Trade-offs in Replicated Systems

Rachid Guerraoui Matej Pavlovic Dragos-Adrian Seredinschi
{*firstname*}.{*lastname*}@epfl.ch
EPFL

Abstract

Replicated systems provide the foundation for most of today's large-scale services. Engineering such replicated system is an onerous task. The first—and often foremost—step in this task is to establish an appropriate set of design goals, such as availability or performance, which should synthesize all the underlying system properties. Mixing design goals, however, is fraught with dangers, given that many properties are antagonistic and fundamental trade-offs exist among them. Navigating the harsh landscape of trade-offs is difficult because these formulations use different notations and system models, so it is hard to get an all-encompassing understanding of the state of the art in this area.

In this paper, we address this difficulty by providing a systematic overview of the most relevant trade-offs involved in building replicated systems. Starting from the well-known FLP result, we follow a long line of research and investigate different trade-offs, assembling a coherent perspective of these results. Among others, we consider trade-offs which examine the complex interactions between properties such as consistency, availability, low latency, partition-tolerance, churn, scalability, and visibility latency.

1 Introduction

On-line services are continuously growing in scale and are being used by more and more users worldwide. Given their unprecedented size, these services must be deployed in the form of scalable distributed systems. Examples include social networks, search and aggregation, collaborative tools, entertainment, storage and backup, and many more. Informally, to achieve success, such services must, first and foremost, provide a smooth user experience. This requirement means that these services should ensure correct, uninterrupted operation, with fast response times. Technically, this translates to the important design goal of *high availability*, which necessarily implies tolerance to faults, including network and replica failures, and low latency of user requests.

Towards achieving this design goal, the common approach is to use geo-replication, i.e., to replicate both data and computation in different geographic regions. As such, the system may isolate faults and ensure that users in unaffected regions of the world can continue to access the service with low latency. On the surface, it might seem like geo-replication successfully solves all the issues faced by these systems. Unfortunately, this is not the case. The truth is that replication brings up the burden of ensuring *data consistency* and there is a fundamental friction between consistency and availability. In such a system, trying to guarantee high availability and uphold low latency without sacrificing strong consistency is, in fact, a hopeless endeavor [1, 12, 20].

These two properties—availability and consistency—are not the only attributes which weigh upon a replicated system; they are merely the tip of the iceberg. Depending on the specifics of the higher-level application,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

many other properties are equally or even more important. Latency, throughput, partition-tolerance, churn-tolerance, scalability, bandwidth usage, and energy-efficiency, are examples of other relevant properties of a replicated system. A comprehensive overview of how all these properties blend together in real systems is impossible, given that most of them have a continuous scale and even their precise definitions vary throughout the literature. Some of these properties can be achieved at the same time, while some are inherently antagonistic. There is a long line of research which focuses on how the most important design goals interact. Even a simple replicated system with a select few design goals could encounter inescapable conflicts between these goals, and from these conflicts, an array of trade-offs arise. This article outlines the most important trade-offs, puts them in perspective, and discusses the limits involved in engineering replicated systems.

We structure our investigation on trade-offs to start as general as possible, from the most wide-ranging trade-offs. We follow the research trends over the years, as they pursue the demands of real-world applications, by progressively narrowing down the context. This directs our investigation towards several specialized trade-offs, many of which can be seen as refinements of earlier results.

1.1 Overview on Trade-offs

In the broader context of distributed systems, the properties of any algorithm can be separated into two classes: *safety* properties and *liveness* properties [26]. Informally, safety properties convey the requirement that nothing bad should ever occur, whereas liveness properties state that something good should (eventually) happen. For instance, availability of a storage service is a liveness property, while the consistency guarantees of this service are a safety property. In practice, safety has to do with the correctness of a system, while liveness refers to a system's performance (i.e., the service should deliver results and make some progress).

The separation of distributed system properties in these two categories is relevant to our discussion as most, if not all, of the trade-offs encountered in distributed systems are, at a sufficiently high level of abstraction, safety versus liveness trade-offs. This is intuitive also from a practical perspective: the stronger the correctness guarantees a system should provide, the harder it is to implement those guarantees. The price to pay is performance, since implementing stringent correctness guarantees entails bigger complexity and resource overheads, compared to more lenient guarantees.

The canonical example of a result which puts safety and liveness at odds with each other is the FLP impossibility [18], which is the first trade-off we address. This result applies specifically to the *consensus* problem in an asynchronous system, whereby replicas need to reach agreement on a single value, called decision, among multiple proposals. Briefly, FLP specifies that either safety (no process takes a wrong decision) or liveness (all correct processes eventually reach a decision) of consensus must be sacrificed if processes are prone to failure. Consensus is at the center of the state machine replication (SMR) approach, which is a general technique suitable for replicating any deterministic service [29].

After FLP and general SMR, we narrow down the context to that of a storage service, with operations restricted to `read` and `write`. In other words, in terms of semantics, we narrow our focus from a general setting towards storage-oriented services, which are an important class of replicated systems. We first examine the CAP theorem [12, 20] and then we discuss a formulation, called PACELC [1], which descends directly from CAP. In discussing these two results, the spotlight is mainly on strong forms of consistency, and how this safety property impacts liveness. Next, we confine our discussion to a specific form of consistency, namely causal consistency. Therefore, we aim our attention at the CAC result [29] and the throughput versus visibility—or shortly, TV—trade-off [6, 11].

Then, we shift the discussion to consider dynamic storage systems, where nodes may join and leave, i.e., churn, at fast rates. In such systems, churn creates further friction among the service properties, and gives rise to a trade-off between scalability, allowable churn rate, and availability, or SCA [9].

Finally, we discuss a generalization of this last result. We switch back from storage-oriented semantics to general-purpose semantics and propose a new trade-off which is applicable to any replicated system subject

	FLP (§3)	CAP (§4.1)	PACELC (§4.2)	CAC (§5.1)	TV (§5.2)	SCA (§6.1)	RCR (§6.2)
Semantics	General	← - - - - - - - - - - - - - - - Storage - - - - - - - - - - →					General
Membership	← - - - - - - - - - - - - - - Static - - - - - - - - - - - - - - →					← - - - - Dynamic - - →	
Replication	← - - - - - - - - - - - - - - Full - - - - - - - - - - - - - - →					← - - - - - Partial - - - →	

Table 1: Perspective on the discussed trade-offs.

to churn (§6.2). We do so by introducing the notion of *reconciliation mechanism*, which allows us to capture the conflicting interaction between churn and robustness (i.e., fault-tolerance) in such a system. Any system which is subject to churn has a reconciliation mechanism; reconciliation can take various forms, such as a load-balancing scheme, replica reconfiguration [21] or regeneration [35], or the cuckoo rule [4]. Reconciliation is triggered by churn events and is necessary to uphold robustness in the face of membership fluctuations. In other words, this mechanism affects both churn and robustness: for instance, it may constrain the former by curbing the churn rate; likewise, reconciliation influences the latter by ensuring either strict measures or best-effort measures to preserve robustness. Our findings point to the existence of a trade-off between reconciliation, churn, and robustness (RCR). The specifics of the reconciliation mechanism will tip the scale of this trade-off either in favor of churn or in favor of robustness.

In Table 1, we give a concise perspective on all the trade-offs we consider. We use three coarse-grained dimensions to trace these results: semantics, membership, and replication. Before diving into the main study of these trade-offs, however, we introduce some preliminary notions (§2). The first trade-off we present is the FLP result (§3), after which we move on to results on strongly consistent storage systems, namely CAP and PACELC (§4). We then study causal consistency with its associated trade-offs (§5). We also take dynamic membership into account, by investigating what happens in the presence of churn and propose a general trade-off concerning churn (§6). Finally, we conclude with a summary of the presented trade-offs (§7).

2 Preliminaries

In this section, we give the necessary background and prepare the ground for our study of trade-offs. Given that these trade-offs span across multiple system models, we describe here the minimal variations on these models which are necessary for our discussion. We introduce a general model and then present a few refinements on it, which become relevant later on in our discussion, as we report on subsequent trade-offs.

Throughout the whole article, we assume a service running as a distributed system. Initially, we consider the service state being replicated at all the nodes of this system, called *replicas*. In other words, we consider a *full* replication model, with each replica storing a complete copy of the service state. The system is asynchronous, which means that no assumptions can be made regarding relative computation or communication speeds. A common aspect of all the trade-offs we consider is that neither of them explicitly assumes synchrony. This is a pragmatic choice, especially relevant for today's online services, since the prevailing approach—as highlighted above (§1)—is to employ geographic replication, where achieving synchrony is remarkably difficult [7].

In terms of semantics, we model our service in two iterations. First, we represent the service as a general state machine, which is relevant for the opening result, the FLP impossibility. After that, all subsequent results (except RCR, §6.2) model the service as a key-value store—or alternatively, a set of shared registers [27]. Each register is accessible by a unique key and stores a data *object*, supporting `read` and `write` semantics on that object.

The set of replicas composing the distributed system is also a moving target. This is another aspect which we refine as we go along: at the beginning, our discussion centers on trade-offs that generally apply even if the replica set (i.e., membership) is *static*. Later on, we examine trade-offs which consider *dynamic* (i.e., changing in time) membership and discuss the friction between this dynamicity and some other system properties. Once

we consider that the set of replicas is dynamic, we also drop the full replication assumption. We do so to adhere to the system model of these dynamic systems, which employ *partial* replication, as is common in file sharing or peer-to-peer systems. This distinction between partial and full replication, however, is not crucial, and the same trade-off would still apply under either of these replication models.

3 FLP

As we stated earlier, FLP[1] puts safety at odds with liveness, by proving the impossibility of a deterministic algorithm solving consensus in an asynchronous system if even a single replica can crash [18]. The actual impossibility result is more precise in its definition, as we shall see; this high-level perspective, however, is instructive, and helps us tie this result together with all the other trade-offs. In this section, we first investigate this impossibility result. Then, we veer the discussion towards a practical standpoint: we study what are the implications FLP has for system designers, and examine a trade-off arising from this result.

The main insight underlying this result is the following: in an asynchronous system, message delays can be arbitrarily long, and thus it is impossible to distinguish a crashed replica from a slow one. Consider a system where all replicas are fast, except one of them which is particularly slow. On one hand, if the slow replica is falsely suspected to have crashed, it could lead to disagreement: the fast replicas will agree among themselves, while ignoring the suspected replica ,violating safety. On the other hand, if the fast replicas do not suspect the slow replica to have crashed and simply wait for it, then they run the risk of waiting indefinitely: owing to the nature of the asynchronous system, the slow replica might—in fact—have crashed, and the other replicas never reach a decision, i.e., the algorithm does not terminate, breaking liveness.

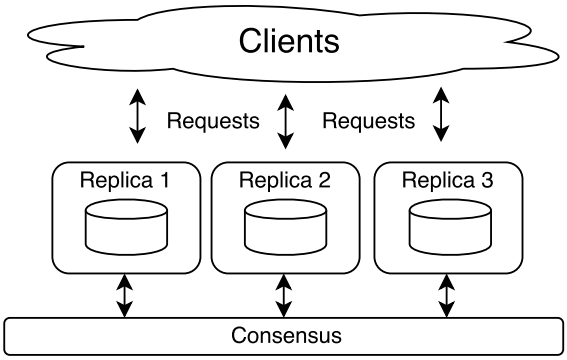

Figure 1: State machine replication: replicas use a consensus protocol to agree on a common order on operations, e.g., client requests.

FLP is an important stepping stone in our understanding of replicated systems because consensus plays a central role in *state machine replication* [29], which, in turn, is a basic and general method towards ensuring high-availability and tolerating failures. In state machine replication (SMR), the service is modelled as a deterministic state machine. All replicas hold a copy of this state machine (i.e., employing full replication) and use consensus to agree on the order of operations which they apply on the state machine (see Figure 1). This agreement step is critical, as it ensures that replicas keep their states synchronized; furthermore, the operations must be deterministic, otherwise the state at replicas could diverge. Consequently, even if a replica fails, it will not incur service downtime, since the other replicas can continue operation.

For system designers, this impossibility result has ample implications, partly due to the generality of the consensus problem, and partly because of the widespread use of SMR. Namely, other important problems in replicated systems, such as atomic broadcast or group membership, can be reduced to consensus, thus extending the reach of this result [23]. Since SMR can be used to replicate *any* deterministic service—achieving high availability for that service—it is an essential technique in replicated systems. In practice, FLP highlights the chief importance of synchrony assumptions in distributed systems, forcing engineers to reason about worst-case scenarios and to understand what conditions are necessary to be able to solve consensus.

To reach an alternative perspective on FLP, we start from our earlier observation that SMR can model *any* deterministic service. This generality of SMR is both an asset and a burden: on one hand, SMR lends itself to almost universal application; on the other hand, some services do not necessarily need this generality, and benefits

[1]The name of this result is derived from the last names of its authors: Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson.

may be reaped by restricting the allowed semantics. A notable example is storage services supporting read/write operations on single objects. These services, even with the strongest correctness condition (linearizability [25]), can be implemented despite asynchrony or failures [3]. The intuition why replicated storage is easier to implement than a general replicated state machine is that *read* and *write* are very simple operations. In particular, their outcome does not in any way depend on the previous state of the accessed object, and thus atomic *read-modify-write* behavior is not possible. Implementing an atomic transaction, for example, is not achievable using only independent read and write operations. Realizing the weaker semantics of read/write objects, we can regard FLP as a trade-off between service semantics, synchrony assumptions, and fault tolerance. Hence, in practice, we can choose to trade a certain side of the trade-off (e.g., service semantics) to the benefit of the other (e.g., synchrony assumptions).

Briefly, by lessening the requirements on semantics, a service such as a read/write storage can achieve fault-tolerance despite asynchrony. Fortunately, many applications tolerate weakened semantics and do not require a full read-modify-write model as SMR offers. For example, consider a website which recommends movies to watch based on a user's rating history. It is preferable that the website loads fast and works despite asynchrony and failures, even though it might give imperfect recommendations. It is not an issue if the recommended titles do not consider the entire rating history or the newest arrivals. For such a system, it is wise to give up read-modify-write semantics to the benefit of better performance. This subtle interplay between the semantics of a storage service and other properties is the subject of the next trade-offs which we address.

4 Strong Consistency in Replicated Storage Systems

The FLP result states that, in the given system model, replicating a general state machine is not possible. FLP does not apply, however, to storage systems with simple read/write objects, as these systems do not require solving consensus. Hence, the following question ensues: what can we guarantee in terms of storage system properties? The CAP theorem [12, 20] sheds light on this matter, stating that there is a fundamental trade-off between strong consistency, availability, and partition-tolerance of a replicated storage system. Namely, it demonstrates the impossibility of achieving all three properties at the same time. CAP was first stated by Eric Brewer [12] as a conjecture, and later proven by Gilbert and Lynch [20].

4.1 CAP: Consistency, Availability, and Partition-Tolerance

The main idea behind this result is that, in order to implement a replicated read/write object with strong consistency guarantees, communication among the replicas is necessary. If some replicas become isolated due to a network partition, upon receiving a request, these replicas have two choices: either (1) to return a value without coordinating with others, possibly violating consistency, or (2) to return no value at all, violating availability. Thus, to achieve partition-tolerance, one must sacrifice either consistency or availability. Alternatively, we can also view CAP through the lens of a safety versus liveness trade-off, because strong consistency is a classic safety property and availability is a typical liveness property.

In practice, this trade-off lies at the root of the distinction between ACID [24] and BASE [30] systems. ACID denotes a set of four properties: atomicity, consistency, isolation, and durability; it represents the gold standard in terms of transactional systems properties. Most of the traditional database management systems are ACID, offering strong consistency (isolation) guarantees. This is convenient for application developers since they are exposed to clear semantics, and do not need any explicit mechanism to deal with inconsistencies. ACID systems, however, must sacrifice availability when network partitions arise.

BASE systems (basically-available, soft state, eventually consistent), such as distributed key/value stores, are willing to trade in some data consistency guarantees in order to increase system availability and performance [30]. Intuitively, when a partition isolates a replica from the rest of the system, that replica—and all others as well—

may continue to serve client requests without employing coordination. Due to absence of coordination, the system sacrifices consistency and burdens the client with the task of dealing with inconsistencies. Both ACID and BASE systems are useful in specific scenarios, and it is actually a common practice to blend both of these models when building real-world application stacks [2].

4.2 PACELC: a Refinement of CAP

The FLP and CAP theorems are the capstones of any modern distributed system. The impossibilities which these two theorems cover are general and far-reaching. FLP proves that asynchrony and failures are a dangerous combination, prohibiting a deterministic consensus algorithm; CAP, in the same vein, teaches us that when a partition occurs in a replicated storage system, we must trade between strong consistency and availability.

PACELC [1] descends directly from CAP and suggests that even in the absence of partitions, i.e., during healthy operation, we still face a trade-off, between consistency and latency. Briefly, PACELC captures the following double trade-off: if partitions occur, then trade between availability and consistency; else trade between latency and consistency. The insight behind PACELC is that even during healthy operation, the coordination required for ensuring strong consistency among replicas results in an increased request latency.

Consider, for instance, a geo-replicated storage system with replicas spread around the globe. If this system guarantees strong consistency, then it requires synchronous inter-replica coordination, resulting in a latency penalty upwards of 100ms. In contrast, such a system could satisfy eventual consistency [15] without any synchronous coordination, and thus zero latency overhead. Low latency is a critical goal in many systems, especially so in users-facing web services, since humans perceive even a slight deterioration in latency on the order of hundreds of milliseconds [32]. Such latency numbers are common if a service uses synchronous geo-replication, given the fundamental limitation on communication speed imposed by the speed of light.

The PACELC formulation recognizes the importance of low latency in modern services, and weaves this dimension into the well-known CAP trade-off. Thus, PACELC encompasses two important trade-offs: one between consistency and availability, and another between consistency and low latency. The former is inherited directly from CAP, with the important observation that this trade-off is only relevant during partitions. The latter trade-off is relevant during normal-case operation: even in the absence of failures, as we observed in the example earlier, latency can reach hundreds of milliseconds if strong consistency is required, which can be prohibitive for some applications.

Conceptually, we can regard unavailability as high (unbounded) latency, which allows us to consolidate the PACELC formulation as a single trade-off. In this light, we see a general trade-off between consistency and latency. For a strongly consistent system, the latency depends on the coordination delay. In the extreme case of a partition—where coordination takes infinite time—the system is unavailable, with unbounded latency. Thus, to achieve low latency, one must sacrifice strong consistency in favor of weaker consistency models, which eschew the need for costly coordination. A similar latency penalty also arises in the context of consensus: even when a solution is possible (i.e., by assuming synchrony), such a solution nevertheless incurs high latency [19].

5 Causal Consistency

From the CAP impossibility result, the following question immediately follows: what is the strongest consistency that is achievable while maintaining availability and partition tolerance? The answer to this question is *causal consistency* [13]. In other words, no consistency model stronger than causal can be ensured while also satisfying availability during network partitions [29]. This model is weaker than strong consistency, but is useful in practice because it provides intuitive semantics: it ensures that, for any operation t, all the operations which causally precede t are guaranteed to take effect before t. Alternatively, causal consistency ensures three guarantees: (1) monotonic reads and writes, (2) read-your-writes, and (3) writes-follow-reads [13]. Indeed, recent work in storage systems embraces causal consistency as the sweet spot on the consistency spectrum [16].

5.1 CAC: Causal Consistency, Availability, and Convergence

The causal consistency model seems to definitively reconcile the antagonism between consistency and availability. This result, however, is not intuitive, because we can formally define even stronger models which are also available during partitions; for instance, one can define consistency models to comprise only safety properties. Such a definition provides a loophole: replicas do not need to ever coordinate, and thus it allows an implementation which is both available and consistent (stronger than causal) despite partitions. The flaw in these consistency models is that they are not useful in practice, since replicas can forever diverge. To model this aspect of usefulness, Mahajan et al. define a property called *convergence* [29].

At an abstract level, convergence captures the coordination, often called synchronization, among replicas. Coordination can be asynchronous (in the background, periodically or in batches), or synchronous (on the critical path of every operation), depending on the desired consistency level, and it is a necessary step towards achieving consistency. Put differently, convergence ensures that replicas (and clients) observe each other's updates. Some consistency protocols encompass a weak version of convergence, where coordination is asynchronous, i.e., if updates stop, then all replicas reach the same state, as in eventual consistency [15]. Other protocols, such as those guaranteeing linearizability, require a stronger variant of convergence, whereby replicas must coordinate synchronously and agree upon a total order on the sequence of updates.

Informally, the property of convergence prescribes that all replicas shall agree on a *common state*. This state should include the updates from all replicas, and, in the case of a causally consistent system, it should adhere to a partial order of operations that satisfies causality. Since the ordering is partial, this convergence property allows replicas to temporarily diverge, i.e., concurrent updates on the same object can lead to conflicting versions of that object. Such a convergence property thus allows for asynchronous coordination, while guaranteeing usefulness in the sense that the replicas eventually reach a common state, applying each other's updates in a causal order.

To sum up, we can regard this result of Mahajan et al. about causal consistency, availability, and convergence [29] as a refinement of CAP which also takes into account the notion of convergence. A system could support a consistency model which is formally stronger than causal, and also ensure high-availability (tolerate partitions), but for the price of sacrificing convergence. To maintain convergence (i.e., usefulness), causal is the strongest implementable consistency in a highly-available system [29]. In the following, we will inspect this notion of high availability in a stricter sense, and discuss causal consistency in slightly more depth.

A Closer Look at Causal Consistency:

The system model of the CAC result has an important aspect which we did not cover earlier. Specifically, the model in this trade-off makes no distinction between clients and system replicas, which means that every node in the system is both a replica of the service and a client of the service. This assumption has the veiled implication that clients are always connected to one of the replicas—a unique replica, to be precise. If, however, we choose to distinguish between the two roles of clients and replicas and draw a definite boundary between the system and its clients, then the argument for causal consistency is no longer watertight. In practice, indeed, clients are separate entities, which may lose communication to a system replica, and which, moreover, can switch from one replica to another. Such switching happens usually due to a load-balancing mechanism [2, 33].

In light of this observation, it is necessary to distinguish between two system models. First, if clients and replicas are embodied in a single, combined role, e.g., by being co-located on the same machine, then clients always access the system via the same node—their co-located replica. In such a case, we say that clients stick to the same replica, and the system provides *sticky availability* [5]. Broadly speaking, these are systems where the client is co-located with the service, e.g., in the same datacenter, rack, or machine; peer-to-peer systems also match this model, because every node is both a client and a server. In the second model, clients and replicas are distinct roles, under separate failure domains, which may be parted by network failures. In this model, clients are permitted to switch between different replicas, and we say that the system provides *high availability*. Figure 2 depicts the contrast between these two models.

The distinction between these two availability models is important because each model has its own limit in terms of strongest achievable consistency. Specifically, the causal consistency result we highlighted earlier applies to the sticky availability model. This means that clients may rely on always being able to access—i.e., stick to—the same replica. In a high availability model, this result does not hold anymore. If a client switches from a service replica to an alternative one, it could mean forsaking causal consistency: this can happen if the alternative replica lacks some of the client's updates, and thus it causes the breaking of the read-your-writes guarantee implied by causal consistency [13].

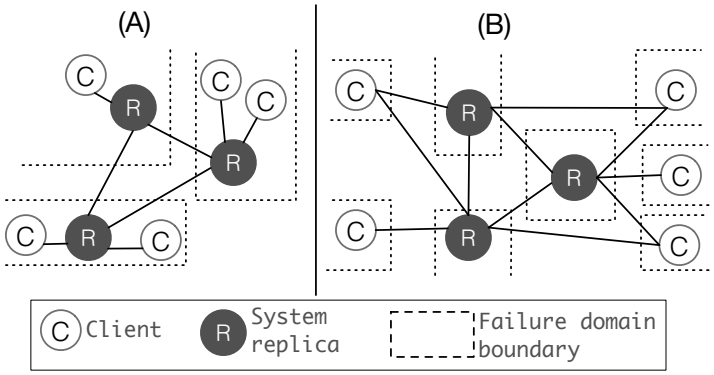

Figure 2: Contrast between a system model under sticky availability **(A)** and under high availability **(B)**. With sticky availability, clients `C` share the failure domain with one of the system replicas `R`.

Our belief is that the problem of finding the strongest consistency model achievable under high availability is very important; yet, to the best of our knowledge, this is still an open issue. At first glance, our investigation of this topic indicates that consistent prefix [34] is the answer. Under consistent prefix guarantees, a client always observes a totally ordered sequence of updates for every data object, but this sequence may lack some of the latest updates, i.e., it does not ensure freshness, and it does not capture causality constraints either. Consequently, compared to causal consistency, consistent prefix can be considered less useful in practice, since it provides neither read-your-writes nor monotonic reads across subsequent client operations. These two guarantees are important because they prohibit anomalous scenarios where the natural cause-effect relation between events is inverted, i.e., an effect may be observed to precede its respective cause [6], and such an inversion would contradict our expectations.

The consistent prefix model can be easily achieved, for instance, using a scheme where a primary replica establishes a total order on write operations and propagates this order asynchronously. The model is achievable under high availability because different replicas may reveal different states of each object. An analogy can be drawn between this consistency model and snapshot isolation [17], which is well-known in the database community, with the caveat that consistent prefix applies to read/write single-key operations [34]. In our future work, we plan to further examine this argument and also consider other alternative paths to an answer.

5.2 TV: the Throughput versus Visibility Trade-off in Causal Consistency Protocols

Despite evading the consistency/availability trade-off, causal consistency still runs into trade-offs of its own. The crux of the issues in causally-consistent systems are due to metadata management. In contrast to most other forms of consistency—such as linearizability, per-key sequential consistency, or eventual consistency—protocols for causal consistency rely on metadata to track causal dependencies between operations and establish a partial order among operations, an order which respects causality. This introduces two main difficulties: (1) each update operation generates some new metadata, and (2) this metadata has to propagate to every replica. The latter is particularly problematic and it arises because causal consistency entails full replication: every node replicates every object, otherwise availability would be sacrificed during partitions [6].

In this context, the trade-off is between the throughput of the storage system and visibility latency [6, 11]. Visibility latency denotes the delay of propagating and applying updates across nodes. Intuitively, update visibility is modeled by convergence (see §5.1), and it captures the delay of propagating both data and metadata [14]. In the case of metadata, a system can choose different tracking granularities. If metadata is fine-grained, then the system tracks causal dependencies among *individual* objects, so each replica may apply an incoming update

as soon as it satisfies the dependencies for that update. In this case, visibility latency is minimized, since only actual dependencies need to be taken into account. The throughput, however, is limited: fine-grained metadata is large and requires high bandwidth, which can lead to the metadata explosion problem [28]. If, on the other hand, metadata is coarse-grained, then the system tracks dependencies among *groups* of objects; this reduces the metadata size, requires less bandwidth, and allows the system to scale better in terms of throughput. The downside is that false dependencies may ensue if objects are grouped, causing the visibility latency to go up [11].

In order to circumvent this trade-off, storage systems have to limit the metadata size—but without introducing false dependencies, i.e., without grouping objects. One way to achieve this is through application-specific *explicit* dependencies, whereby every update includes just the actual application-level dependencies, and not the entire causal past. It is arguable, however, whether applications are equipped to handle such a mechanism, as it requires changes in the application logic [6]. Unless applications include mechanisms to track explicit dependencies among all possible combinations (and sequences) of operations, causality may be broken.

6 Dynamic systems

So far, we only considered systems with static membership. Many real-world systems, however, experience significant *churn*: nodes joining and leaving the system at run-time. This behavior is especially prominent in (but not limited to) peer-to-peer systems. There are many causes that make the membership of system dynamic: node failures and recovery, system scale-out, software or hardware maintenance, or simply the behavior of its users, all require the system to cope with membership changes. Dynamic membership, however, is another potential source of problems for system engineers, as it can interfere with other desirable system properties.

6.1 SCA: Scalability, Churn, and Availability

This trade-off, identified by Blake and Rodrigues [9], is in the context of highly available peer-to-peer storage systems. The formulation is as follows: as a system becomes more dynamic, i.e., as the set of replicas changes at a growing pace, the system becomes less scalable. This trade-off puts emphasis on the necessity of preserving data redundancy, which means that each data object should have, at all times, a minimum number of copies. As it turns out, if nodes churn at a very high rate, then it becomes impossible to maintain a minimum number of copies per each object, since the per-node bandwidth is fixed over time, posing a bottleneck. Hence, the system may only scale by either reducing availability—i.e., forsaking a minimal redundancy level—or by throttling churn—i.e., using a form of rate limiting mechanism to prevent too many nodes from joining and leaving.

The main idea behind this trade-off is based on a simple argument. A node joining the system needs to download the data it is about to store from another node. When a node departs from the system, other nodes have to create copies of the data which the departing node stored, by downloading it from available replicas. This is necessary in order to maintain an appropriate replication factor required for data availability.

To be more precise, assume a fixed amount of data to be stored by the system, while the network bandwidth allocated by each peer is also fixed. By *session time* we express the period of time which a peer spends as a member of the system. Then, higher churn translates to shorter session times, which, in turn, means that more data has to be moved around in order to maintain data availability. In the same vein, if we fix the number of peers belonging to the system at each time, then per-node bandwidth cost increases as churn goes up.

Using conservative estimates (e.g., considering only maintenance bandwidth and disregarding traffic generated by client requests) on the bandwidth needed to maintain data highly available, Blake and Rodrigues show that this bandwidth quickly becomes prohibitive with increasing churn [9]. For example, if each node provisions a bandwidth of 200KBps, then a million peers with session times of one month are necessary to store 1000TB of data with a replication factor of 20. Decreasing the session times to 1 day, only 50 TB could be maintained. Even employing optimizations such as erasure-coding or distinguishing between node departure and temporary

downtime does not make a significant difference. Thus, as a broad conclusion, highly dynamic peer-to-peer systems are unsuitable for large-scale data storage.

6.2 RCR: Robustness, Churn, and Reconciliation

In this section, we investigate how churn affects other properties, but in a more general framework, not restricted to storage, and thus extend the last trade-off we considered (SCA, §6.1). We switch back from a shared register context (with read/write semantics) to a general replication setting. We retain the other properties of the system model: *dynamic* membership and *partial* replication, since churn presumes a dynamic set of replicas. Our findings point to a trade-off between churn, robustness, and reconciliation. To explain this new trade-off, we begin with a short examination of our terms (robustness and reconciliation), and then we dive into some examples.

The common approach of applying replication in a large dynamic system is to partition the nodes into multiple *replication groups*, each group replicating a part of the system state [8, 21]. The rationale behind this separation is twofold: (1) performance, since full replication incurs substantial overhead and scales poorly; and (2) robustness, as each failure is isolated in its group, preventing a system-wide proliferation of faults. To avoid ambiguities, we distinguish between fault-tolerance (which denotes the ability of a replication group to withstand failures) and robustness (the property of the system as a whole of withstanding failures).

To ensure fault-tolerance, a replication group requires a minimal threshold of its members to be non-faulty and participate in the replication protocol [10]. In general, increasing the size of the replication group also increases its fault-tolerance, thus making the whole system more robust. Intuitively, more nodes are allowed to fail before violating the minimal threshold of non-faulty nodes. A small replication group is arguably less fault-tolerant, as fewer failures are sufficient to topple the threshold.

Our intuition is that the simple presence of churn may jeopardize robustness. The argument is as follows. Churn necessarily brings along fluctuations in the sizes of replication groups: a departing node triggers a decrease in the size of its group, while a joining node causes its hosting group to increase in size. Often, the system has limited or no control over the churning nodes. For example, in peer-to-peer systems, nodes may freely decide to join or leave at any time. Consider a small group of just three replicas storing important documents. If a replica leaves the system, the data remains present only on two other nodes; the departure or failure of one of those, under an asynchronous model, would potentially make the data unavailable, or even forever lost. To keep robustness undisturbed in the face of churn, replication groups must remain fault-tolerant despite join and leave events. This is the purpose of a *reconciliation* mechanism.

In the above example of three-way replication, reconciliation might work as follows: upon the departure of a replica, a replacement node has to take its role in the replication group, downloading the important document from the live replicas. This is essential to replenish groups and prevent them from dying out. In this case, reconciliation might involve actions such as: detecting departure, selecting a replacement node, reconfiguring the group to insert the replacement, and downloading the document (i.e., synchronizing states). Note that while this mechanism executes, the group should be locked, to prevent the remaining replicas from departing. In other words, in this case the reconciliation serializes the churn events, throttling the churn rate, to the benefit of robustness. Algorithm 1 summarizes the general pattern of group reconciliation upon churn events.

Algorithm 1 Basic interplay between churn, robustness, and reconciliation.

Require: `E` (a churn event), and G (a replication group).

```
1: G' = apply E to G
2: if robustness(G') < min_robustness then
3:     reconcile(G')
4: end if
```

To enable higher churn rates, we may speed up reconciliation by making it non-blocking. After the departure of a replica, such a best-effort reconciliation would run in the background, hoping that the replication group survives until reconciliation finishes. One may easily imagine generalizations of this approach, e.g. limiting the number of concurrent churn events or setting a lower bound on some measure of fault-tolerance beyond which

the group becomes locked. Generalizing even further, different reconciliation mechanisms may be considered to strike a trade-off between fast and robust reconciliation.

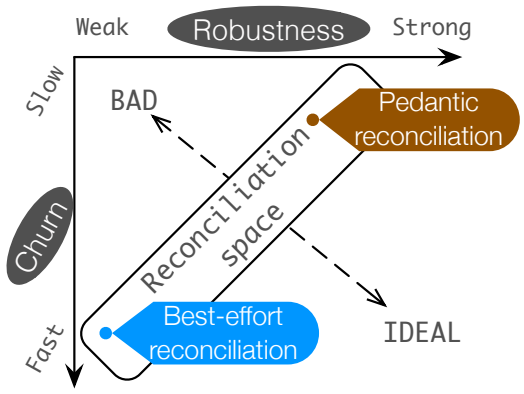

Figure 3: Robustness, churn, and reconciliation trade-off. Reconciliation mechanisms determine a trade-off between robustness (x-axis) and churn (y-axis).

We sketch this basic interaction between churn, robustness, and reconciliation in Figure 3. A system designer may pick a point on the churn-robustness scale by choosing a particular reconciliation implementation. Several notable reconciliation mechanisms are: simple data download (as in the argument for the SCA trade-off §6.1), group replenishing (as in the three-way replication we discussed earlier), or gossip-based membership (used to lazily propagate membership changes and keep the network of replicas connected, as in Dynamo [15]). Note that replenishing replication groups after node departures is not the only case where this principle applies. Counter-intuitively, even new nodes being added to a group may be threatening its fault-tolerance, such as in a Byzantine-tolerant system [22].

In general, a lightweight reconciliation mechanism favors churn: the system would not prevent or delay structural changes, and the reconciliation would be executed on a *best-effort* basis, e.g., as in the case of gossip-based systems. Such a system trades robustness for churn; specifically, in an gossip-based system, if the churn rate is too high, it may lead to cases when the network becomes disconnected, and nodes cannot reach each other, breaking robustness [36]. A *pedantic* reconciliation mechanism, on the other hand, favors robustness: all churn events become expensive operations—thus limiting the churn rate—but the system upholds robustness rigidly, as in the cuckoo rule [4].

7 Conclusion

We discussed the most important trade-offs involved in engineering modern replicated systems. We started with FLP, a fundamental result stating the impossibility of solving consensus in an asynchronous, failure-prone system. As we observed, consensus is a central building block for maintaining consistency among replicas in state machine replication (SMR), i.e., under a general semantics model. Accordingly, we can see FLP as a trade-off between strong consistency for general-purpose replication, synchrony assumptions, and fault tolerance.

We then turned our attention to storage systems with read/write semantics. Here, instead of the consensus problem, we switched to a data consistency problem. We examined the CAP theorem, which postulates that strong consistency, availability, and partition-tolerance are not achievable at the same time. We also discussed two refinements on CAP. The first, called PACELC, states that even in the absence of partitions, replicated storage systems must nevertheless face a trade-off between consistency and latency. The second, called CAC, defines the important notion of convergence. If we ignore convergence, we can devise consistency models that are rather strong while still permitting high availability, yet are useless in practice. Once we add convergence to the mix of system properties—as CAC proved—causal consistency is the strongest consistency model which escapes the CAP impossibility. Concerning causal consistency, we also discussed an essential trade-off between throughput and update visibility (TV).

We also investigated systems with dynamic membership, where churn plays an important role, and which use partial replication. We studied the trade-off between scalability, churn, and availability (SCA), which essentially states that systems susceptible to high churn—such as peer-to-peer—are not suitable for highly available large-scale storage. Finally, we presented a generalization of this trade-off, called RCR, showing that churn not only inhibits scalability, but the overall robustness of a system, in the general context of replicated systems.

References

[1] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, (2):37–42, 2012.

[2] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *HotOS XV*, 2015.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[4] B. Awerbuch, C. Scheideler. Towards a scalable and robust dht. *Theory of Computing Systems*, 45(2):234–260, 2009.

[5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *VLDB*, 7(3), 2013.

[6] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *ACM SoCC*, 2012.

[7] P. Bailis and K. Kingsbury. The network is reliable. *ACM Queue*, 12(7):20, 2014.

[8] C. E. Bezerra, F. Pedone, and R. V. Renesse. Scalable State-Machine Replication. In *IEEE DSN*, 2014.

[9] C. Blake, R. Rodrigues. High availability, Scalable Storage, Dynamic Peer Networks: Pick two. In *HotOS IX*, 2003.

[10] G. Bracha, S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, 1985.

[11] M. Bravo, L. Rodrigues, and P. Van Roy. Towards a scalable, distributed metadata service for causal consistency under partial geo-replication. In *Middleware Doctoral Symposium*, 2015.

[12] E. A. Brewer. Towards robust distributed systems (invited talk). In *PODC*, 2000.

[13] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (IEEE PDP)*, 2004.

[14] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo. In *SOSP*, 2007.

[16] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *ACM SoCC*, 2014.

[17] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[19] E. Gafni, R. Guerraoui, and B. Pochon. From a static impossibility to an adaptive lower bound: The complexity of early deciding set agreement. In *STOC*, 2005.

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

[21] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, T. Anderson. Scalable consistency in Scatter. In *SOSP*, 2011.

[22] R. Guerraoui, F. Huc, A.-M. Kermarrec. Highly dynamic distributed computing with byzantine failures. In *PODC'13*.

[23] R. Guerraoui and A. Schiper. Consensus: the Big Misunderstanding. In *IEEE FTDCS*, 1997.

[24] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 1983.

[25] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[26] L. Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2):125–143, 1977.

[27] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), 1978.

[28] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI*, 2013.

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

[30] D. Pritchett. Base: An acid alternative. *ACM Queue*, 2008.

[31] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[32] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. In *Velocity Web Performance and Operations Conference*, 2009.

[33] P. Shuff. Building a Billion User Load Balancer, 2013. SREcon15.

[34] D. Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[35] H. Yu and A. Vahdat. Consistent and automatic replica regeneration. *ACM Transactions on Storage*, 1(1):3–37, 2005.

[36] P. Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, 2012.