless clients.*

Proof. Just as in Theorem 5 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 4 for the CS model. ■

Theorem 8. *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 2, which shows a pseudocode of a generic protocol for an arbitrary $\mathcal{F}$. Note that, Algorithm 2 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 clk (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 clk 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 $(\mathsf{o}, \mathsf{o}')$ 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.clk is chosen so that it is larger than the clk 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 $H = (E, \mathsf{op}, \mathsf{rval}, \mathsf{rb}, \mathsf{so}, \mathsf{sp}, \mathsf{crash})$) of a system that implements Algorithm 2, we need to show that there exists an abstract execution $A = (H, \mathsf{vis}, \mathsf{ar})$, such that $A \models \mathrm{FABEC}(\mathcal{F})$. 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 2 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 $\mathsf{pre}(e)$ and $\mathsf{post}(e)$ the volatile state of the replica, respectively, just before the execution of e, and just after the execution of e. Similarly, by $\mathsf{sspre}(e)$

and $\mathsf{sspost}(e)$ we denote the contents of stable storage of the replica executing e, before and after the execution of e. Note that $\mathsf{pre}(e) = \mathsf{sspre}(e)$, but not always $\mathsf{post}(e) = \mathsf{sspost}(e)$ (e.g., when the replica executing e crashes before completing the execution). Moreover, $\mathsf{sspre}(e).\mathsf{operations} \subseteq \mathsf{sspost}(e).\mathsf{operations}$ and $\mathsf{sspre}(e).\mathsf{visible} \subseteq \mathsf{sspost}(e).\mathsf{visible}$. Similarly, for any two events $e, e' \in E$ executed on the same replica in that order, $\mathsf{sspre}(e').\mathsf{operations} \subseteq \mathsf{sspost}(e).\mathsf{operations}$ and $\mathsf{sspre}(e').\mathsf{visible} \subseteq \mathsf{sspost}(e).\mathsf{visible}$.

To construct A, for any $a, b \in E$, we let $a \xrightarrow{\mathsf{ar}} b \Leftrightarrow a < b$ and $a \xrightarrow{\mathsf{vis}} b \Leftrightarrow a \in \mathsf{pre}(b).\mathsf{operations}$. 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 $\mathsf{rval}(e) \neq \nabla$ it means that $e \in \mathsf{sspost}(e).\mathsf{operations}$. Similarly, if $\mathsf{vis}(e) \neq \emptyset$, and thus there exists $e' \in E$, such that $e \xrightarrow{\mathsf{vis}} e'$, 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 $e \in \mathsf{sspost}(e).\mathsf{operations}$.

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 $\mathsf{op}(e)$ locally or received e in some UPDATE or RECOVERY message. In either case, R_i will broadcast e as part of its $\mathsf{operations}$ 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 $\mathsf{operations}$ set (which is persisted on stable storage), R_j will add e to its $\mathsf{operations}$ set and write it to stable storage.

For any partition $p \in E/\approx_{\mathsf{sp}}$, any $e, e' \in E$, such that $e' \in p \wedge e \in \mathsf{pre}(e').\mathsf{operations} \wedge \neg \mathit{crash}(e')$, it holds that, there is only a finite number of events $e'' \in p$, such that $e \notin \mathsf{pre}(e'').\mathsf{operations}$. Therefore, $|p \cap \overline{\mathsf{vis}}(e)| < \infty$. On the other hand, for an event $e \in E$, for which there does not exist such an event $e' \in p$, naturally $\big(p \cap \mathsf{crash}^{-1}(\mathsf{false}) \cap (\mathsf{vis}(e) \cup \{e\}) = \emptyset\big)$, since $\big(p \cap \mathsf{crash}^{-1}(\mathsf{false}) = \emptyset\big)$. 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, $a \xrightarrow{\mathsf{rb}} b$. 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 $a \in \mathsf{pre}(b).\mathsf{operations}$. 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, $\mathsf{vis} \subseteq \mathsf{rb}$. Additionally, $\mathsf{so} \subseteq \mathsf{rb}$, by well-formedness of a history. Therefore, $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^+ \subseteq \mathsf{rb}$, and since rb is a partial order, hb is acyclic, and $A \models \text{NCC}$.

Finally, we turn our attention to $\text{RVAL}(\mathcal{F})$. The return value for each not pending event $e \in E$ is computed using the function $\mathcal{F}$ itself. We need to show that the output $C' = (E', \mathsf{op}', \mathsf{vis}', \mathsf{ar}')$ of the function $\mathsf{makeContext}$ is isomorphic with $C'' = \mathsf{context}(A, e) = (E'', \mathsf{op}, \mathsf{vis}, \mathsf{ar})$. Firstly, by definition $\mathsf{vis}^{-1}(e) = \mathsf{pre}(e).\mathsf{operations}$, thus $E' = E''$. Secondly, for each $e' \in E'$, $\mathsf{opFun}(e') = e'.op$, and $e'.op = \mathsf{op}(e')$. Thus, for each $e' \in E'$, $\mathsf{op}'(e) = \mathsf{op}(e)$. Thirdly, for any three events $a, b, c \in E$, if $a \in \mathsf{pre}(b).\mathsf{operations} \wedge a, b \in \mathsf{pre}(c).\mathsf{operations}$, then $(a, b) \in \mathsf{pre}(c).\mathsf{visible}$, because the sets $\mathsf{operations}$ and $\mathsf{visible}$ are always modified, persisted and disseminated together atomically. Thus, for any two $a, b \in E'$, such that $a \xrightarrow{\mathsf{vis}} b$, $(a, b) \in \mathsf{pre}(e).\mathsf{visible}$, which means that $\mathsf{vis}' = \mathsf{vis}|_{E'}$. Fourthly, $(a, b) \in \mathsf{ar}' \Leftrightarrow a, b \in E' \wedge a < b$, and $(a, b) \in \mathsf{ar} \Leftrightarrow a, b \in E \wedge a < b$. Thus, $\mathsf{ar}' = \mathsf{ar}|_{E'}$. 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 $A \models \text{RVAL}(\mathcal{F})$, $A \models \text{FABEC}(\mathcal{F})$.

■

Algorithm 2 Protocol implementing replicated data type $\mathcal{F}$, for replica R_i identified by rid

```
1: struct OpRec(clk : integer, rid : integer, op : ops(F))
2: operator <(o : OpRec, o′ : OpRec)
3:     return (o.clk < o′.clk) ∨ (o.clk = o′.clk ∧ 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: operation Perform(op : ops(F))
12:     var rval = F(op, makeContext(operations, visible))
13:     var o = OpRec(max(operations).clk + 1, rid, 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
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
```